

TECHNICAL REPORT STUDENT PROJECT

I-MHERE 2012

GraphBT

Integrated Software Development Tool in Behavior Tree

Disusun oleh:

Agung Pratama

Ardi

Chairunnisa Atimas

Emerson Chan Simbolon

Ikhsanul Habibie

Nurul Qomariyah



Fakultas Ilmu Komputer Universitas Indonesia

2012

Table of Contents

Table of Contents	2
Abstract	4
I. Introduction	5
1.1 Background	5
1.2 Project Description.....	5
II. Literature.....	6
2. 1. Eclipse.....	6
2. 2. Framework	8
2. 3. Behavior Tree	9
2. 4. Features.....	9
III. Application Implementation	60
3. 1. Implementation Steps.....	60
3. 2. Finished Features	60
3. 3. Application Result	60
IV. Conclusion and Recommendation	78
4.1 Conclusion	78
4.2 Recommendation.....	78
V. Application Features	79
5.1 Conclusion	79
5.2 Recommendation.....	79
VI. Miscellaneous Implementation	80
Get XML representation of .BT file.....	80
GraphBT Tree Layout.....	80
Get BEModel	80
VII. Sample Project	81
7.1 Conclusion	81
7.2 Recommendation.....	81
Appendix	82
Appendix 1. TextBE Meta Model	82
Appendix 2. GraphBT Look.....	83
Appendix 3. GraphBT Meta model.....	87
Appendix 4. Guide for developer	87

Abstract

The objective of this project is to create integrated software development tool to manipulate Behavior Tree (BT) specification into an executable language. This project will use some existing research in Formal Method in Software Engineering (FMSE) laboratory: Symbolic Analysis Laboratory (SAL) model-checker, BT animation, and executable java code generator. The resulted Java code can be executed without further modification as long as the BT specification is correct. The tool reflects a new software methodology, where the programmer is not required to spend much time and effort in the implementation phase; hence also reduce the mistake in this phase.

I. Introduction

1.1 Background

This report is intended to provide technical documentation of the features and usability of GraphBT, an integrated software development tool for creating Behavior Tree.

1.2 Project Description

GraphBT is an integrated software development tools used to create Behavior Tree. One of the main goals of the project is to provide a more user friendly tool for developing Behavior Tree using graphical editor compared to the common text based tools.

II. Literature

2. 1. Eclipse

2.1.1 Eclipse as IDE

Eclipse IDE is one of the most popular IDE used by open source community. Eclipse IDE runs in any kind of platform (Linux, Windows, and Mac). It also supports many programming language, such as Java, C++, PHP, Perl, and other languages. It is possible to support many languages since Eclipse platform constructed by plug-in concept. User can develop basic functionality from Eclipse IDE so it capable to support custom programming language.

Eclipse platform consists of many plug-ins. The core of Eclipse provides basic functionality that can be developed by building a plug-in on it. Figure 1 shows the three layers of Eclipse.



Figure 1 The Three Layers of Eclipse

- Platform: The Eclipse platform defines the common programming-language-neutral infrastructure
- Java Development Tools (JDT): The Java development tools add a full-featured Java IDE to Eclipse
- Plug-In Development Environment (PDE): The PDE extends the JDT with support for developing plug-ins

The platform consists of several key components that are layered into a user interface (UI)-independent core and a UI Layer, as shown in Figure 2.

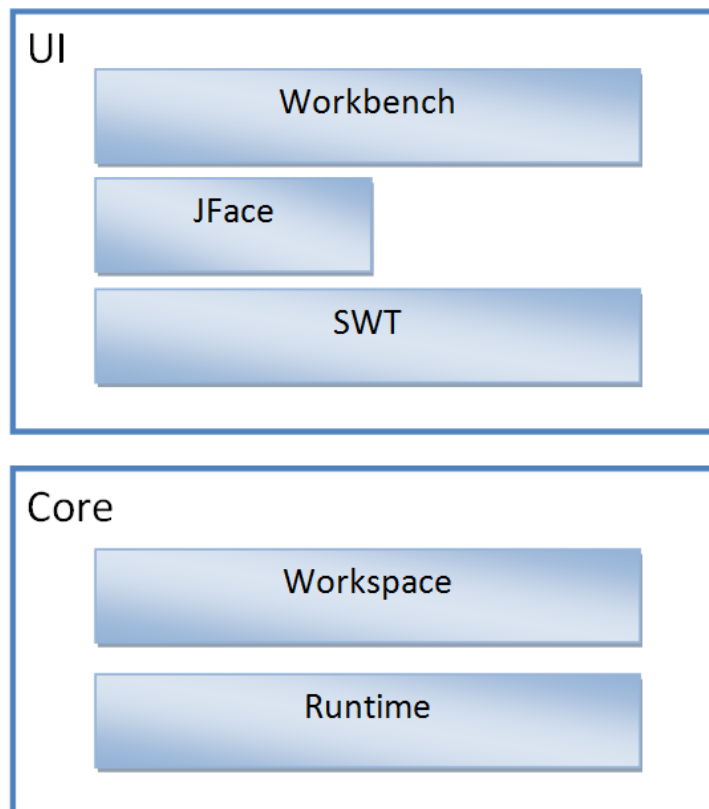


Figure 2 UI Independent Core & UI Layer

- Runtime: The run time component defines the plug-in infrastructure. It discovers the available plug-ins on start-up and manages the plug-in loading
- Workspace: A workspace manages one or more top-level projects. A project consists of file and folders that map onto the underlying file system.
- Standard Widget Toolkit (SWT): The SWT provides graphics and defines a standard set of widgets.
- JFace: A set of smaller UI frameworks built on top of SWT supporting common UI tasks.
- Workbench: The workbench defines the Eclipse UI paradigm. It centers on editors, views, and perspectives.

2.1.2 Eclipse as Framework

However, Eclipse is not only used as development tools. Eclipse may also functionate as Framework. In this case, a number of Eclipse project can be used to accelerate software development process. Several frameworks provided in Eclipse include JFace, Business Intelligence and Reporting Tools, Graphical Modeling Framework, Eclipse Communication Framework, and Eclipse Modeling Framework (EMF) which has been used in this project. These Eclipse projects can be found at:

<http://www.eclipse.org/home/categories/frameworks.php>.

2.1.3 Contributing in Eclipse

Eclipse facilitates developers to contribute in Eclipse. The contribution can be report bugs and enhancements, fix bugs and implement enhancements, join projects or start a new project, etc. Everyone can contribute in Eclipse in many ways:

<http://www.eclipse.org/contribute/>.

2. 2. Framework

2.2.1 Graphiti

Graphiti is a framework used to provide graphical representation of a domain model. One of the easiest domain model to work with Graphiti is Eclipse Modeling Framework(EMF) – based even though it should work well with any Java-based model.

Graphiti has several special features:

- Graphiti's API doesn't show platform dependent technologies such as GEF and Draw2D, so that users only deal with plain Java API and other specific objects from Graphiti.
- Graphical editor can be reached from first result by using provided default implementation. Increasingly, editor can be developed and improved further.
- User interaction for Graphiti can be modified freely to specific needs.
- Graphiti only supports Eclipse. However, Eclipse framework is open for other platform, so the specific framework need to be used can be integrated.

For further information about Graphiti, you can visit Eclipse site for Graphiti:

<http://www.eclipse.org/graphiti/>.

2.2.2 M2M ATL

Model-to-Model Transformation (M2M) ATL is model transformation framework providing the way to transform target models from the set of source models. M2M ATL enables developers to define the mechanism to match the source model elements and produce target model elements. This M2M ATL's model transformation process is implemented in TextBE, so that we can use it easily from TextBE.

For further information about M2M ATL, you can visit Eclipse site for M2M ATL:

<http://wiki.eclipse.org/ATL/Concepts>.

2.2.3 ZEST

Zest is Eclipse plug-in provides a set of visualization components built for Eclipse. Zest's main purpose is to make graph based programming. To make graph, Zest use SWT Component which have been wrapped using standard JFace viewers, so developers can use Zest as they use JFace Tables, Trees, and Lists.

For further information and tutorials about Zest, you can:

http://wiki.eclipse.org/index.php/GEF_Zest_Visualization.

2.2.4 EMF

Graphiti is used as graphical representation of a domain model, the domain model itself is provided by EMF. EMF stands for Eclipse Modeling Framework, an Eclipse based modeling framework and code generation facility for building tools and other applications based on a structured data model. EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Models can be specified using annotated Java, UML, XML documents, or modeling tools, then imported into EMF.

For further information about EMF, you can visit Eclipse site for EMF:

<http://www.eclipse.org/modeling/emf>.

2.3. Behavior Tree

Behavior Tree is one of Behavior Modeling Language which represents whole behavior activity of a system. Behavior Tree is discussed here because it is the main tree that can be developed using the GraphBT tool. Behavior Tree has a metamodel that describes its property and how it relates to each other.

2.3.1 BT Meta Model

BT meta model is described by Behavior Tree group. The released v.1 can be accessed in Appendix 1.

2.3.2 Notation

As a diagram, BT has a notation to formalize its form. The notation is published by Behavior Tree Group and can be accessed [here](#).

2.3.3 Consortium

BT Consortium is managed by Behavior Tree Group. Its official page can be accessed [here](#).

2.4. Features

2.4.1 Graphical Editor

2.4.1.1 Editor

Editor is a frame in Eclipse where user can edit their work on. In this project we develop two kinds of frame where user can develop the required Behavior Tree: the diagram editor and textual editor.

These editors can be added into the plug-in by extending `MultiPageEditorPart` from Graphiti and creating `MultiPageEditor` in the GraphBT project. The diagram editor and textual editor then created inside the `MultiPageEditor`. The real implementation for `MultiPageEditor` is contained in `behaviortree.graphBT/src/behaviortree.graphbt.editors/MultiPageEditor.java`, as shown in **Error! Reference source not found..**

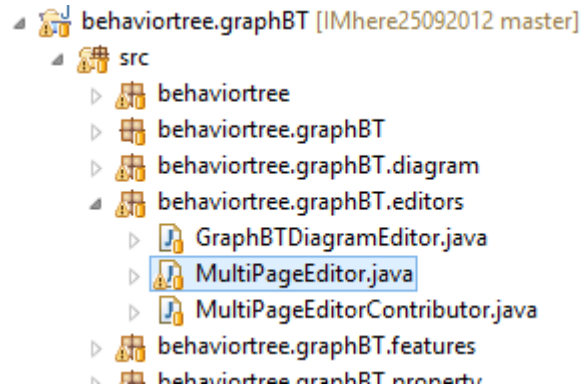


Figure 3 Package Explorer of GraphBT Project for MultiPageEditor

2.4.1.2 Editor Contributor

After the editor is created, the next step is to contribute to the editor. This can be provided by extending `MultiPageEditorActionBarContributor` to the Graphiti. The implementation of editor contributor in GraphBT is provided in `behaviortree/graphBT/src/behaviortree/graphBT/editors/MultiPageEditorContributor.java`.

2.4.1.3 Add BT Node

Adding new BT node to the diagram editor use the same method with adding new pictogram element to the clipboard. This feature is implemented by extending `AbstractAddShapeFeature` class in the implementation and implementing the `IAddFeature` interface. The real implementation can be found in `behaviortree/graphBT/src/behaviortree/graphBT/features/AddGeneralBtNodeFeature.java`, as shown in **Error! Reference source not found..**

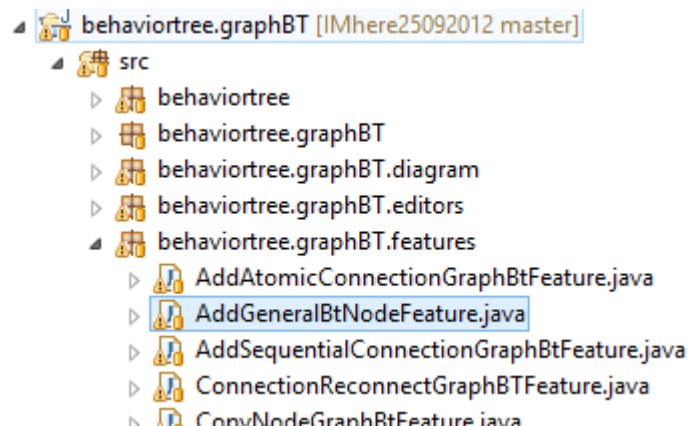


Figure 4 Package Explorer of GraphBT Project for Add BT Node

```
public class AddGeneralBtNodeFeature extends
AbstractAddShapeFeature implements
IAddFeature {
```

```

        public AddGeneralBtNodeFeature(IFeatureProvider fp) {
            super(fp);
        }

        private static final IColorConstant E_CLASS_TEXT_FOREGROUND
=
            IColorConstant.BLACK;

        private static final IColorConstant E_CLASS_FOREGROUND =
            new ColorConstant(0, 0, 0);

        public static final IColorConstant ORIGINAL_BEHAVIOR_COLOR =
            new ColorConstant("99FF66");

        public static final IColorConstant IMPLIED_BEHAVIOR_COLOR =
            new ColorConstant("FFFF66"); //yellow

        public static final IColorConstant MISSING_BEHAVIOR_COLOR =
            new ColorConstant("FF6666"); //red

        public static final IColorConstant UPDATED_BEHAVIOR_COLOR =
            new ColorConstant("66CCFF"); //blue

        public static final IColorConstant DELETED_BEHAVIOR_COLOR =
            new ColorConstant("FFFFFF"); //white


        public boolean canAdd(IAddContext context) {
            if (context.getNewObject() instanceof StandardNode) {
                if (context.getTargetContainer() instanceof Diagram)
                {
                    return true;
                }
            }
            return false;
        }

        public PictogramElement add(IAddContext context) {
            StandardNode node = (StandardNode) context.getNewObject();
            Diagram targetDiagram = (Diagram)
context.getTargetContainer();
            return createPENode(targetDiagram,
node, context.getX(), context.getY());
        }

        public PictogramElement createPENode(Diagram targetDiagram,
StandardNode node, int x, int y) {
            // CONTAINER SHAPE WITH RECTANGLE
            IPeCreateService peCreateService =
Graphiti.getPeCreateService();
            ContainerShape containerShape =
                peCreateService.createContainerShape(targetDiagram,
true);

            // default size for the shape
            int width = 170;
            int height = 80;

```

```

IGaService gaService = Graphiti.getGaService();

...

...

...

```

2.4.1.4 Create BT Node

Before the diagram of the corresponding BT Node can be added, we should first create the domain model. In this project we use EMF as modeling framework to create needed classes for domain model. Afterwards, we need to create a class in Graphiti as graphical editor which extends `AbstractCreateFeature` and implements `ICreateFeature`. The implemented class will initialize core features needed to define a BT Node such as Component, Behavior, Operator, Traceability Link, Traceability Status, and Operator. This class will also invoke a wizard implementation where user can input the BT Node features listed above. The implementation of wizard class will be explained later. The implementation of creating BT Node can be found in `behaviortree/graphBT/src/behaviortree/graphBT/features/CreateGeneralBtNodeFeature.java`, as Figure 5.

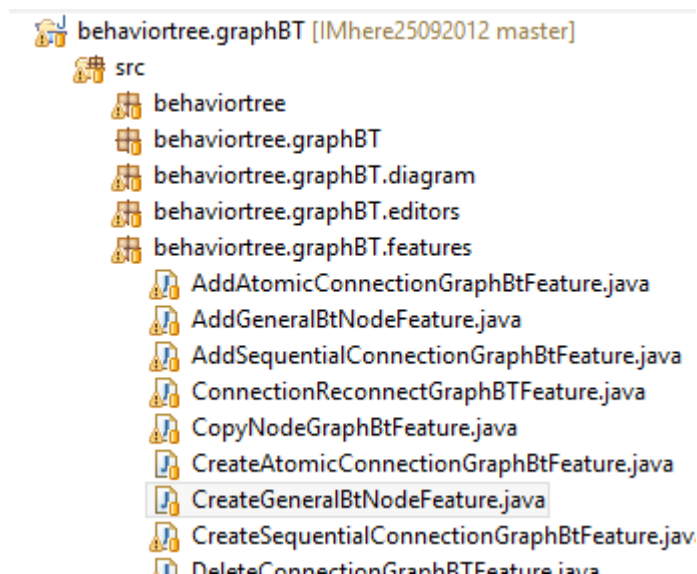


Figure 5 Package Explorer of GraphBT Project for Create Node

```

public class CreateGeneralBtNodeFeature extends
AbstractCreateFeature implements
ICreateFeature {
    private BEModel beModel;

    public CreateGeneralBtNodeFeature(IFeatureProvider fp) {
        // set name and description of the creation feature
        super(fp, "General BT Node", "Create Behavior Tree
Node");
    }
}

```

```

/**
 * Check whether the BT node model can be created
 */
public boolean canCreate(ICreateContext context) {
    return context.getTargetContainer() instanceof Diagram;
}

public void initiateBT(Node node) {
    beModel.setDbt(GraphBTUtil.getBEFactory().createBehaviorTree());
    beModel.getDbt().setRootNode(node);
}

public Object[] create(ICreateContext context) {
    Resource resource = getDiagram().eResource();
    HashMap<Integer,String> map = new HashMap<Integer,String>();

    WizardDialog wizardDialog = new
    WizardDialog(PlatformUI.getWorkbench().
        getActiveWorkbenchWindow().getShell(),
        new CreateStandardNodeGraphBTWizard(map,
        getDiagram()));

    if (wizardDialog.open() != Window.OK) {
        return null;
    }

    StandardNode node =
    BehaviorTreeFactory.eINSTANCE.createStandardNode();
    node.setLabel(System.currentTimeMillis()+"");
    resource.getContents().add(node);

    BEModel beModel =
    GraphBTUtil.getBEModel(getDiagram());

    if (map.get(StandardNode.TRACEABILITYSTATUS_VALUE)==null
    || (map.get(StandardNode.TRACEABILITYSTATUS_VALUE) !=null && map.get(StandardNode.COMPONENT_VALUE).equals("")) ) {
        node.setTraceabilityStatus(TraceabilityStatus.ORIGINAL.getLiteral());
    }
    else {
        node.setTraceabilityStatus(TraceabilityStatus.getByName(map.get(StandardNode.TRACEABILITYSTATUS_VALUE)).getLiteral());
    }
    if (map.get(StandardNode.OPERATOR_VALUE)==null
    || (map.get(StandardNode.OPERATOR_VALUE) !=null && map.get(StandardNode.COMPONENT_VALUE).equals("")) ) {
        node.setOperator(Operator.NO_OPERATOR.getLiteral());
    }
    else {
        node.setOperator(Operator.getByName(map.get(StandardNode.OPERATOR_VALUE)).getLiteral());
    }
}

```

```

        PERATOR_VALUE)).getLiteral());
    }

    Component c = null;
    if (map.get(StandardNode.COMPONENT_VALUE) == null
        || (map.get(StandardNode.COMPONENT_VALUE) != null && map.get(Standard
Node.COMPONENT_VALUE).equals(""))) {
        c =
BehaviortreeFactory.eINSTANCE.createComponent();
        c.setComponentName("DefaultComponent");
        c.setComponentRef("DefaultComponent");
    }

    ...
    ...
    ...

```

2.4.1.5 Add BT Connection

BT Node can have connection between them which usually referred as Edge. This Edge have two types, the sequential edge and atomic edge. Add BT Connection feature is responsible for generating the graphical representation of Edge. The implementation of Sequential Edge can be found in `behaviortree/graphBT/src/behaviortree/graphBT/features/AddSequentialGraphBtConnection.java` while the implementation of atomic Edge can be found in `behaviortree/graphBT/src/behaviortree/graphBT/features/AddAtomicGraphBtConnection.java`.

The code listing for `AddSequentialConnectionGraphBtFeature.java` is shown below

```

public class AddSequentialConnectionGraphBtFeature extends
AbstractAddFeature implements
    IAddFeature {

    public
AddSequentialConnectionGraphBtFeature(IFeatureProvider fp) {
        super(fp);
    }

    @Override
    public boolean canAdd(IAddContext context) {
        if (context instanceof IAddConnectionContext &&
            context.getNewObject() instanceof Link)
        {
            return true;
        }
        return false;
    }

    @Override
    public PictogramElement add(IAddContext context) {
        IAddConnectionContext addConContext =
(IAddConnectionContext) context;
        IPeCreateService peCreateService =
Graphiti.getPeCreateService();
        IGaService gaService = Graphiti.getGaService();

```

```

        Connection connection =
peCreateService.createFreeFormConnection(getDiagram());

        connection.setStart(addConContext.getSourceAnchor());
        connection.setEnd(addConContext.getTargetAnchor());

        Polyline polyline =
gaService.createPlainPolyline(connection);

        polyline.setForeground(manageColor(IColorConstant.BLACK));

        Link addedLink = (Link) context.getNewObject();
        link(connection, addedLink);

        ConnectionDecorator cd;
        cd =
peCreateService.createConnectionDecorator(connection, false,
1.0, true);
        createArrow(cd);

        if(addedLink.getSource().getEdge().getBranch().getLiteral(
).equals(Branch.ALTERNATIVE.getLiteral())) {
            ConnectionDecorator cd2;
            cd2 =
peCreateService.createConnectionDecorator(connection, false,
0.0, true);

        }
        GraphBTUtil.applyTreeLayout(getDiagram());
        GraphBTUtil.updateReversionNode((DiagramEditor)
getDiagramEditor());
        return connection;
    }

    private Polygon createArrow(GraphicsAlgorithmContainer
gaContainer) {

        Polygon polygon =
Graphiti.getGaCreateService().createPlainPolygon(gaContainer,
new int[] { -15, 10, 0, 0, -15, -10, -15, 10 });
        return polygon;
    }

    private Polygon
createAlternativeIdentifier(GraphicsAlgorithmContainer
gaContainer) {
        Polygon polygon =
Graphiti.getGaCreateService().createPlainPolygon(gaContainer,
new int[] { -5, 5, 5, 5, 5, -5, -5, -5, -5, 5 });
        polygon.setBackground(manageColor(new
ColorConstant(0, 255, 0)));
        polygon.setForeground(manageColor(new
ColorConstant(0, 0, 0)));
        polygon.setLineWidth(1);
        return polygon;
    }

```

```

        private Ellipse
        createIdentifier(GraphicsAlgorithmContainer gaContainer) {
            Ellipse ellipse =
            Graphiti.getGaCreateService().createEllipse(gaContainer);
            ellipse.setHeight(10);
            ellipse.setWidth(10);
            return ellipse;
        }
    }
}

```

The code listing for AddAtomicConnectionGraphBtFeature.java is shown below

```

public class AddAtomicConnectionGraphBtFeature extends
AbstractAddFeature implements
    IAddFeature {

    public AddAtomicConnectionGraphBtFeature(IFeatureProvider
fp) {
        super(fp);
    }

    @Override
    public boolean canAdd(IAddContext context) {
        if (context instanceof IAddConnectionContext &&
            context.getNewObject() instanceof Link)
        {
            return true;
        }
        return false;
    }

    @Override
    public PictogramElement add(IAddContext context) {
        IAddConnectionContext addConContext =
        (IAddConnectionContext) context;
        IPeCreateService peCreateService =
        Graphiti.getPeCreateService();
        IGaService gaService = Graphiti.getGaService();

        Connection connection =
        peCreateService.createFreeFormConnection(getDiagram());

        connection.setStart(addConContext.getSourceAnchor());
        connection.setEnd(addConContext.getTargetAnchor());

        Polyline polyline =
        gaService.createPlainPolyline(connection);

        gaService.setLocation(polyline,
        addConContext.getX(), addConContext.getY());

        polyline.setForeground(manageColor(IColorConstant.BLACK));

        Link addedLink = (Link) context.getNewObject();
        link(connection, addedLink);
        GraphBTUtil.applyTreeLayout(getDiagram());
        return connection;
    }
}

```


2.4.1.6 Create BT Connection

The domain model implementation of Edge can be found in `behaviortree/graphBT/src/behaviortree/graphBT/features/CreateSequentialGraphBtConnection.java` and `behaviortree/graphBT/src/behaviortree/graphBT/features/CreateAtomicGraphBtConnection.java`

```
public class CreateSequentialConnectionGraphBtFeature extends
AbstractCreateConnectionFeature
    implements ICreateConnectionFeature {

    public
    CreateSequentialConnectionGraphBtFeature(IFeatureProvider fp) {
        super(fp, "Sequential", "Creates a new sequential
edge between nodes");
    }

    @Override
    public boolean canStartConnection(ICreateConnectionContext
context) {

        PictogramElement pe =
context.getSourcePictogramElement();
        if (this.getBusinessObjectForPictogramElement(pe) !=
null)
        {
            if
            (this.getBusinessObjectForPictogramElement(pe) instanceof
StandardNode)
            {
                StandardNode st =
(StandardNode) this.getBusinessObjectForPictogramElement(pe);

                if (st.getEdge() == null ||
(st.getEdge() != null &&
st.getEdge().getComposition().getValue() !=
Composition.ATOMIC_VALUE)) {

                    return true;
                }
            }
        }

        return false;
    }

    /**
     * Criteria that a node can be created
     * - the target
     */
    @Override
    public boolean canCreate(ICreateConnectionContext context)
    {

        PictogramElement pe =
context.getSourcePictogramElement();
        PictogramElement peS =
```

```

context.getTargetPictogramElement();
        if (this.getBusinessObjectForPictogramElement(pe) !=
null)
        {
            if
(
this.getBusinessObjectForPictogramElement(pe) instanceof
StandardNode)
            {
                StandardNode ss =
(StandardNode) this.getBusinessObjectForPictogramElement(peS);

                return true;
            }
        }
        return false;
    }

@Override
    public Connection create(ICreateConnectionContext context)
    {
        Connection newConnection = null;
        // get EClasses which should be connected
        StandardNode source =
getStandardNode(context.getSourcePictogramElement());
        StandardNode target =
getStandardNode(context.getTargetPictogramElement());

        if (source != null && target != null && target !=
source) {
            // create new business object

            Link l = createLink(source, target);
            if (l==null)
            {
                return null;
            }

            // add connection for business object
            AddConnectionContext addContext =
new
AddConnectionContext(context.getSourceAnchor(),
context.getTargetAnchor());
            addContext.setNewObject(l);
            newConnection = (Connection)
getFeatureProvider().addIfPossible(addContext);

            PictogramElement pes =
context.getSourcePictogramElement();
            PictogramElement pet =
context.getTargetPictogramElement();
            ...
            ...
            ...

```

```

public class CreateAtomicConnectionGraphBtFeature
    extends AbstractCreateConnectionFeature
    implements ICreateConnectionFeature {

```

```

        public
        CreateAtomicConnectionGraphBtFeature(IFeatureProvider fp) {
            super(fp, "Atomic", "Creates a new atomic " +
                "composition edge between nodes");
        }

        @Override
        public boolean canStartConnection(ICreateConnectionContext
context) {
            if (getStandardNode(context.getSourceAnchor()) !=
null) {
                StandardNode md =
getStandardNode(context.getSourceAnchor());
                if(md.getEdge()==null)
                    return true;
            }
            return false;
        }

        @Override
        public boolean canCreate(ICreateConnectionContext context)
{
            StandardNode source =
getStandardNode(context.getSourceAnchor());
            StandardNode target =
getStandardNode(context.getTargetAnchor());

            if (source != null && target != null && source !=
target && target != source) {
                if(!target.isLeaf())
                    return true;
            }

            return false;
        }

        @Override
        public Connection create(ICreateConnectionContext context)
{
            Connection newConnection = null;

            // get EClasses which should be connected
            StandardNode source =
getStandardNode(context.getSourceAnchor());
            StandardNode target =
getStandardNode(context.getTargetAnchor());

            if (source != null && target != null) {
                // create new business object
                Link l = createLink(source, target);
                if(l==null)
                {
                    return null;
                }
                // add connection for business object
                AddConnectionContext addContext =
                    new
AddConnectionContext(context.getSourceAnchor(),
                        context.getTargetAnchor());
            }
        }
    }

```

```

        addContext.setNewObject(1);
        newConnection = (Connection) getFeatureProvider().
            addIfPossible(addContext);

        PictogramElement pes =
context.getSourcePictogramElement();
        PictogramElement pet =
context.getTargetPictogramElement();

pet.getGraphicsAlgorithm().setX(pes.getGraphicsAlgorithm().getX(
));
        pet.getGraphicsAlgorithm().setY(
            pes.getGraphicsAlgorithm().getY() +
            pes.getGraphicsAlgorithm().getHeight());
    }

    return newConnection;
}

private StandardNode getStandardNode(Anchor anchor) {
    if (anchor != null) {
        Object object =
getBusinessObjectForPictogramElement(anchor.getParent());
        if (object instanceof StandardNode) {
            return (StandardNode) object;
        }
    }
    return null;
}

private Link createLink(StandardNode source, StandardNode
target) {
    if (GraphBTUtil.isAncestor(target, source))
    {
        return null;
    }
    Edge edge = source.getEdge();
    if (edge == null)
    {
        edge = BehaviortreeFactory.eINSTANCE.createEdge();
        edge.setComposition(Composition.ATOMIC);
        source.setEdge(edge);
        edge.setContainer(source);
    }
    Link l = GraphBTUtil.getBEFactory().createLink();
    l.setSource(source);
    l.setTarget(target);
    edge.getChildNode().add(l);
    target.setLeaf(true);
    target.setParent(source);
    return l;
}
}

```

2.4.1.7 Update Features

Another important feature in graphical editor is handling every changed state of the graphics. This task is handled in update features which also provided by Graphiti. The

implementation of Update can be found in
behaviortree/graphBT/src/behaviortree/graphBT/features/UpdateGraphBtFeature
.java

```
public class UpdateGraphBtFeature extends AbstractUpdateFeature
{
    public UpdateGraphBtFeature(IFeatureProvider fp) {
        super(fp);
    }

    public boolean canUpdate(IUpdateContext context) {
        // return true, if linked business object is a
        StandardNode
        PictogramElement pictogramElement =
        context.getPictogramElement();
        Object bo =
        getBusinessObjectForPictogramElement(pictogramElement);

        return ((bo instanceof Component) || (bo instanceof
        Behavior) || (bo instanceof OperatorClass) ||
        (bo instanceof Requirement) || (bo instanceof
        TraceabilityStatusClass) || (bo instanceof AlternativeClass));
    }

    public IReason updateNeeded(IUpdateContext context) {
        // retrieve name from pictogram model
        String pictogramName = null;
        PictogramElement pictogramElement =
        context.getPictogramElement();

        Object bo =
        getBusinessObjectForPictogramElement(pictogramElement);
        Object oSN =
        getBusinessObjectForPictogramElement(((Shape) context.getPictogra
        mElement()).getContainer());

        if(! (oSN instanceof StandardNode)) {
            return Reason.createFalseReason();
        }

        StandardNode node = (StandardNode) oSN;
        String businessName = null;

        if (bo instanceof Component) {
            Component c =
            GraphBTUtil.getComponentByRef(GraphBTUtil.getBEModel(getDiagram(
            )), node.getComponentRef());
            if(c != null) {
                businessName = c.getComponentName();
            }
        }
        else if (bo instanceof Behavior) {
            if(node.getBehaviorRef() != null) {
                Component c =
                GraphBTUtil.getComponentByRef(GraphBTUtil.getBEModel(getDiagram(
                )), node.getComponentRef());
                businessName =
                GraphBTUtil.getBehaviorFromComponentByRef(c,
                node.getBehaviorRef()).toString();
            }
        }
    }
}
```

```

        }
        else {
            businessName = null;
        }
    }
    else if (bo instanceof Requirement) {
        Requirement c =
GraphBTUtil.getRequirement(GraphBTUtil.getBEModel(getDiagram()),
node.getTraceabilityLink());
        if(c != null) {
            businessName = c.getKey();
        }
        else {
            businessName = "";
        }
    }
    else if (bo instanceof TraceabilityStatusClass) {
        businessName = node.getTraceabilityStatus();
    }
    else if (bo instanceof OperatorClass) {
        businessName = node.getOperator();
    }
    else if (bo instanceof AlternativeClass) {
        if(node.getEdge() == null) {
            return Reason.createFalseReason();
        }
        else if (node.getEdge().getBranch() == null) {
            return Reason.createFalseReason();
        }
        else if (node.getEdge().getBranch() != null) {
            if (node.getEdge().getBranch().getName().equals(Branch.ALTE
RNATIVE.getName())) {
                businessName = "A";
            }
            else {
                businessName = "";
            }
        }
    }
}

if(((Shape)pictogramElement).getGraphicsAlgorithm()
 instanceof Text) {
    pictogramName =
((Text)((Shape)pictogramElement).getGraphicsAlgorithm()).getValu
e();
}

boolean updateNameNeeded = ((pictogramName == null
&& businessName != null) ||
    (pictogramName != null &&
!pictogramName.equals(businessName)));
    if (updateNameNeeded) {
        return Reason.createTrueReason("Name is out of
date");
    } else {
        return Reason.createFalseReason();
    }
}
// retrieve name from business model

```

```

    }

    public boolean update(IUpdateContext context) {
        // retrieve name from business model
        String businessName = null;
        PictogramElement pictogramElement =
context.getPictogramElement();

        Object bo =
getBusinessObjectForPictogramElement(pictogramElement);
        Object oSN =
getBusinessObjectForPictogramElement(((Shape) context.getPictogra
mElement()).getContainer());
        StandardNode node = (StandardNode) oSN;

        if (bo instanceof Component) {
            Component comp =
GraphBTUtil.getComponentByRef(GraphBTUtil.getBEModel(getDiagram(
)), node.getComponentRef());
            businessName = comp.getComponentName();
            Shape shape = (Shape) pictogramElement;

pictogramElement.getLink().getBusinessObjects().clear();

pictogramElement.getLink().getBusinessObjects().add(comp);
            if (shape.getGraphicsAlgorithm() instanceof Text) {
                Text text = (Text) shape.getGraphicsAlgorithm();
                text.setValue(businessName);
                return true;
            }
        }
        if (bo instanceof Behavior) {
            Behavior beh =
GraphBTUtil.getBehaviorFromComponentByRef(GraphBTUtil.getCompone
ntByRef(GraphBTUtil.getBEModel(getDiagram()),
node.getComponentRef()), node.getBehaviorRef());
            businessName = beh.toString();
            Shape shape = (Shape) pictogramElement;

pictogramElement.getLink().getBusinessObjects().clear();

pictogramElement.getLink().getBusinessObjects().add(beh);
            if (shape.getGraphicsAlgorithm() instanceof Text) {
                Text text = (Text) shape.getGraphicsAlgorithm();
                text.setValue(businessName);
                return true;
            }
        }
        if (bo instanceof Requirement) {
            Requirement r =
GraphBTUtil.getRequirement(GraphBTUtil.getBEModel(getDiagram()),
node.getTraceabilityLink());
            Shape shape = (Shape) pictogramElement;

pictogramElement.getLink().getBusinessObjects().clear();

pictogramElement.getLink().getBusinessObjects().add(r);

            if (r != null)

```

```

        businessName = r.getKey();
    else
        businessName = "";

    if (shape.getGraphicsAlgorithm() instanceof Text) {
        Text text = (Text) shape.getGraphicsAlgorithm();
        text.setValue(businessName);
        return true;
    }
}
if (bo instanceof TraceabilityStatusClass) {
    businessName = node.getTraceabilityStatus();
    Shape shape = (Shape) pictogramElement;

    if (shape.getGraphicsAlgorithm() instanceof Text) {
        Text text = (Text) shape.getGraphicsAlgorithm();
        text.setValue(businessName);
        link(shape,
GraphBTUtil.getTraceabilityStatus(getDiagram(), businessName));
        return true;
    }
}
if (bo instanceof OperatorClass) {
    businessName = node.getOperator();
    Shape shape = (Shape) pictogramElement;

    if (shape.getGraphicsAlgorithm() instanceof Text) {
        Text text = (Text) shape.getGraphicsAlgorithm();
        text.setValue(businessName);
        return true;
    }
}
if (bo instanceof AlternativeClass) {
    if (node.getEdge().getBranch() != null &&
node.getEdge().getBranch().equals(Branch.ALTERNATIVE)) {
        businessName = "A";
    }
    else {
        businessName = "";
    }

    Shape shape = (Shape) pictogramElement;

    if (shape.getGraphicsAlgorithm() instanceof Text) {
        Text text = (Text) shape.getGraphicsAlgorithm();
        text.setValue(businessName);
        return true;
    }
}

return false;
}
}

```

2.4.1.8 Property section

User can modify the components of BT Node like Component, Behavior, Traceability Link, Traceability Status, Operator, and Branch Type (if the node has branch connection) using property view. This feature can be provided by manipulating

property section of the Eclipse. In order to manipulate property section, we first need to create a class which extends `AbstractPropertySectionFilter` which will check whether the pictogram element will be connected to the property section or not. The class will override method `accept(PictogramElement pe)`. The implementation of this feature can be found in `behaviortree/graphBT/src/behaviortree/graphBT/features/BehaviorTreeFilter.java`. The code listing is shown below:

```
public class BehaviorTreeFilter extends
AbstractPropertySectionFilter {

    @Override
    protected boolean accept(PictogramElement pe) {
        EObject eObject =
            Graphiti.getLinkService()
                .getBusinessObjectForLinkedPictogramElement(pe);

        EObject eObj = null;
        if(pe instanceof Shape){
            eObj = Graphiti.getLinkService()
                .getBusinessObjectForLinkedPictogramElement(((Shape)pe).getContainer());
        }
        else {
            return false;
        }

        boolean isTrue = false;
        if (eObject instanceof StandardNode) {
            isTrue = true;
        }
        if (eObject instanceof Component) {
            isTrue = true;
        }
        if (eObject instanceof Behavior) {
            isTrue = true;
        }
        if (eObj instanceof StandardNode) {
            isTrue = true;
        }
        if (eObj instanceof Component) {
            isTrue = true;
        }
        if (eObj instanceof Behavior) {
            isTrue = true;
        }
        return isTrue;
    }
}
```

After the filter class is created, we then implement a class that extends the `GFPPropertySection` to implement the property section so that we can modify the needed components from the BT Node. In the implementation, we first need to override `createControls(Composite parent, TabbedPropertySheetPage`

tabbedPropertySheetPage) method to create controls for the property section. Since property section will be used to modify a BT Node, in this method we define create CCombo's for the modifiable components in the Node such as Components, Behavior, Traceability Link, Traceability Status, Operator, and Branch Type. The code listing of the method is shown below

```
public class BehaviorTreePropertySection extends
GFPropertySection
    implements ITabbedPropertyConstants {
    private CCombo componentCombo;
    private CCombo behaviorCombo;
    private CCombo requirementCombo;
    private CCombo operatorCombo;
    private CCombo statusCombo;
    private CCombo branchCombo;
    private CLabel branchLabel;

    @Override
    public void createControls(Composite parent,
        TabbedPropertySheetPage tabbedPropertySheetPage) {
        super.createControls(parent,
tabbedPropertySheetPage);
        TabbedPropertySheetWidgetFactory factory =
getWidgetFactory();
        Composite composite =
factory.createFlatFormComposite(parent);

        FormData componentData;
        FormData behaviorData;
        FormData requirementData;
        FormData operatorData;
        FormData statusData;
        FormData branchData;

        IWorkbenchPage
page=PlatformUI.getWorkbench().getActiveWorkbenchWindow().getAct
ivePage();
        final DiagramEditor ds;
        if(page.getActiveEditor() instanceof DiagramEditor) {
            ds = (DiagramEditor)page.getActiveEditor();
        }
        else {
            ds =
((MultiPageEditor)page.getActiveEditor()).getDiagramEditor();
        }

        final Diagram d =
ds.getDiagramTypeProvider().getDiagram();

        BEModel model = GraphBTUtil.getBEModel(d);

        componentCombo = factory.createCCombo(composite);
        behaviorCombo = factory.createCCombo(composite);
        requirementCombo = factory.createCCombo(composite);
        operatorCombo = factory.createCCombo(composite);
        statusCombo = factory.createCCombo(composite);
        branchCombo = factory.createCCombo(composite);
        for(Branch branch : Branch.VALUES) {
            branchCombo.add(branch.getName());
        }
    }
}
```

```

    }
...
...
...

```

After defining the CCombos, we also need to add listeners to each CCombo so that we can modify the components from the BT Node. An example of listener implementation for CCombo is shown below

```

...
...
...
componentCombo.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        PictogramElement pe =
        getSelectedPictogramElement();
        Object ob = Graphiti.getLinkService()

        .getBusinessObjectForLinkedPictogramElement(pe);
        if(!(ob instanceof StandardNode))
            return;
        final StandardNode node = (StandardNode) ob;

        CCombo combo = (CCombo)e.widget;
        final String selected =
        combo.getItem(combo.getSelectionIndex());

        final Component c =
        GraphBTUtil.getComponent(GraphBTUtil.getBEModel(d), selected);

        Command cmd = new
        RecordingCommand(ds.getEditingDomain(), "Nope") {
            protected void doExecute() {

                node.setComponentRef(c.getComponentRef());
                String beh =
                c.getBehaviors().size()>0?c.getBehaviors().get(0).getBehaviorRef
                ():"";

                node.setBehaviorRef(beh);
                behaviorCombo.setText(beh);
                try{
                    refresh();
                }
                catch(RuntimeException re)
                {

                    WizardDialog wizardDialog =
                    new WizardDialog(PlatformUI.getWorkbench().

                    getActiveWorkbenchWindow().getShell(),

                    new
                    ManageComponentsGraphBTWizard(d));

                    if(wizardDialog.open()

                    != Window.OK)

                        {

                            return;
                        }
                }
            }
        }
    }
}

```

```

        };
        TransactionalEditingDomain f =
ds.getEditingDomain();
        f.getCommandStack().execute(cmd);
        ContainerShape cs = (ContainerShape)pe;
        Iterator<Shape> s =
cs.getChildren().iterator();
        while(s.hasNext()) {
            Shape n = s.next();

            Object bo =
Graphiti.getLinkService()

.getBusinessObjectForLinkedPictogramElement((PictogramElement)n)
;
            if(bo instanceof Component||bo
instanceof Behavior)
                updatePictogramElement(n);
        }

        behaviorCombo.removeAll();

        if(c!=null)
        for(Behavior behavior: c.getBehaviors()) {
            behaviorCombo.add(behavior.toString());
        }
    }
});
...
...
...

```

2.4.1.9 Wizards

We use wizard as the medium to receive the input from user to the components in the behavior tree. For example, if user wants to create a BT Node on the clipboard, before the node appeared a wizard will appear and user then can specify the node by selecting the components the want to make from the wizard. In order to create a wizard, we first need to create a class that will extend the Wizard class. In this class we then create objects of wizard pages in `addPages()` method where the wizard pages is implemented in another file. The wizard page class itself is where we define the controls of the wizard page. In order to implement a wizard page, we need to create a class that extends the `WizardPage` class and override the `createControl(Composite parent)` method. In this method we then create the `Composite` class and then set the layouting of the page and then add the widgets needed, such as buttons, combobox, selection list, and labels. In this method we also add every needed listeners to the widgets.

In this project we created 11 different wizards, each with different purposes. The implementations of the wizards can be found in `behaviortree/graphBT/src/behaviortree/graphBT/wizards`, as shown in Figure 6.

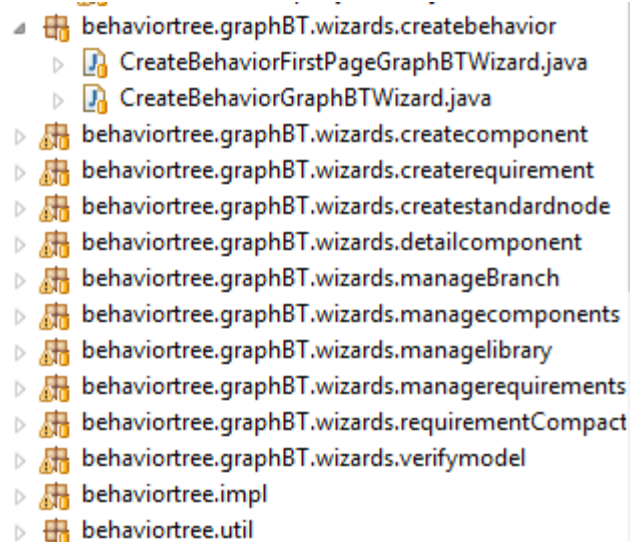


Figure 6 Package Explorer of GraphBT Project for Wizard

As an example of wizard implementation, the code listing of CreateStandardNodeGraphBTWizard class is shown in below

```
public class CreateStandardNodeGraphBTWizard extends Wizard {

    protected CreateStandardNodeFirstPageGraphBTWizard one;
    protected HashMap<Integer, String> map;
    protected Diagram d;

    public CreateStandardNodeGraphBTWizard(HashMap<Integer,
String> map, Diagram d) {
        super();
        setNeedsProgressMonitor(true);
        this.map = map;
        this.d = d;
    }

    @Override
    public void addPages() {
        one = new
CreateStandardNodeFirstPageGraphBTWizard(map, d);
        addPage(one);
    }

    @Override
    public boolean performFinish() {
        return true;
    }
}
```

The implementation of wizard page of creating BT Node wizard is shown in the code listing below

```
public CreateStandardNodeFirstPageGraphBTWizard(HashMap<Integer,
String> map, Diagram d) {
    super("Create Standard Node Wizard");
}
```

```

        setTitle("Create Standard Node Wizard");
        setDescription("Fill in the Behavior Tree node
elements below.");
        this.map = map;
        this.d=d;
    }

    @Override
    public void createControl(Composite parent) {

        container = new Composite(parent, SWT.NULL);
        GridLayout layout = new GridLayout();
        container.setLayout(layout);
        layout.numColumns = 4;
        GridData gridData;

        Button componentButton = new Button(container,
SWT.NULL);
        componentButton.setText("Add New Component");

        final Button behaviorButton = new Button(container,
SWT.NULL);
        behaviorButton.setText("Add New Behavior");
        behaviorButton.setEnabled(false);

        Button manageComponentsButton = new Button(container,
SWT.NULL);
        manageComponentsButton.setText("Manage Components");

        Button requirementButton = new Button(container,
SWT.NULL);
        requirementButton.setText("Add Requirement");

        Combo operatorCombo;
        Combo traceabilityStatusCombo;

        Label operatorLabel = new Label(container,
SWT.NULL);
        operatorLabel.setText("Operator Name:");

        operatorCombo = new Combo(container, SWT.BORDER |
SWT.READ_ONLY);
        gridData = new GridData(GridData.FILL_HORIZONTAL);
        gridData.horizontalSpan = 3;
        operatorCombo.setLayoutData(gridData);

        for (Operator op : Operator.VALUES) {
            operatorCombo.add(op.getName());
        }

        Label traceabilityLinkLabel = new Label(container,
SWT.NULL);
        traceabilityLinkLabel.setText("Traceability Link
Name:");

        traceabilityLinkCombo = new Combo(container,

```

```

SWT.BORDER | SWT.READ_ONLY);
    gridData = new GridData(GridData.FILL_HORIZONTAL);
    gridData.horizontalSpan = 3;
    traceabilityLinkCombo.setLayoutData(gridData);
...
...
...

```

2.4.1.10 Copy/Paste

The implementation copy and paste feature can be done by extending the class `AbstractCopyFeature` which we need to override the `canCopy(ICopyContext context)` and `copy(ICopyContext context)` method and class `AbstractPasteFeature` which we need to override the `canPaste(IPasteContext context)` and `paste(IPasteContext context)`. The real implementation can be found in `behaviortree/graphBT/src/behaviortree/graphBT/features/CopyNodeGraphBtFeature.java` and `behaviortree/graphBT/src/behaviortree/graphBT/features/PasteNodeGraphBtFeature.java`.

The code listing of `CopyNodeGraphBtFeature.java` and `PasteNodeGraphBtFeature.java` is shown below

```

public class CopyNodeGraphBtFeature extends AbstractCopyFeature
{
    public CopyNodeGraphBtFeature(IFeatureProvider fp) {
        super(fp);
    }

    @Override
    public boolean canCopy(ICopyContext context) {
        final PictogramElement[] pes =
context.getPictogramElements();
        if (pes == null || pes.length == 0) {
            return false;
        }

        for (PictogramElement pe : pes) {
            final Object bo =
getBusinessObjectForPictogramElement(pe);
            if (!(bo instanceof StandardNode)) {
                return false;
            }
        }
        return true;
    }

    @Override
    public void copy(ICopyContext context) {
        PictogramElement[] pes =
context.getPictogramElements();
        Object[] bos = new Object[pes.length];
        for (int i = 0; i < pes.length ; i++) {
            PictogramElement pe = pes[i];
            bos[i] = getBusinessObjectForPictogramElement(pe);

```

```

    }

    putToClipboard(bos);
}

```

```

public class PasteNodeGraphBtFeature extends
AbstractPasteFeature {

    public PasteNodeGraphBtFeature(IFeatureProvider fp) {
        super(fp);
    }

    public void paste(IPasteContext context) {
        PictogramElement[] pes =
context.getPictogramElements();
        Diagram diagram = (Diagram) pes[0];

        Object[] objects = getFromClipboard();
        for (Object object : objects) {
            AddContext ac = new AddContext();
            ac.setLocation(context.getX(),
context.getY());
            ac.setTargetContainer(diagram);
            if(object instanceof StandardNode)
            {
                object = copyNode((StandardNode) object);

                try {

                    GraphBTUtil.saveToModelFile((EObject) object,
getDiagram());
                } catch (CoreException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
                addGraphicalRepresentation(ac, object);
            }
        }

        public boolean canPaste(IPasteContext context) {
            PictogramElement[] pes =
context.getPictogramElements();
            if (pes.length != 1 || !(pes[0] instanceof Diagram))
            {
                return false;
            }

            Object[] fromClipboard = getFromClipboard();
            if (fromClipboard == null || fromClipboard.length ==
0) {
                return false;
            }
            for (Object object : fromClipboard) {
                if (!(object instanceof StandardNode)) {
                    return false;
                }
            }
        }
    }
}

```



```

        }
    }
    return true;
}
private StandardNode copyNode(StandardNode nd)
{
    StandardNode cnd =
GraphBTUtil.getBEFactory().createStandardNode();
    cnd.setBehaviorRef(nd.getBehaviorRef());
    cnd.setComponentRef(nd.getComponentRef());
    cnd.setLabel(""+System.currentTimeMillis());
    cnd.setTraceabilityLink(nd.getTraceabilityLink());
    cnd.setOperator(nd.getOperator());

    cnd.setTraceabilityStatus(nd.getTraceabilityStatus());
    return cnd;
}
}

```

2.4.2 Code Generator

Code Generator is a feature to produce java code from the given BT Diagram. The code generator receive an XML representation of .BT file (See Miscellaneous 1.1). The XML file resulted then processed by the plugin.

2.4.3 BT2ABS Design

The tool's name is BT2ABS because its function is to translate BT to ABS file. ABS model is generated by knowing the semantic and the structure of ABS. Figure 7 illustrates the structure of ABS.

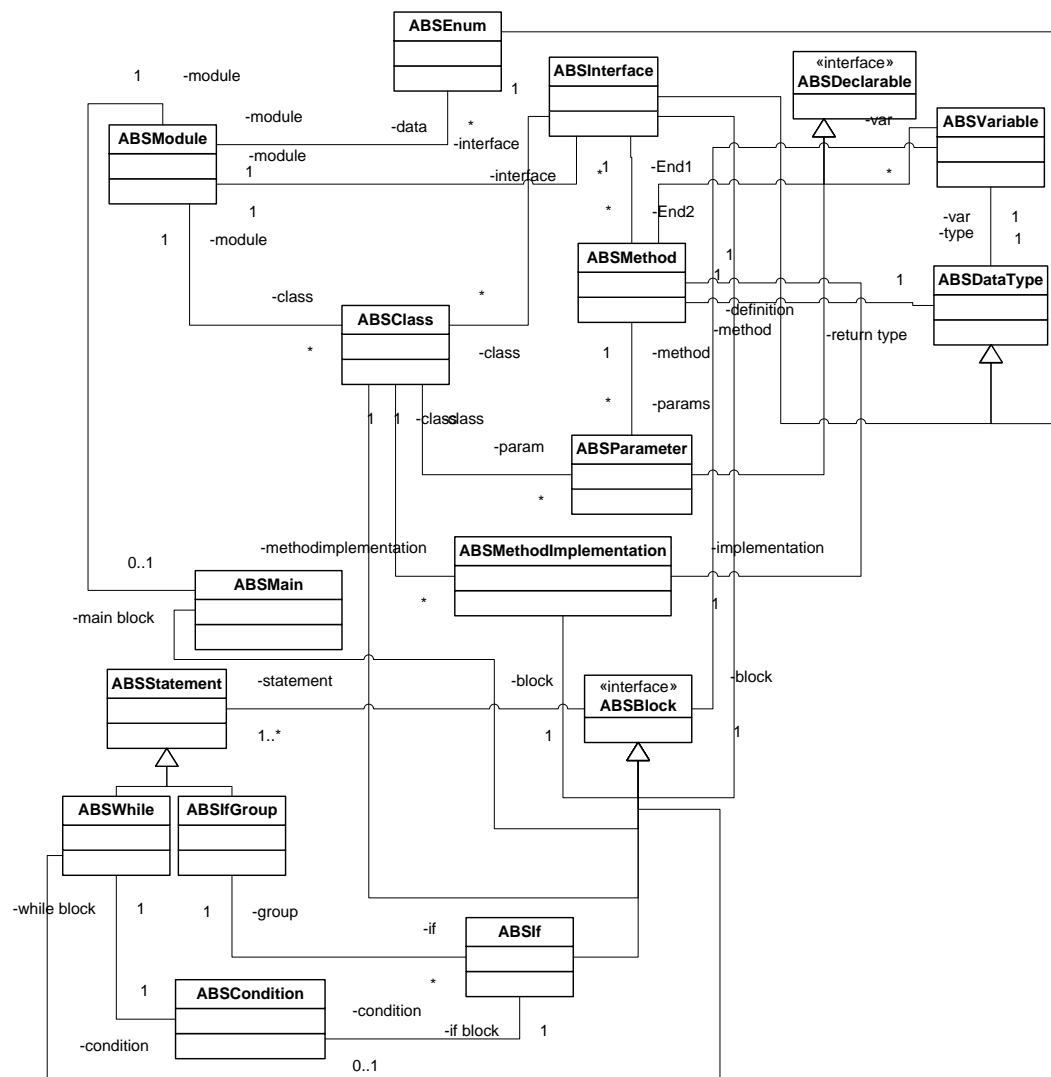


Figure 7 ABS Structure

However, the structure is not fixed, and still evolves as long as needed.

Figure 8 illustrates the structure of BT that need for translation:

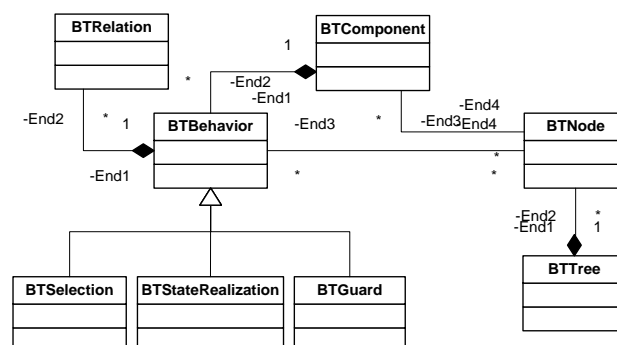


Figure 8 BT Structure

Figure 9 illustrates the ABS structure of the translated BT file:

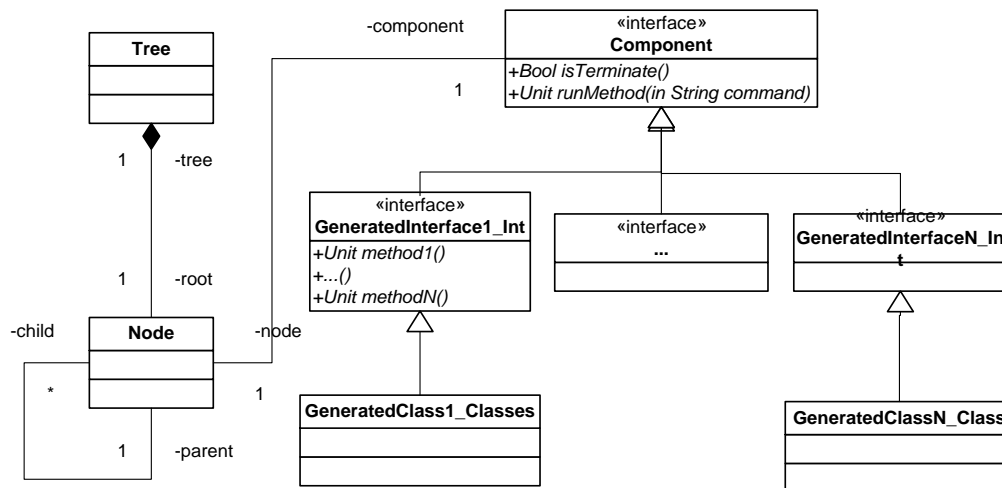


Figure 9 Generated ABS Structure

BT2ABS requires XML file of BT specification as an input. From section 3.3.1, TextBE automatically produces XML file from a BT file. The XML can easily be processed because it has formal semantic. Java also provides XML processing library.

Java XML parser library in package `javax.xml.parsers` is used. The used classes are: `DefaultHandler`, `SAXParserFactory` and `SAXParser`. Those classes will be used in parsing the XML file to BT object. Class `DefaultHandler` has standard properties and method declaration to process XML file.

The methods that should be implemented are:

1. Start Element

Declaration:

```

public void startElement (String uri, String localName, String qName,
Attributes attrs) throws SAXException
{
    ...
}
  
```

This method is needed to handle open tag of XML. Its parameters are:

- `uri`: the location of xml file
- `localName`: local name, namespace
- `qName`: name of XML tag
- `attrs`: list of XML attributes

In this method, object of each component, behavior, relation, BT node, and tree structure are created based on the corresponding tag.

2. End Element

Declaration:

```

public void endElement(String uri, String localName, String qName)
{
    ...
}
  
```

```
}
```

This method needed to handle close tag of XML. Its parameters are:

- `uri`: location of xml file
- `localName`: local name, namespace
- `qName`: name of XML tag

In this method, the created component to component list, behavior to behavior list, and relation is added to relation list.

3. End Document

Declaration:

```
public void endDocument()  
{  
    ...  
}
```

This method is needed to handle the post processing of the XML file. The post processing includes generating ABSModule, collecting ABSInterface and ABSClass from each BTComponent and also generating ABSMethods from each BTBehavior.

Translation Scheme

Translation scheme has two parts, first, declares the ABS structure, and second, declares execution steps.

ABS structure includes module declaration, interfaces, classes, and methods. The corresponding ABSInterface and ABSClass are created from each BTComponent. ABSInterface name specified by name of BTComponent appended with “_Int”, the name of class appended with “_Class”. The example is shown in Table 3.

Table 1 Example of interface handling

BTComponent		ABSInterface	ABSClass
Reference	Name	Name	Name
CPROD	Producer	Producer_Int	Producer_Class
CCONS	Consumer	Consumer_Int	Consumer_Class
CBUFF	Buffer	Buffer_Int	Buffer_Class
CHOST	Host	Host_Int	Host_Class

All possible variables from behavior declaration are needed to be extracted. State which its name is “LOCKED” could be a boolean variable with its value “True”, and “CTR:=0” is an Integer variable which assigned to value “0”. There are some rules to define variable name in ABS, e.g. first letter cannot be a capital letter. All variable must be in lowercase and appended with “_var”, to distinguish it as a generated variable instead of hard-coded one.

BT node consists of a reference BTBehavior in a BTComponent. The node is executed by executing the behavior in the referenced component. Each BTBehavior can be considered as ABSMethod. Behavior reference is used as the name of its method. BTBehavior reference is a number; which is

not allowed as the name of method in ABS, so it needs to be prepended with a string (e.g. “method9”, where “9” is behavior reference)¹. BTBehavior name is used as the statement in that method. ABSInterface contains each methods declaration, and ABSClass contains its method implementation².

BTBehavior type will specify how to deal with the corresponding code based on the behavior name. With the same behavior name, it has different meaning if it is represented in different type. For example, in *State realization*, LOCKED means that a variable LOCKED is assigned to *True*, but in *Selection*, LOCKED means it check whether LOCKED is *True* or not, in *Guard*, it will loop until LOCKED is *True*.

Translation in detail

Some heuristic (H) is defined to maintain each element in BT with corresponded ABS code which mentioned in section 3.3.1.

H1: For each component that contains only Enumeration state³, it will be declared as “Data” and all its state behavior as its possible value.

Example:

```
#C CWATER WATER
#S 1 COLD
#S 2 WARM
#S 3 HOT
```

Resulting:

```
data WATER_DATA = COLD | WARM | HOT | WATER_DEFAULT;
```

Explanation:

In BT, cold, warm and hot are some kind of state enumeration; each defines the possible temperature of water. The ABS code result appended “_DATA” to WATER to distinguish generated data type from other type, and also as convention of component naming. Component that declared as ‘Data’ should have a default value, for this purpose “WATER_DEFAULT” is used.

H2: Each component that contains behavior and does not meet the condition mentioned in H1 can be modeled as an ABS Interface.

Example:

```
#C CWATER WATER
#S 1 COLD
#S 2 WARM
#S 3 degrees:=90
#G 4 NOT(BOILED)
```

Result:

```
interface WATER_Int {
...
}
```

¹ “methodN” is used for conventional naming purpose, as it corresponded to its behavior number.

² It is obvious that all method declared in one interface should be implemented too in the class.

³ We define enumeration state as a state realization which have only single name, no “not”, assignment or symbol

```
class WATER_Class{
...
}
```

Explanation:

Behavior 4, have type guard, so it is not a state realization, behavior 3 is, but it is an assignment.

H1 is not fit, so it is defined as an interface.

H3: Each behavior definition in a BT component becomes a method. The methods are defined in the interface, and implemented in the class. Method name must be unique, so the reference number is used as the method name, and prepended with “method”. Method from behavior definition has no return value (Unit).

By implementing H3 in example from H2, the result is:

```
interface WATER_Int{
    Unit method1();
    Unit method2();
    Unit method3();
    Unit method4();
}
class WATER_Class{
    Unit method1()
    {
        ...
    }
    Unit method2()
    {
        ...
    }
    Unit method3()
    {
        ...
    }
    Unit method4()
    {
        ...
    }
}
```

H4: Collect all possible variables from behavior name, and decide what type they are.

By implementing H4 in example in H2, the result is:

Table 2 Type, variables and its default value

Ref	In BT	In ABS	Type	Initialization
1	COLD	cold_var	Bool	False
2	WARM	warm_var	Bool	False
3	degrees	degrees_var	Int	0
4	BOILED	boiled_var	Bool	False

Then, each variable is declared as attributes in corresponding class.

H5: Each type of behavior decides the implementation of each method. Since the behavior is represented by a method, it can be implemented as the body. First thing to do before

implementing the method is converting the behavior name into ABS compatible syntax (e.g change the variable, with variable name result of H4). The types are:

3. 3. State realization

If behavior type is state realization, then the code result is obvious. There is an adjustment if the behavior name is not a native statement, which is must parsed into corresponding code.

3. 3. Guard

A guard means if the component satisfies the condition, the node below will be executed. It also means that if the condition is not satisfied, it must wait until the condition is true. The wait should allow another process to run, so keyword `suspend` is used instead of `skip`.

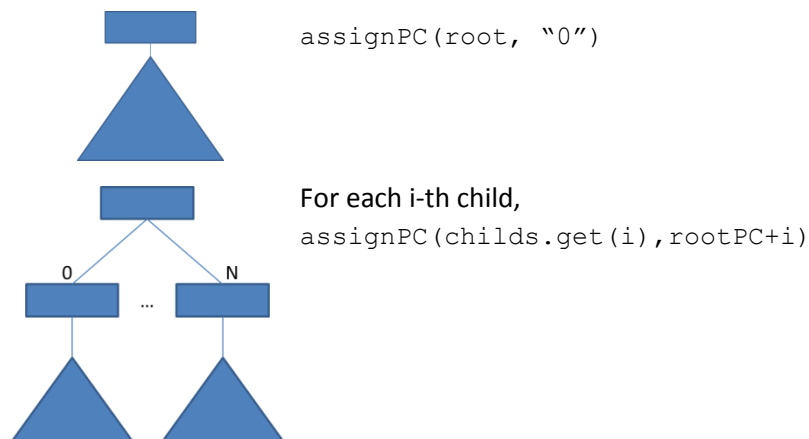
3. 3. Selection

A selection means if a component not satisfies the condition, then it process will be terminated. It implies that an “`if`” block is set where its condition is the negated condition in behavior name. If it satisfies the condition, it sets the process to termination.

By implementing H4 and H5 to experiment H2, the result is:

```
class WATER_Class{
Int degrees_var = 0;
Bool cold_var=False;
Bool warm_var=False;
Bool boiled_var=False;
Unit method1()
{
    cold_var = True;
}
Unit method2()
{
    warm_var = True;
}
Unit method3()
{
    degrees_var = 90;
}
Unit method4()
{
    while(not(not(boiled_var)))
    {
        suspend;
    }
}
}
```

Because BT has tree like structure, each node in BT is need to be specified into node of execution in ABS. The order of each node is specified by a program counter (PC) and the process is called as node labeling, as shown in Figure 10.



Labeling example:

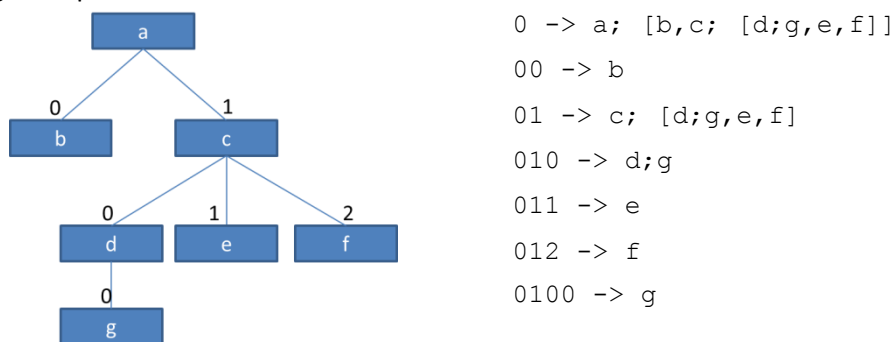


Figure 10 Prefix Tree

H6: To handle the concurrency, a tree is used to execute its child nodes asynchronously. Each node contains the node type, reference to a class (ABS representation of BTComponent), label, and string of method name that should be executed.

```
interface N {
    Bool runNode();
    String getPC();
    List<N> getChilds();
    Unit addChild(N node);
}

class Node(Component c, String cmd, String nt, String bt, String pc)
implements N {
    Component comp=c; String command=cmd; String nodetype=nt;
    String btype = bt; String pC = pc; List<N> childs= Nil;
    List<N> getChilds () { ... }
    Unit addChild(N node) { ... }
    Bool runNode() {
        Bool ret = False;
        if(comp == null) {
        }
        else if(btype == "SELECTION") {
            comp.runMethod(command);
            ret = comp.isTerminate();
        }
    }
}
```



```

        ... //handle another case
        ...
        return ret; //True if not terminate
    }
    String getPC() { ... }
}

```

To traverse the root, execute corresponding method in its reference class. If the tree has no sign of termination, traverse each child node⁴, and run every child behavior. This must be done until there is no more child node to be executed.

```

interface T {
    Unit execute();
    Unit setRoot(N n);
}
class Tree(N r) implements T {
    N rt = r;
    Unit execute() {
        this.executes(r);
    }
    Unit executes(N r) {
        Bool b = r.runNode();
        if (not(b)) {
            ...//execute asynchronously each child node of r
        }
    }
    Unit setRoot(N n) { ... }
}

```

Main Block

Main block is needed for execution. In the main block, all component objects, and the structure of the tree are generated. The tree is built automatically, so every node need to be declared and set them as a child of the parent node. The label is used as a node name⁵. For example, node with label 00 is named “node00”. “node00” is the ancestor of node with label “000” and “001”, so node00 adds “node000” and “node001” as its child.

2.4.4 Integrated BT2SAL Translator and SAL Model Checker

In this project, BT2SAL translator and SAL Model Checker are tools that have important role to build “verify model” feature. “Verify model” feature is used to verify the correctness and consistency of modeled requirement in behavior tree. When modeled requirement, or commonly called model, is verified, users can ensure that their model is safe, correct and consistent.

Model verification is one of the strengths of requirement modeling using behavior tree. This is the uniqueness that owned by behavior tree and makes behavior tree is especial. This chapter will

⁴ Since the node type can be parallel block, the node can contains more than one child.

⁵ Variable name will be unique, for it used label that assigned uniquely in every BT node.

describe this uniqueness side of behavior tree: how verification works. For this purpose, we will explain in detail about Behavior Tree Specification, SAL Model Checker, SAL Specification, LTL, BT2SAL Translator, and Model Verification Process Example. We will use “R2 Train” simple problem to make explanation easier to understand. Before, Figure 11 is the big picture of verification tools and its specification file.

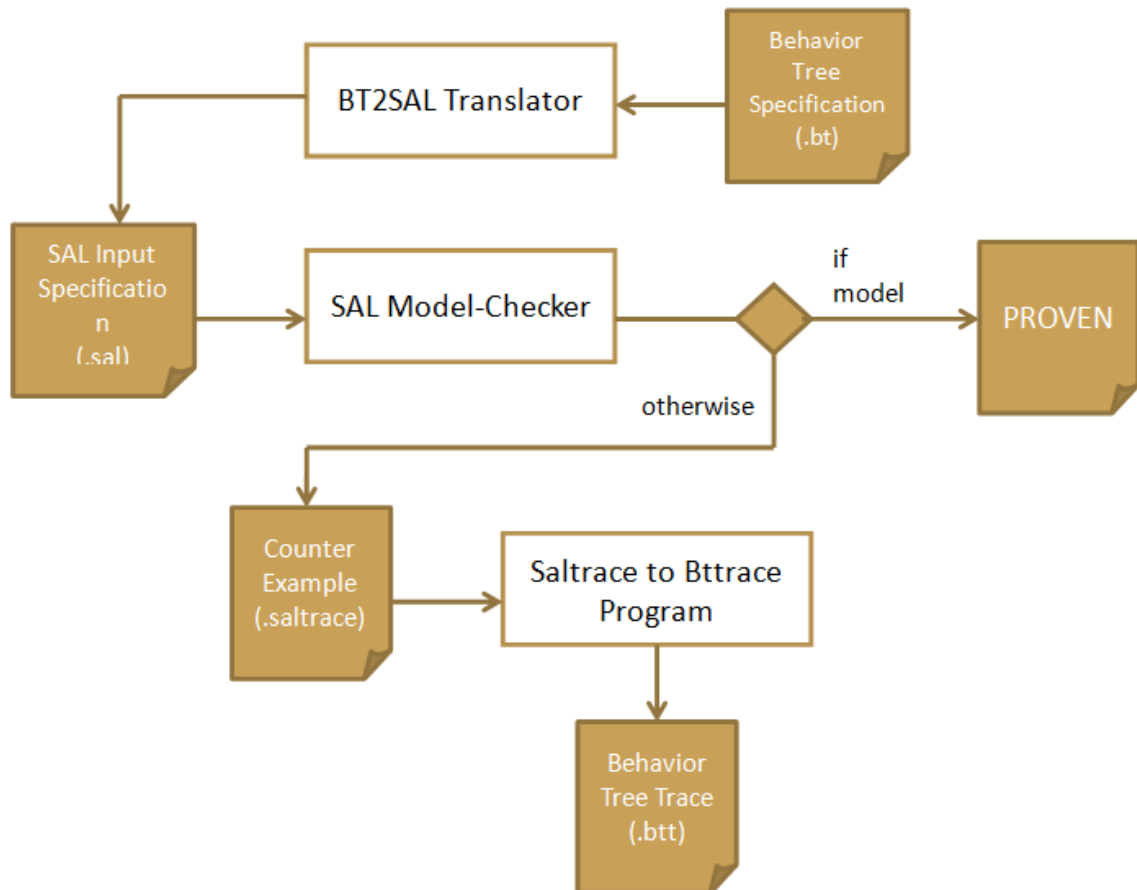


Figure 11 Tool and Specification Integration Design

2.4.4.1 Behavior Tree Specification

Behavior tree is graphical representation that has form like a tree to model requirements formally and visually. Behavior tree formed by entities that are connected each other in a tree structure. Each entity has definition of component and behavior. An entity can reach a state, change state, make decision, give or cause an event, and interact with other entities through the exchange of information and control.

Behavior tree is designed to help people understand the actual intent of the requirements. Software requirements written in natural language for large-scale system have big details. It is certainly difficult to get a deep and comprehensive understanding of the system. Moreover, natural language which is the language for requirements has the possibility likely to be understood

differently, ambiguous, inconsistent, redundancy, and incomplete. Behavior tree with formal semantics based on a new process algebra definition comes to resolve that problems.

Figure 12 shows simple example of behavior tree for “R2 train” problem.

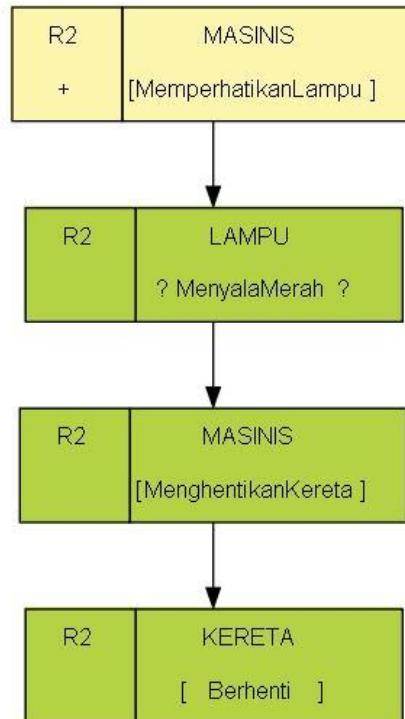


Figure 12 Example of Behavior Tree

Example of behavior tree is taken from one of requirements named R2, “if the signal turns red, the train conductor stops the train”. Now we have component named LAMPU (signal), MASINIS (train conductor), and KERETA (train). Then behavior for each component is defined. LAMPU has behavior MenyalaMerah (TurnRed), KERETA has behavior Berhenti (Stop), and MASINIS has behavior MemperhatikanLampu (noticeSignal) and MenghentikanKereta (StopTrain).

Behavior has certain type. Behavior is enclosed by symbol that states behavior type. For example, behavior of KERETA is enclosed by “[]” symbol. This symbol indicates State Realization. In State Realization state, a component realizes its behavior. It is a state reached by component. Figure 13 shows more behavior types available in behavior tree.

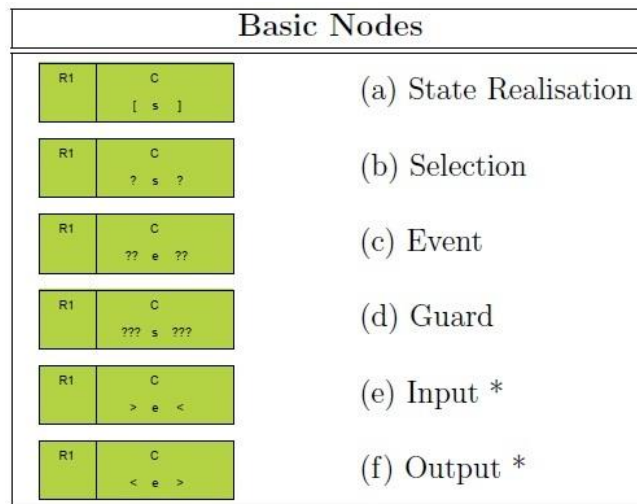


Figure 13 Behavior Types

Modeled requirement in behavior tree can be represented in a specification. It is called textual representation of behavior tree and is saved to the file with .bt extension. Specification of behavior tree in "train" problem can be written as follows.

```
#RT R2 R2

#C C1 KERETA
#S 1 Berhenti

#C C2 MASINIS
#S 1 MemperhatikanLampu
#S 2 MenghentikanKereta

#C C3 LAMPU
#L 1 MenyalaMerah

#T R2 C2 1; R2 C3 1; R2 C2 2; R2 C1 1
```

Behavior tree specification defines requirement list first. It can be declared with #RT symbol.

```
#RT R2 R2
```

The line of code means requirement list of behavior tree. In "R2 train" problem, we just use one requirement namely R2. In "#RT R2 R2", the first R2 indicates reference for tree structure writing. The second one indicates value for requirement name in graphical representation of behavior tree.

After requirement list declaration, all components and its behaviors must be defined. In behavior tree specification above, definition of component MASINIS (train conductor) is written as follows.

```
#C C2 MASINIS
#S 1 MemperhatikanLampu
#S 2 MenghentikanKereta
```

The #C flag indicates that it is component definition. This component definition have “C2” as reference and “MASINIS” as value also component name at the same time. Then behaviors of component are defined. In “#S 1 MemperhatikanLampu”, Symbol “#S” indicates “state realization” behavior type, “1” indicates behavior reference, and “MemperhatikanLampu” is behavior value or name. Here is complete behavior symbol and its meaning.

Table. Behavior Symbol and Its Meaning in BT Specification

Behavior Symbol	Meaning
#S	State realisation
#L	Selection
#E	Event
#G	Guard
#EI	External Input Event
#EO	External Output Event
#II	Internal Input Event
#IO	Internal Output Event
#A	Assertion

In the last part of behavior tree specification is tree structure writing. This tree structure writing aims to represent graphical behavior tree. It is started by symbol "#T" and then followed by component, behavior, and requirement forming in tree.

```
#T R2 C2 1; R2 C3 1; R2 C2 2; R2 C1 1
```

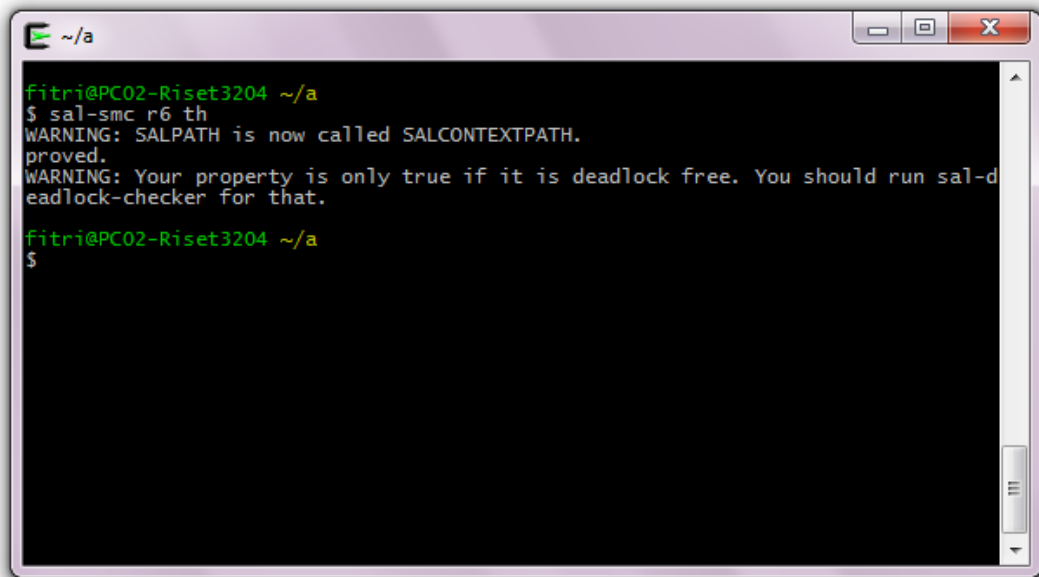
2.4.4.2 SAL Model Checker

SAL is a framework which consists of several tools for abstraction, program analysis, theorem reasoning, and model checking. SAL is used to check the property symbolically of a transition system. SAL traverses and checks every path in the transition system.

The SAL framework has specific language created to describe the transition of the system. SAL language consists of expressions, transition languages, modules, asynchronous and asynchronous composition of modules, and system specifications. Using this language, the property of the system can be defined according to the requirements. A more detailed explanation of the SAL language is available in the SAL Language Report.

The specification of the SAL language will be an input for a tool named SAL Model-Checker (sal-smc). SAL Model-Checker will check the property that needs to be satisfied by the system. If the property is satisfied, the tool will inform that the property of the system is satisfied. Otherwise, the tool will produce counter-example that shows the path of the system that cause the property unsatisfied. In that counter-example, the variable value of the system for each state of the path is shown. This counter-example is also called Saltrace.

SAL Model Checker is actually running on Linux environment. To meet with Windows environment, this tool needs Cygwin. Cygwin is a collection of tools which provide a Linux look and feel environment for Windows. Figure 14 shows screenshot of SAL Model Checker on Cygwin.



```
fitri@PC02-Riset3204 ~/a
$ sal-smc r6 th
WARNING: SALPATH is now called SALCONTEXTPATH.
proved.
WARNING: Your property is only true if it is deadlock free. You should run sal-d
eadlock-checker for that.

fitri@PC02-Riset3204 ~/a
$
```

Figure 14 Screenshot of SAL Model Checker on Cygwin

2.4.4.3 SAL Specification

SAL Language is part of the SAL framework. This language has specific specification to describe transition system. SAL input specification applies this language. It will be input for SAL Model Checker for property checking.

SAL input specification is saved to the file with .sal extension. This specification is related to textual representation of behavior tree. It has two main parts, global declaration and module. Global declaration consists of variable declarations that apply globally, while module consists of specification for system transition.

Below is example of SAL specification from textual representation of behavior tree from “R2 train” problem.

```
kereta:CONTEXT=
BEGIN
Lampu: TYPE={lampu_menyalamerah};
Masinis: TYPE={masinis_memperhatikanlampu, masinis_menghentikankereta};
Kereta: TYPE={kereta_berhenti};

behavior:MODULE=
BEGIN

LOCAL
lampu: Lampu,
```

```

masinis: Masinis,
kereta: Kereta,
pc1: [0..5]

INITIALIZATION
pc1= 1;
pc2= 0;
pc3= 0;

TRANSITION
[
A1:pc1=1
--> masinis'=masinis_memperhatikanlampu;
pc'=2;
[]

A2:pc1=2 AND lampu=lampu_menyalamerah
--> pc1'=3;
[]

A3:pc1=3
--> masinis'=masinis_menghentikankereta;
pc1'=4;
[]

A4:pc1=4
--> kereta'=kereta_berhenti;
pc1'=5;
[]

A5:pc1=2 AND NOT (lampu=lampu_menyalamerah)
--> pc1'=0;
]
END; %end of module
END

```

2.4.4.4 LTL (Linear-time Temporal Logic)

Linear-time Temporal Logic or LTL is a logic which the truth value depends on time. Truth value has a dynamic property, meaning that its value fluctuates in a system. LTL uses linear-time. Linear-time is used for a deterministic system that has a definite path. LTL also uses discrete time, meaning that the system uses a certain time unit, for example hours, minutes, seconds, or milliseconds. It is not necessary to observe the system continuously.

The syntax of LTL in the Backus-Naur Form (BNF) is as follows.

$$\begin{aligned} \varphi ::= & T \mid \perp \mid p \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid \\ & (X\varphi) \mid (F\varphi) \mid (G\varphi) \mid (\varphi U \varphi) \mid (\varphi W \varphi) \mid (\varphi R \varphi) \end{aligned}$$

Where T, \perp, p are atomic proposition and φ is a formula in LTL.

There are two important terms in LTL, the model and path. Model is a triplet $m = (S, \rightarrow, L)$. S is a set of states, \rightarrow is a transition relation of a state so that a state can have a relation with another state, or have a relation to it. L is the labeling function from set S to set P (atomic). $L(s)$ is a set of atomic propositions that are valued true. Path in a model M is a sequence of states where the next state can be accessed from the current state. Formally, path is defined as follows: a set of path satisfies a formula φ if φ applied to every path in the set.

$$\pi := s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots \text{ where } s_k \in S \text{ for } k = 1, 2, 3, \dots$$

$$\text{And for } i > 1, \pi^j := s_i \rightarrow s_{i+1} \rightarrow s_{i+2} \rightarrow s_{i+3} \rightarrow \dots$$

In implementation, LTL formula is used to define properties of the model which would be checked. It is useful for model verifying need. The LTL formula variable will be taken from SAL specification file. Several typical LTL operators that are commonly used for property checking of a model are as follows.

- $G(p)$: read “*global p*”, defines that p is always true in every state.
- $F(p)$: read “*p in the future*”, defines that p will be true sometime in the future in a state.
- $U(p, q)$: read “*p until q*”, defines p must be true until it meets a state where q is true.
- $X(p)$: read “*next p*”, defines p is true in the next state.

Based “R2 train” problem and its SAL specification, we can define properties to be checked. Here are examples.

$F(kereta=kereta_berhenti)$

It means “*kereta*” variable will have “*kereta_berhenti*” state in the future. If it is satisfied, the property of model is correct and safe based on above LTL formula.

2.4.4.5 BT2SAL Translator

BT2SAL Translator is a tool to translate from behavior tree specification into SAL input specification automatically. BT2SAL Translator maps each node of the tree from behavior tree specification into SAL code using translation rules that have been defined before. This tool integrates SAL Model Checker for model checking need. After BT2SAL Translator finishes its translation, it will produce output in form of SAL file. Then the SAL file will be processed by SAL Model Checker.

BT2SAL Translator implements defined translation rules to maps behavior tree specification to SAL input specification. A more detailed explanation of the implemented rules by BT2SAL Translator is available in the thesis [] chapter 3.

In implementation, BT2SAL Translator is implemented in one Java project. Figure 15 shows the screenshot of structure of project.

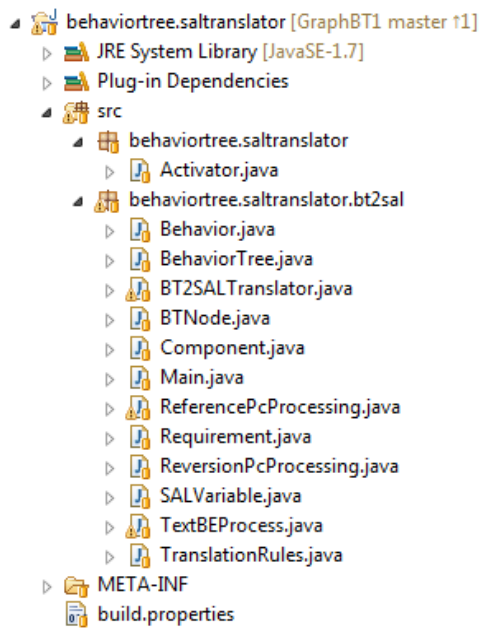


Figure 15 Structure of BT2SAL Translator Project

2.4.4.6 Model Verification Process Example

In model verification process, a behavior tree specification is translated to SAL input specification using BT2SAL Translator. Before, we can see example of behavior tree specification for “R2 train” problem in section “Behavior Tree Specification”. Translation result can be found in section “SAL Specification”. For model verification need, users must define properties of model in SAL specification to be checked using LTL formula.

The property that needs to be checked in the system is coded in the LTL formula. In section “LTL (Linear-time Temporal Logic)”, we have LTL formula “ $F(kereta=kereta_berhenti)$ ” for “R2 train” problem. For model checking, this LTL formula is added to SAL specification. Then it is processed by SAL Model Checker. If fulfilled, SAL Model Checker produces “PROVEN” text output. Otherwise, counter-example, also called Saltrace, will be generated, and then it will be processed by Saltrace to Bttrace Program to produce bttrace specification. Bttrace specification contains information of path transition from the tree that causes property violation. Path transition is represented by the node numbering.

2.4.5 Debugger and Simulator Tools

Debugger and Simulator Tools for Behavior Tree is a one tool included in GraphBT Project. This tool consists of some features:

- a. Show the Behavior Tree diagram representation from the Behavior Tree Specification (input)
- b. Simulate and animate the execution of Behavior Tree
- c. Store the sequence of simulation execution Behavior Tree in a BT Trace File

2.4.5.1 General Process

The processes of Debugger and Simulator for Behavior Tree is divided into 4 steps, as shown in Figure 16.

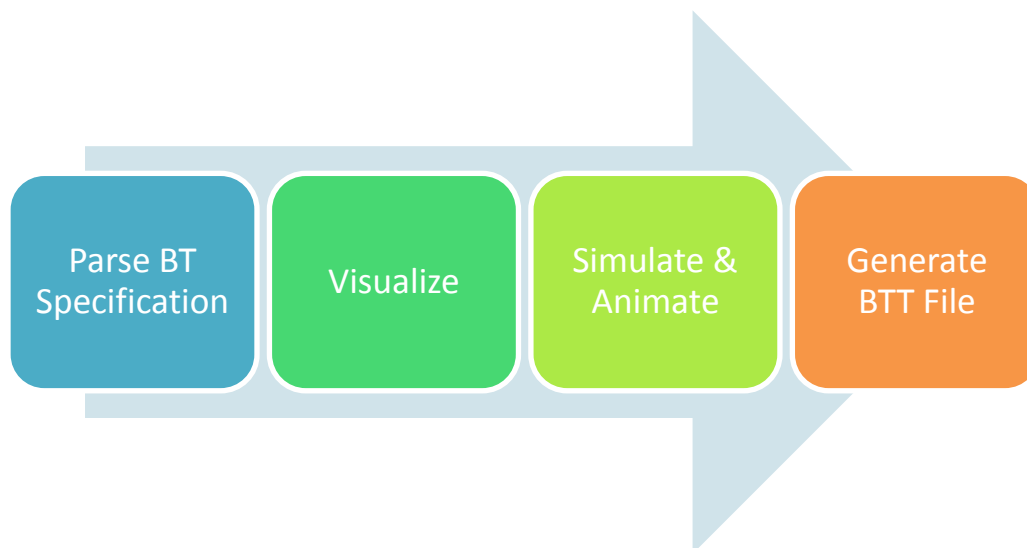


Figure 16 General Process

a. Parse BT Specification

In this process, the program parse a xml file that represent a Behavior Tree Specification.

b. Visualize

The parsing result from previous process will be converted into a tree. This tree then will be visualized in a frame view

c. Simulate & Animate

This process is the main process of BT Debugging Tool, which simulate the execution of the Behavior Tree.

d. Generate BTT File

After the simulation running until completion, the program generates a file BTT. BTT file consist information about the user input, simulation output, and also the sequence of execution node.

2.4.5.2 Architecture of BT Debugging Tool

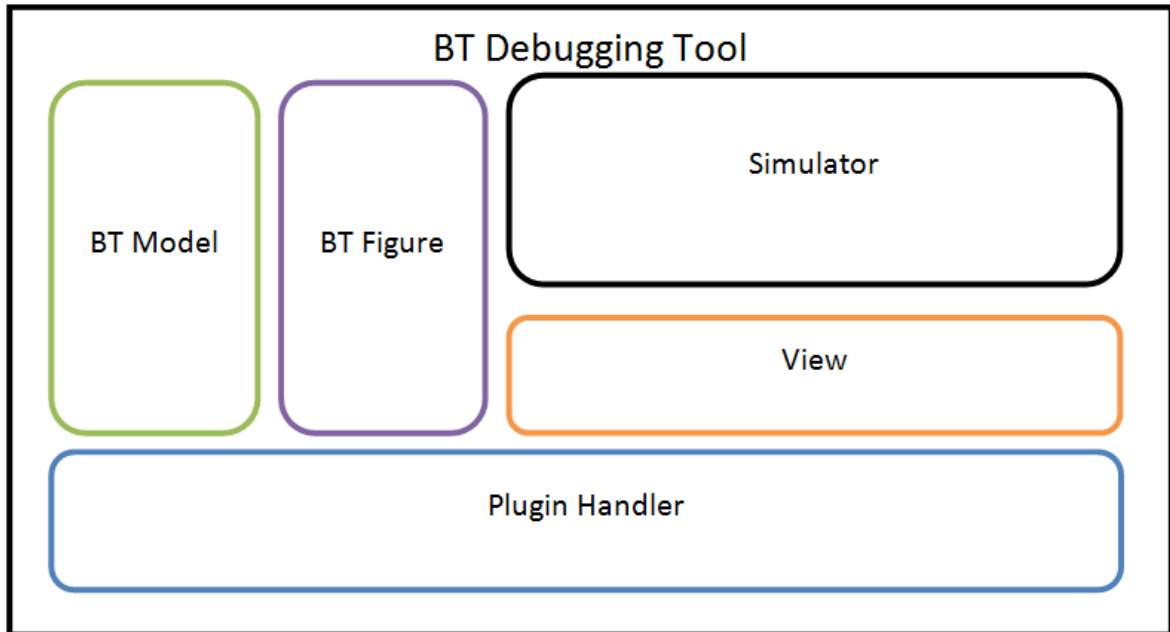


Figure 17 Architecture of BT Debugging Tool

In general, BT Debugger Tool consists of five main modules: **BT Model**, **BT Figure**, **Simulator**, **View**, and **Plugin Handler** (see Figure 17). **BT Model** includes representations of objects from BT Specification, where each of BT Node is visualized as a view figure using **BT Figure** module. Data models and visual representation (figure) will be the input for the module **View** and **Simulator**. Module **View** will handle animated simulation, input and output from the user, display state variable, and displays a collection of figures with simple tree layout algorithm. Module **Simulator** executes the simulation of BT Specification. Finally, the module **Plugin Handler** integrates four functions of other modules by using the Eclipse IDE plugin concept.

2.4.5.3 Implementation

2.4.5.3.1 Parse BT Specification

Below is the format of XML file that represent a BT Specification. This format is the same format which can be handled in BT2ABS parser.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <textbt:TextBT>
3 <requirements>
4     <requirements ref="R1" val="R1" />
5     <!--other requirement-->
6 </requirements>

```

```

7 <components>
8   <components ref="CBUFF" val="BUFFER">
9     <behaviors      xsi:type="textbt:InternalInput"      ref="1"
val="setCapacity(CAPACITY)"/>
10    <behaviors xsi:type="textbt:State" ref="11" val="ADD">
11      <relations question="What" componentRef="CDATA" />
12      <relations      question="Where"      preposition="to"
componentRef="CLIST" />
13      <!-- other relation -->
14    </behaviors>
15    <!-- other behavior-->
16  </components>
17  <!-- next component-->
18</components>
19<behaviorTree>
20  <rootNode componentRef="CBUFF" behaviorRef="1">
21    <childNodes      xsi:type="textbt:SequentialNode"      componentRef="CBUFF"
behaviorRef="11">
22      <childNodes      xsi:type="textbt:SequentialNode"
componentRef="CBUFF" behaviorRef="1">
23        </childNodes>
24      </childNodes>
25      <childNodes      xsi:type="textbt:SequentialNode"      componentRef="CBUFF"
behaviorRef="1">
26        </childNodes>
27    </rootNode>28</behaviorTree>
29</textbt:TextBT>

```

Source Code 0.1 Format xml BT Specification

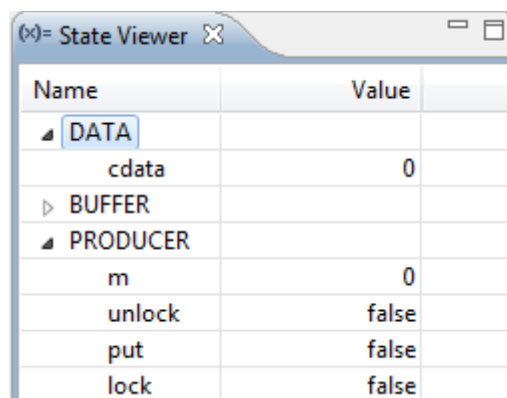
Because it uses the same input format, for the parser implementation is also using the same parser implementation of BT2ABS tool. This parser then instantiates a behavior tree object (in BT Model module) that represent that BT Specification.

2.4.5.3.2 Behavior Tree Visualization

There are 3 viewers included in this process:

a. State Viewer

State Viewer's function is to show the value of variable from BT, as shown in Figure 18.



Name	Value
DATA	
cdata	0
BUFFER	
PRODUCER	
m	0
unlock	false
put	false
lock	false

Figure 18 Illustration of State Viewer

The implementation involves JFace Tree Viewer to display the state variables of each BT Component. JFace Tree is a widget that can display a list of data that is expandable. Figure 19 is a condition of the State Viewer, showing some of the variables that are grouped according to the BT Component's name. In the example image, there are 3 pieces BT Component, namely DATA BUFFER, and PRODUCER. Each BT Component can be expanded so that it displays all existing variable.

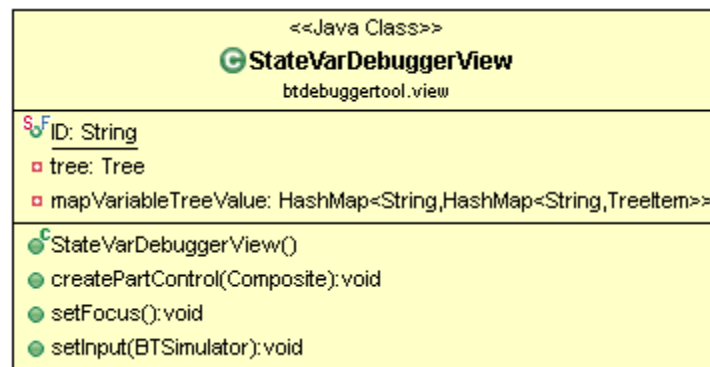


Figure 19 Class Diagram of StateVarDebuggerView

Input from the State Viewer, which is implemented by the class `StateVarDebuggerView`, is `BTSimulator`, which is a class to simulate the execution of the Behavior Tree. `BTSimulator` is used to monitor the changes of value of each state variable of BT Component, so it can be invoked directly when the simulation is running. To accommodate this, there is a listener, `DataListener`, which must be registered to the `BTSimulator` instance. In this listener, there is method that must be implemented on `StateVarDebuggerView`.

```

1 //register the listener to monitor the variable's value
2 simulator.setOnDataChangeListener(new DataListener() {
3     @Override
4     public void onDataChanged(String componentName, String variableName, String
value) {
5         final TreeItem item =
mapVariableTreeValue.get(componentName).get(variableName);
6
7         item.setText(new String[]{variableName, value});
8         item.setForeground(ColorConstants.red);
9
10    }
11});
  
```

Source Code 0.2 How to register DataListener to BTSimulator at class StateVarDebuggerView

b. Console View

The Implementation of Console View is a feature that has been found on the Eclipse IDE, so no need to implement from scratch. In order to use the feature, add a new dependency: `org.eclipse.ui.console` at `plugin.xml` file of the program (see Figure 20).

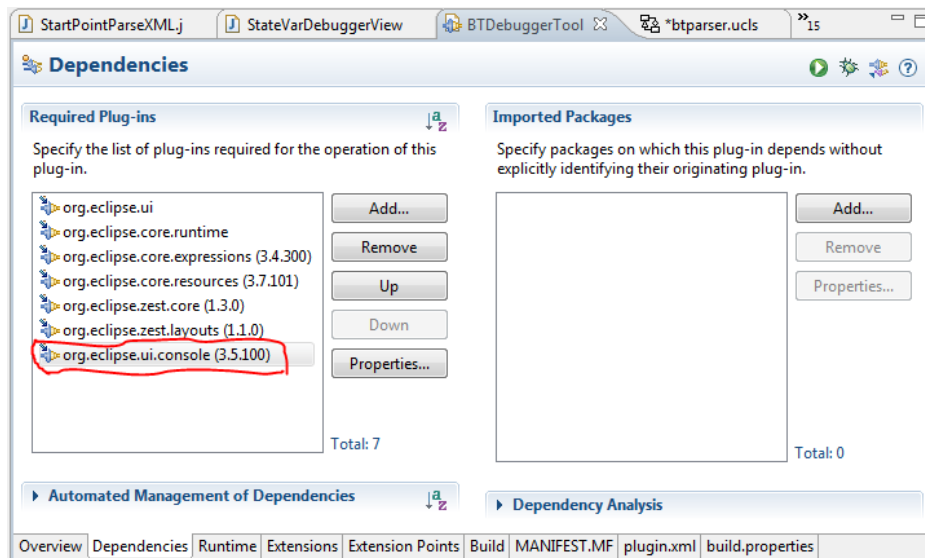


Figure 20 Adding dependency `org.eclipse.ui.console`

Console View on Eclipse Plugin has an instance `MessageConsoleStream`, the component that can write to the output stream when the simulation of BT is running.

```

1  ...
2  //show console view
3  IWorkbenchPage perspectivePage =
PlatformUI.getWorkbench().showPerspective(DEBUGGER_PERSPECTIVE_ID, window);
4  IConsole myConsole = findConsole("btdebuggerTool.view.consoleView");
5  IConsoleView consoleView =
(IConsoleView) perspectivePage.showView(IConsoleConstants.ID_CONSOLE_VIEW);
6  consoleView.display(myConsole);
7
8  //get output stream instance from Console View
9  ((MessageConsole)myConsole).clearConsole();
10 MessageConsoleStream out = ((MessageConsole)myConsole).newMessageStream();
11 ...
12 private MessageConsole findConsole(String name) {
13     ConsolePlugin plugin = ConsolePlugin.getDefault();
14     IConsoleManager conMan = plugin.getConsoleManager();
15     IConsole[] existing = conMan.getConsoles();
16     for (int i = 0; i < existing.length; i++)
17         if (name.equals(existing[i].getName()))
18             return (MessageConsole) existing[i];
19     //if console with name = name is not found, then make new console with
that name, and register it to Eclipse plugin
20     MessageConsole myConsole = new MessageConsole(name, null);
21     conMan.addConsoles(new IConsole[] {myConsole});
22     return myConsole;
23 }

```

Source Code 0.3 Implementation of *Console View*

c. Zest Viewer

To assist users in analyzing the execution of the Behavior Tree, the animator is designed which can animate and simulate the Behavior Tree. The part to be in charged on initiating the animation and simulation is Zest Viewer. Before it was done, Zest Viewer's duty is to show the graphical representation of the Behavior Tree and adjust the position of each node of the tree (see Figure 21).

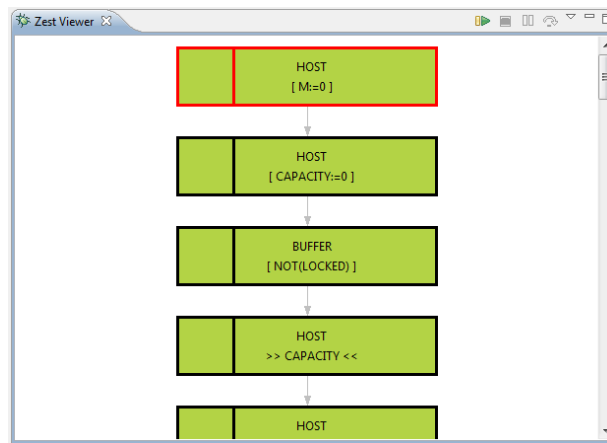


Figure 21 The Zest Viewer on tree layouting and showing the BT Diagram

2.4.5.3.3 BT Simulation and Animation

2.4.5.3.3.1 Animation

Active Node will be the center of attention of the user to animate the center of the screen. The implementation exists in the method `setToCenter` at class `ZestDebuggerView`.

```

1 private void setToCenter(final BTNode node, final int animationDelay){
2     BTGraphNode graphNode = this.mapDataToGraph.get(node);
3     if(graphNode!=null){
4         org.eclipse.swt.graphics.Rectangle rectParent = this.graph.getBounds();
5         Rectangle rectNode = graphNode.getFigure().getBounds();
6         int centerXNode = rectNode.x + rectNode.width / 2;
7         int centerYNode = rectNode.y + rectNode.height / 2;
8
9         int centerXParent = rectParent.x + rectParent.width / 2;
10        int centerYParent = rectParent.y + rectParent.height / 2;
11
12        final int selisihX = centerXNode - centerXParent;
13        final int selisihY = centerYNode - centerYParent;
14        Display.getDefault().asyncExec(new Runnable() {
15
16            @Override
17            public void run() {
18                Animation.markBegin();
19                List nodes = graph.getNodes();
20
21                for (Object itemNode : nodes) {
22                    Point p = ((GraphNode)itemNode).getLocation();
23                    ((GraphNode)itemNode).setLocation(p.x - selisihX, p.y
24 - selisihY);
25                }
26            }
27        });
28    }
29 }

```

```

25             Animation.run(animationDelay);
26
27         }
28
29     });
30 }else{
31     System.out.println("Set To Center is failed");
32 }
33}

```

Source Code 0.4 Implementation of setToCenter method

Source Code 0.4 calculates the translation's value, in each vertical and horizontal direction, which is needed in order for a node positioned at the center of the screen provided. This is done by calculating the location of the center point of the screen (lines 9 and 10) and the location of the point node (lines 6 and 7), and then calculates the difference between the two point vertically and horizontally. Then the translation will be done on all nodes that exist in the graph (tree), which is on the line to 18 to 25. The process of translation is animated with a delay of value animationDelay.

There are 3 *methods* that implement the coloring of each nodes:

```

1  private void setActiveNode(BTNode node, int animationDelay) {
2      BTGraphNode graphNode = this.mapDataToGraph.get(node);
3      if(graphNode!=null) {
4          graphNode.getFigure()
5              .setBackgroundColor(BTColorConstanta.CURRENT_ACTIVE);
6      }
7  private void setInterleaveNodes(ArrayList<BTNode> nodes, int animationDelay) {
8      for (BTNode node : nodes) {
9          BTGraphNode graphNode = this.mapDataToGraph.get(node);
10         if(graphNode!=null) {
11             graphNode.getFigure()
12                 .setBackgroundColor(BTColorConstanta.INTERLEAVE_ACTIVE);
13         }
14     }
15 }
16 private void setNonActiveNode(BTNode node, int animationDelay) {
17     BTGraphNode graphNode = this.mapDataToGraph.get(node);
18     if(graphNode!=null) {
19         graphNode.getFigure().setBackgroundColor(BTColorConstanta.NOT_ACTIVE);
20     }
21 }

```

Source Code 0.5 Node Coloring implementation: active, inactive, dan interleave node

BTColorConstanta.CURRENT_ACTIVE is a yellow constant,
 BTColorConstanta.INTERLEAVE_ACTIVE is a light green constant, and
 BTColorConstanta.NOT_ACTIVE is a green constant

2.4.5.3.3.2 Simulation

Simulation implementation is put on BTSimulator class (Figure 22).

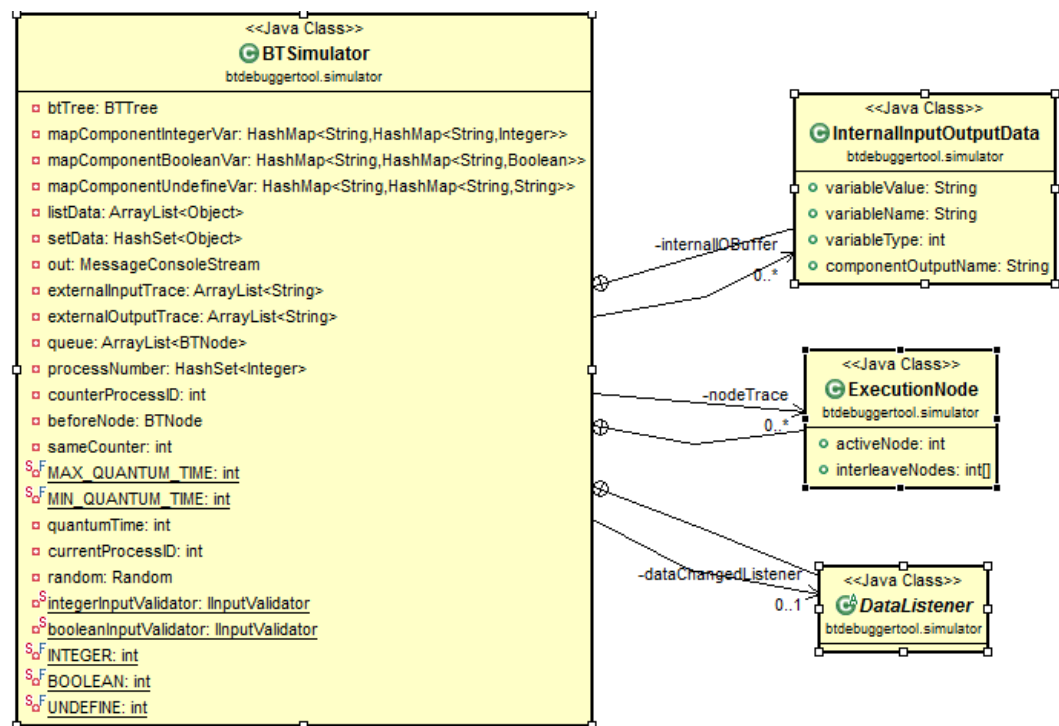


Figure 22 Class Diagram of BTSimulator

In **BTSimulator** there are Behavior Tree data stored on `btTree` attributes. To save the state of variables, there are 3 pieces of `HashMap <String, HashMap <String, T>>`, i.e: `mapComponentIntegerVar`, `mapComponentBooleanVar`, and `mapComponentUndefineVar`. Each of them storing a `HashMap` with the *key* in the form of a variable name, variable value as the *value*.

Attribute `queue` serves as a `process_queue`. Attribute `processNumber` stores the process id, active or waiting-for-its-turn node. The value of `counterProcessID` is given to new process, that is when the simulation meets the *parallel branch*, and value of `counterProcessID` will be incremented by one.

In the **BTSimulator**, there is a **InternalInputOutputData** class. This class serves as a wrapper in the message-passing when there is the execution of the internal input and output events.

ExecutionNode class serves as a wrapper for the purpose of generating a data file BTT, which consists of an array of `interleaveNode` and `activeNode`. These attributes store `BTNode` `nodeNumberForBTT` owned by `BTNode`. This class represents an animated status on BT Trace Animator at a time, i.e there must be 1 piece active node and can have many nodes interleave..

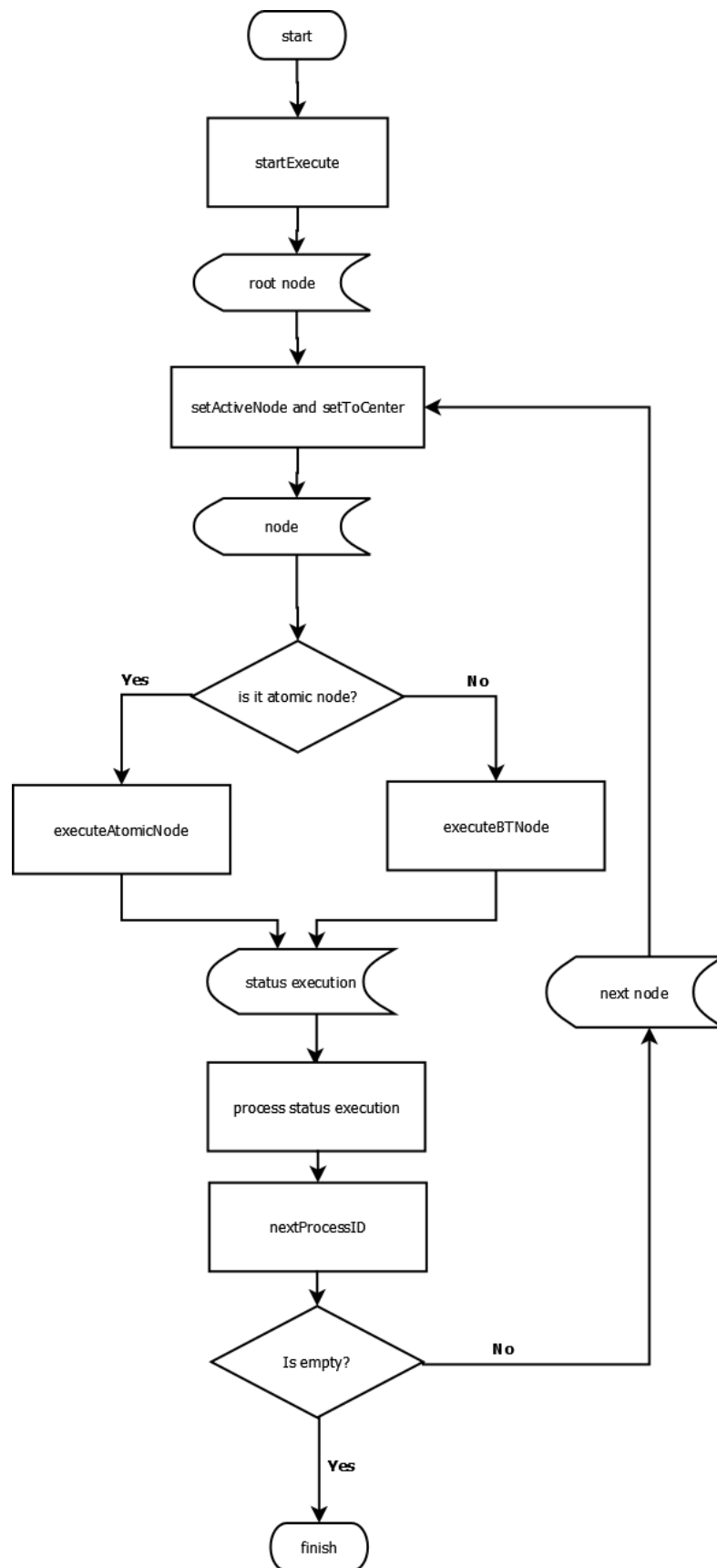


Figure 23 Flowchart diagram of Simulator

Figure 23 is a flowchart diagram that explains the general concept on how the BT simulation is done.

2.4.5.3.4 Generate BTT File

The last process implementation is to produce a BTT file, assisted by doing the recording made during the previous process, Simulate and Animate. User input will be stored at `externalInputTrace`, output of the simulation will be stored at `externalOutputTrace`, and the order of execution will be stored at `nodeTrace` node. All that variable can be accessed on `BTSimulator` class. To save it into BTT file, there is method in `BTSimulator` called `saveTraceToFile` (see Source Code 4.4.1).

```
1 /**
2  * To save btt file into file system
3  * @param pathToFile
4  * @return
5  */
6 public boolean saveTraceToFile(String pathToFile){
7     try{
8         File file = new File(pathToFile);
9         BufferedOutputStream out =
10             new BufferedOutputStream(new FileOutputStream(file));
11         out.write(("INPUT "+externalInputTrace.size()).getBytes());
12         out.write("\n".getBytes());
13         for (String inputElement : externalInputTrace) {
14             out.write(inputElement.getBytes());
15             out.write("\n".getBytes());
16         }
17         out.write(("OUTPUT "+externalOutputTrace.size()).getBytes());
18         out.write("\n".getBytes());
19         for (String outputElement : externalOutputTrace) {
20             out.write(outputElement.getBytes());
21             out.write("\n".getBytes());
22         }
23         out.write("TRACE ".getBytes());
24         for (ExecutionNode execNode : nodeTrace) {
25             int activeNode = execNode.activeNode;
26             int[] interleaveNode = execNode.interleaveNodes;
27             StringBuffer buf = new StringBuffer();
28             buf.append("[ "+activeNode);
29             if(interleaveNode!=null && interleaveNode.length>0){
30                 for(int ii = 0;ii<interleaveNode.length;++ii){
31                     buf.append(", "+interleaveNode[ii]);
32                 }
33             }
34             buf.append("] ");
35             out.write(buf.toString().getBytes());
36         }
37         out.flush();
38         out.close();
39         return true;
40     } catch (Exception e){
41         e.printStackTrace();
42         return false;
43     }
44 }
```

Source Code 0.6 Implementation of saving the record into BTT File

Lines 10-15 writes down the information from the user input. The output of simulation will be written down on the line 16-21. Then the sequence of execution node is on the line 22-35.

III. Application Implementation

3. 1. Implementation Steps

The implementation stage consists of several steps, which are:

1. Developing BT Code Generator for Behavior Tree
This step of development builds Code Generator as an Eclipse plug-in to produce generated Java code from a given BT code.
2. Developing Debugger and BT Execution Simulator
In this step an Eclipse plug-in to Debug and simulate a BT model will be developed.
3. Developing BT Model Checker
This tool is developed to model-check a given BT model as an Eclipse plug-in.
4. Developing Graphical Editor for Behavior Tree
This Eclipse plug-in is used as a graphical development tool for Behavior Tree as the first stage of the development. Later on, this tool will be integrated with another tools listed above. This tool also came along with a textual editor.
5. Tool Integration
Integrating BT Code Generator, Debugger, BT Execution Simulator, and Model Checker with Graphical Editor and Textual Editor as an Eclipse plug-in
6. Documentation
Create documents to supplement the tools, which consists of User Manual, Installation Guide, Release Note, Technical Report, and Final Report

3. 2. Finished Features

1. BT Code Generator
2. BT Model Checker
3. BT Debugger Tool and Execution Simulator
4. GraphBT as Diagram and Textual Editor for Behavior Tree
5. Integrated tool which consists of all the tools listed above

3. 3. Application Result

The result of this is an integrated software development tool for Behavior Tree. The editor is placed in the middle of the window. User can switch the editor from graphical view to text view by clicking the Text tab or vice versa. Palette is positioned in the right side of the window and can be used to provide some features such as creating connection or object to the graphical editor. Some features provided in the GraphBT are select, marquee, create sequential or atomic connection, and create BT Node. Creating a BT Node, for instance, can be performed by clicking General BT Node label from the palette and then click it to the graphical editor. Afterward, a wizard will appear and user can specify Traceability Status Requirement or create or specify the Components, Behaviors, and Requirements of the BT Node. To invoke another features, user can also use the toolbar menu

which contain icons which provides some functionalities such as Create Component, Manage Component, Manage Requirement, Validate Model, Verify Model, Debug and Simulate tool, and Generate Java code. User also can alter an already created BT Node from the Property section.

The screenshot of the explanation of the tool design will be provided [below](#).

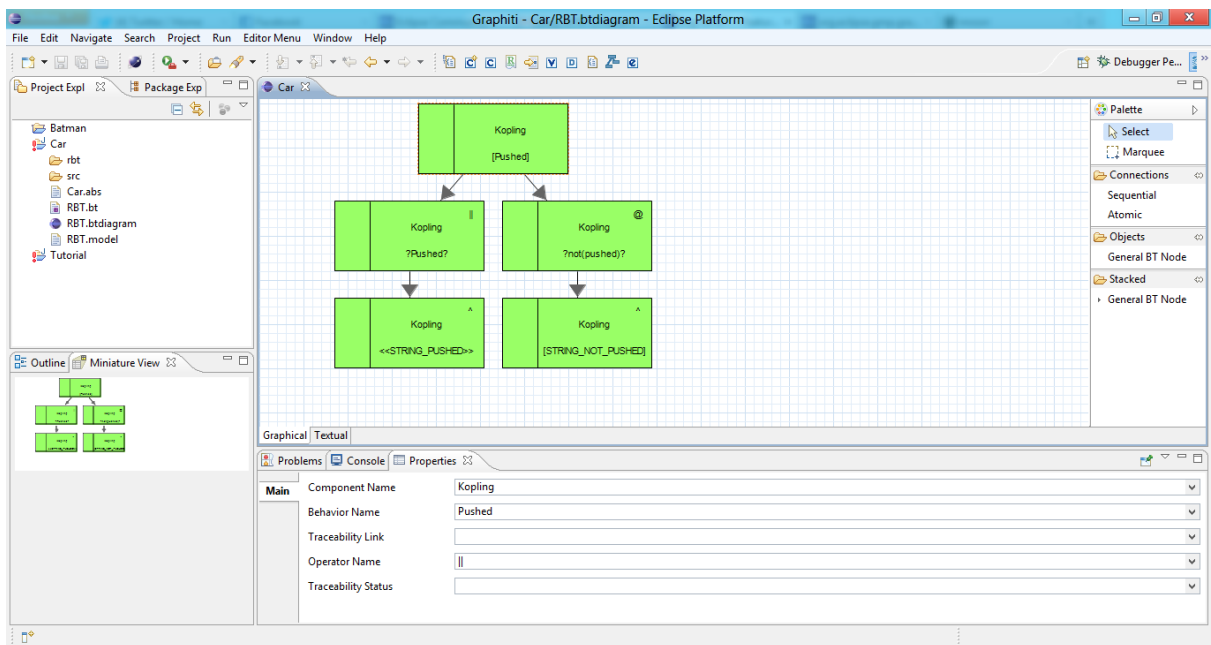


Figure 24 General View

The screenshot shows the Properties view for a BT Node. The view has a tabbed interface with 'Problems', 'Console', and 'Properties'. The 'Properties' tab is active. The 'Main' section contains the following fields:

Field	Value
Component Name	Kopling
Behavior Name	not(push)
Traceability Link	
Operator Name	@
Traceability Status	

Figure 25 Property for BT Node

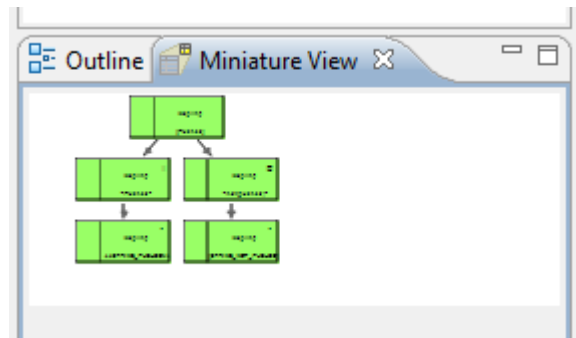


Figure 26 Miniature View

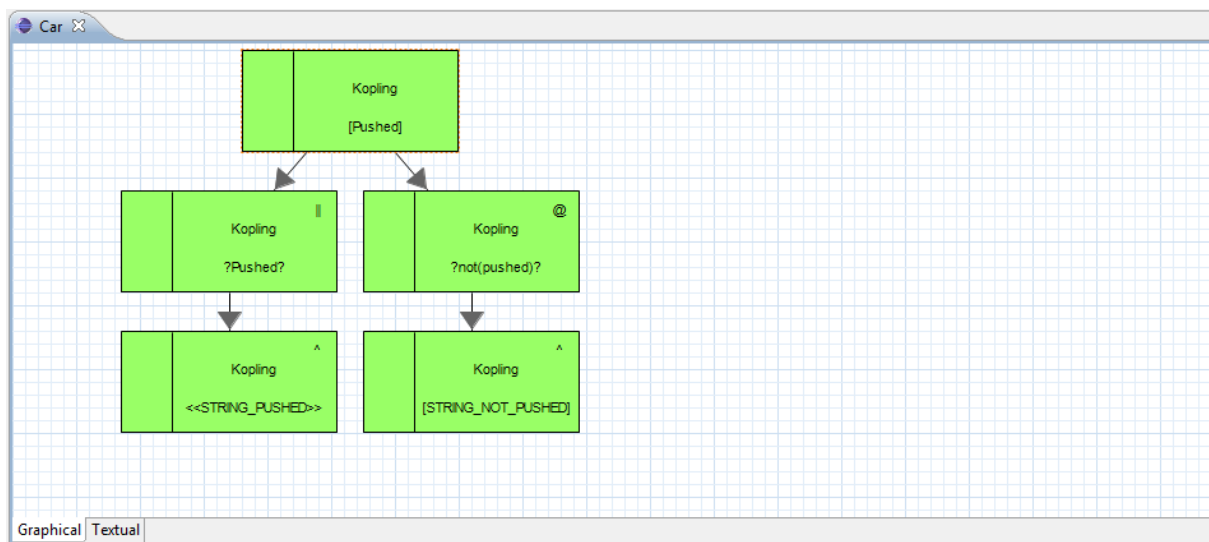


Figure 27 Graphical View

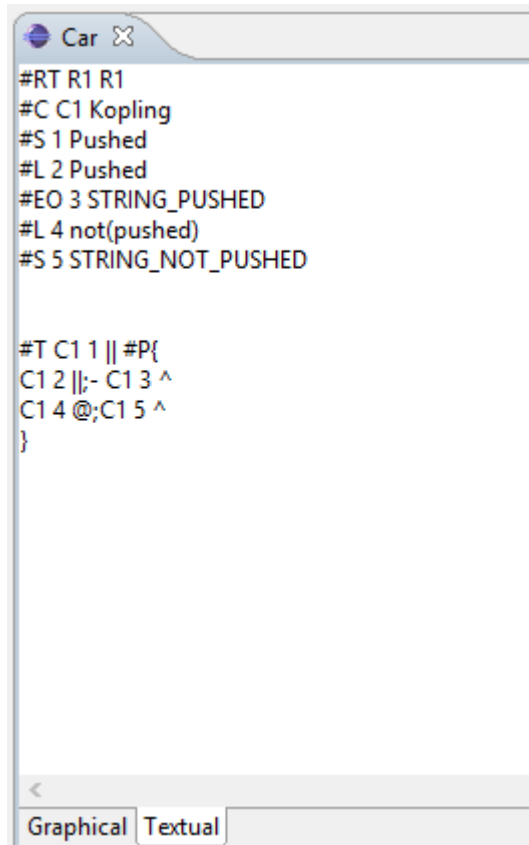


Figure 28 Textual View

In this section we describe features provided in GraphBT:

Add Component

Specifying a new Behavior Tree component in GraphBT can be performed by using add component feature.

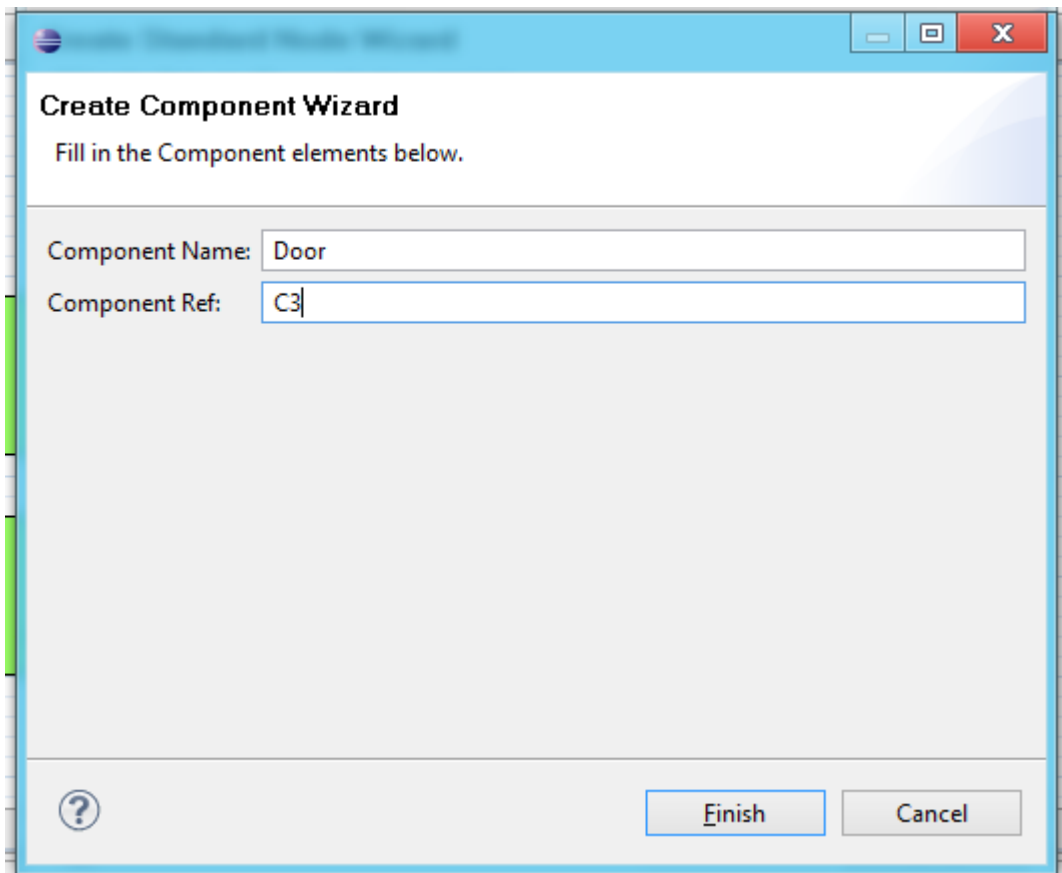


Figure 29 Create Component wizard

Part	Definition
Component Name:	Insert component name
Component Ref:	Insert component reference
Finish	Save component with specified name and reference
Cancel	Cancel creating a new component

Add Behavior

Adding new behavior can be accomplished after the respective component has been specified using Add Behavior feature.

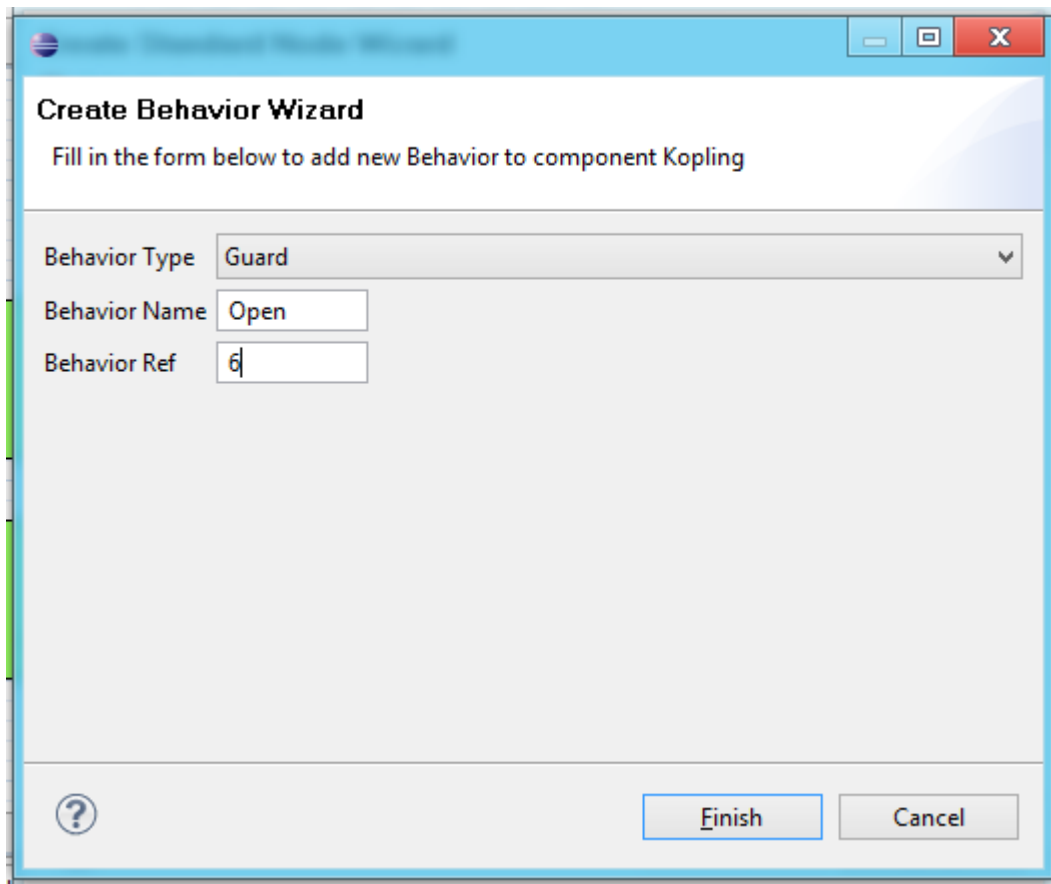


Figure 30 Add Behavior wizard

Part	Function
Behavior Type:	Insert behavior type
Behavior Name:	Insert behavior name
Behavior Ref:	Insert behavior reference
Finish	Save new behavior with specified type, name, and reference
Cancel	Cancel creating a new behavior

Manage Components

BT Components and their respective behaviors can be managed using Manage Component Wizard which can be invoked by pressing Manage Component button in the toolbar.

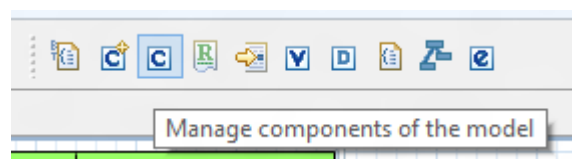


Figure 31 Manage Components icon in the toolbar

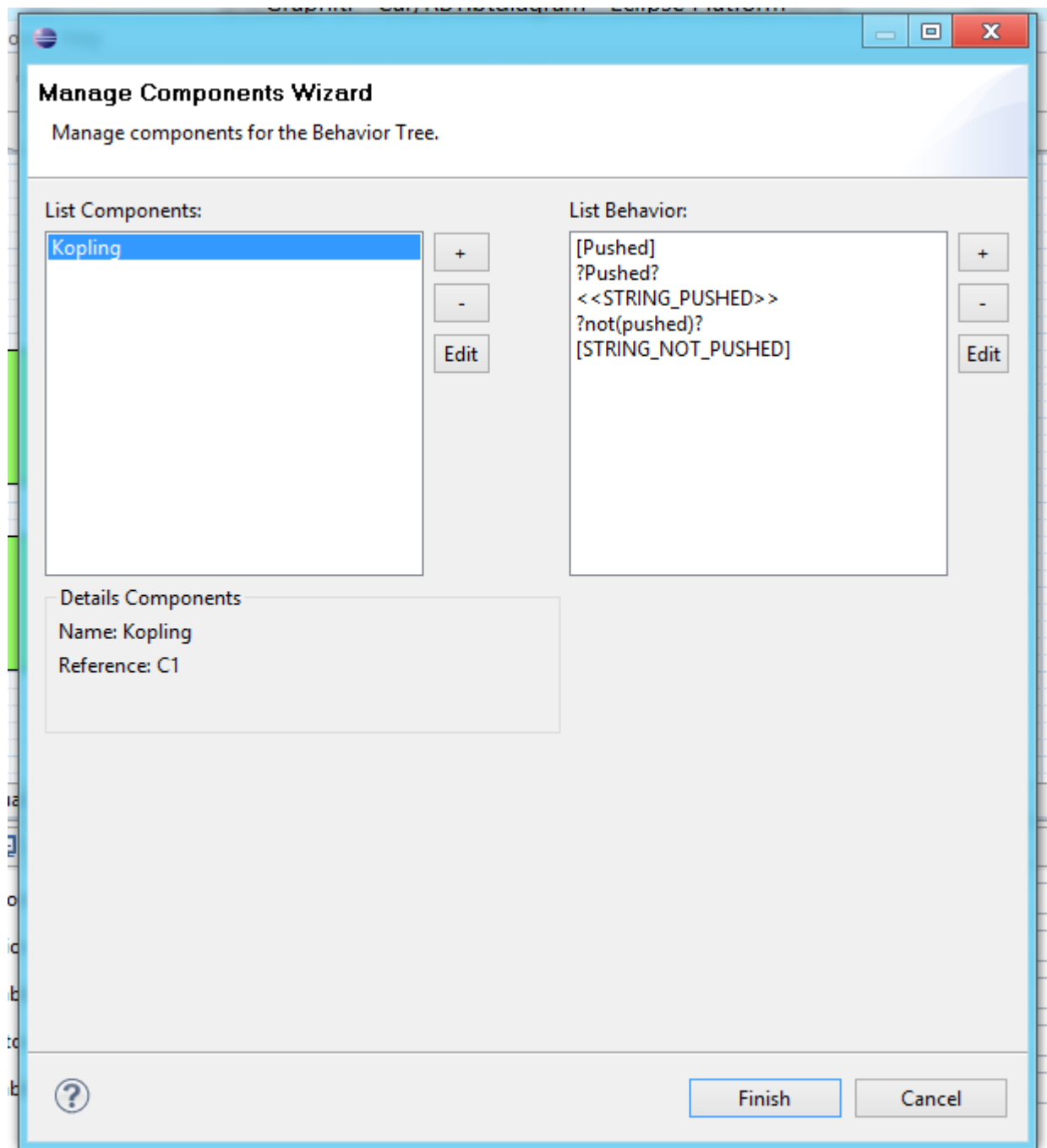




Figure 32 Manage Components wizard

Part	Function
List Components:	List of created components
List Behavior:	List of created behaviors
Details Components	Display the detail information about selected component
	Add new component
	Delete a component

<input type="button" value="Edit"/>	Change component information
<input type="button" value="Finish"/>	Close Manage Components Wizard

Add Requirement

Adding new requirement can be performed in the same fashion with add component feature. It can be done by pressing Add Requirement button of Create BT Node Wizard.

Figure 33 Create Requirement wizard

Part	Function
Requirement Name:	Insert requirement name
Requirement Ref:	Insert requirement reference
Requirement Description:	Insert requirement description
<input type="button" value="Finish"/>	Save new requirement with specified name, reference, and description
<input type="button" value="Cancel"/>	Cancel creating new requirement

Manage Requirement

User can add, modify, and delete requirements using Manage Requirement feature.

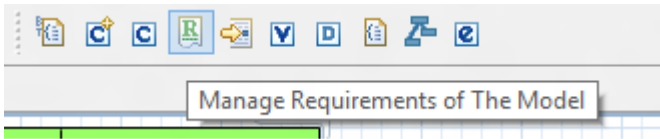


Figure 34 Manage Requirements icon in the toolbar

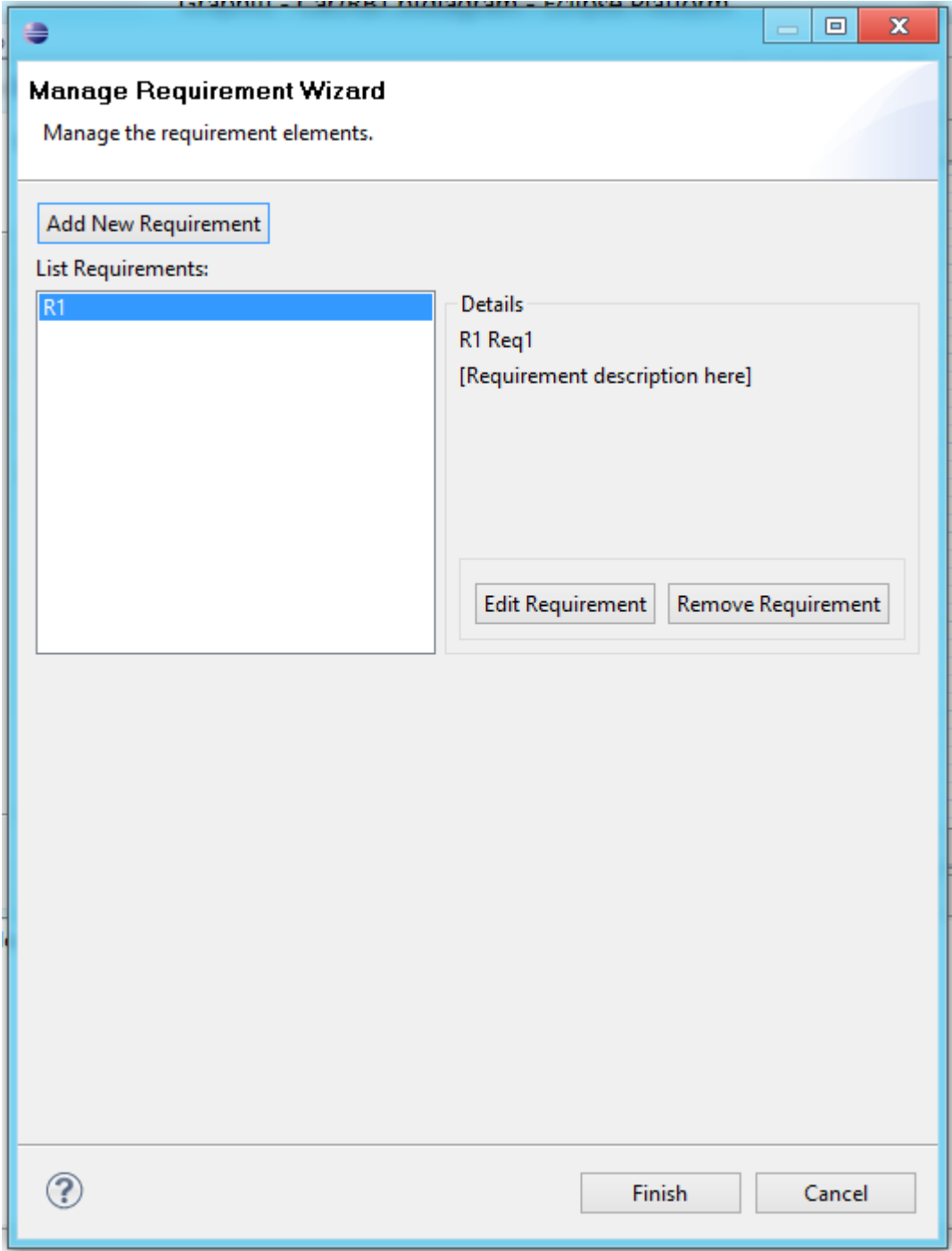


Figure 35 Manage Requirement wizard

Part	Function
------	----------

Add New Requirement	Display a wizard to create a new requirement
List Requirements:	List of created requirements
Details	Display the detail information about selected requirement
Edit Requirement	Edit selected requirement
Remove Requirement	Remove selected requirement
Finish	Close Management Requirement Wizard

Edit Requirement

User can edit any created Requirements by selecting a Requirement from selection list and press Edit Requirement button afterward.

Manage Requirement Wizard
Manage the requirement elements.

Add New Requirement

List Requirements:

R1

Edit Requirement

Name: Req1

Description:

Save

?

Finish Cancel

Figure 36 Edit Requirement view in Manage Requirement Wizard

Part	Function
Name:	Change requirement name
Description:	Change requirement description
Save	Save edited requirement
Finish	Close Manage Requirement Wizard

Add Behavior Tree Node feature

We can add new BT node to the clipboard by clicking General BT Node label in the palette. Add New BT Node wizard will appear. User can select the Component, Behavior, Traceability Link, Traceability Status, and Operator

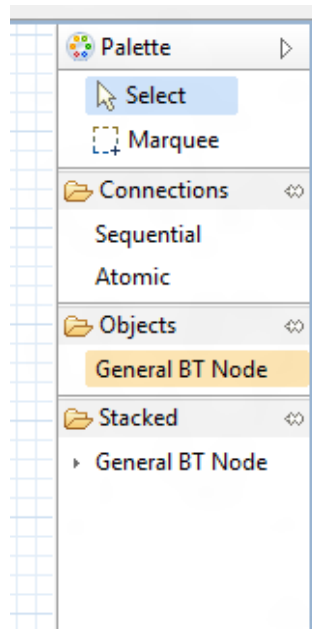


Figure 37 Select General BT Node label in the palette to create a new BT Node

Create Standard Node Wizard

Fill in the Behavior Tree node elements below.

Operator Name: NoOperator

Traceability Link Name:

Traceability Status Name: Original

Component Name: Kopling

Behavior Name: ?not(push)?

Add New Component

Add New Behavior

Manage Components

Add Requirement

Finish Cancel

Figure 38 User can specify a new BT Node in Create Standard Node wizard

Part	Function
Add New Component	Display Create New Component Wizard
Add New Behavior	Display Create New Behavior Wizard
Manage Components	Display Manage Components Wizard
Add Requirement	Display Create New Requirement Wizard
Operator Name:	Insert operator name
Traceability Link Name:	Insert traceability link name
Traceability Status Name:	Insert traceability status name
Component Name:	Insert component name
Behavior Name:	Insert behavior name
Finish	Save a new standard node with specified operator name, traceability link name and status name, component name, and behavior name
Cancel	Cancel creating new standard node

Add Behavior Tree Connection feature

There are two types of connection in Behavior Tree specification: sequential and atomic connection. Both of these connections can be invoked from the connection list in the palette.

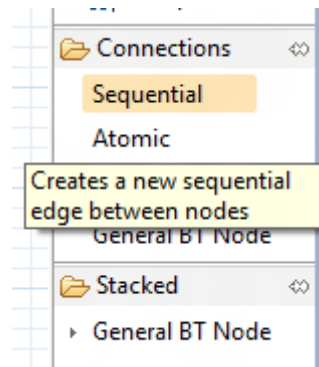


Figure 39 to create an edge between nodes, click Sequential or Atomic label from the palette, then click both nodes

Validate Behavior Tree

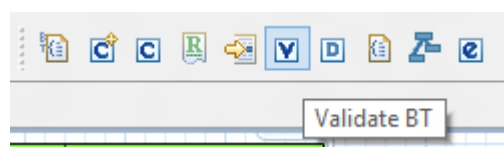


Figure 40 Validate BT icon in the toolbar

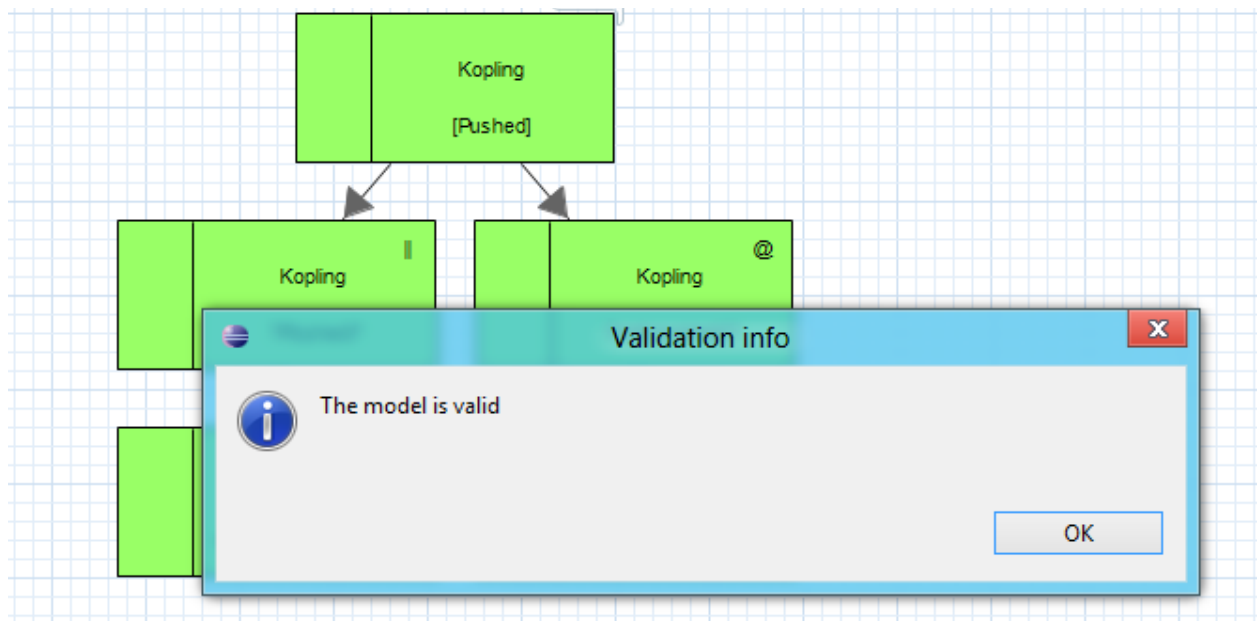


Figure 41 after validating the BT, a message will appear to inform whether the BT is valid or not

Generate BT Code

After the Tree is added to the diagram editor, user can generate the .bt code by pressing Generate BT Code button in the toolbar. The generated BT code will then appear in the Project Explorer.

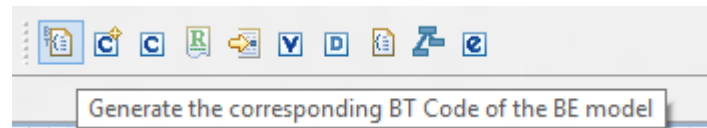


Figure 42 Generate BT code icon in the toolbar

Generate Java Code

Provided that the BT model exists, this feature will generate executable code in Java programming language.

Verify Behavior Tree

User can verify the correctness and consistency of modeled requirement in behavior tree by clicking Verify BT button in the toolbar. For verifying process, user must add LTL formula to check property of behavior tree. The BT verifying result can be saved to Project Explorer.

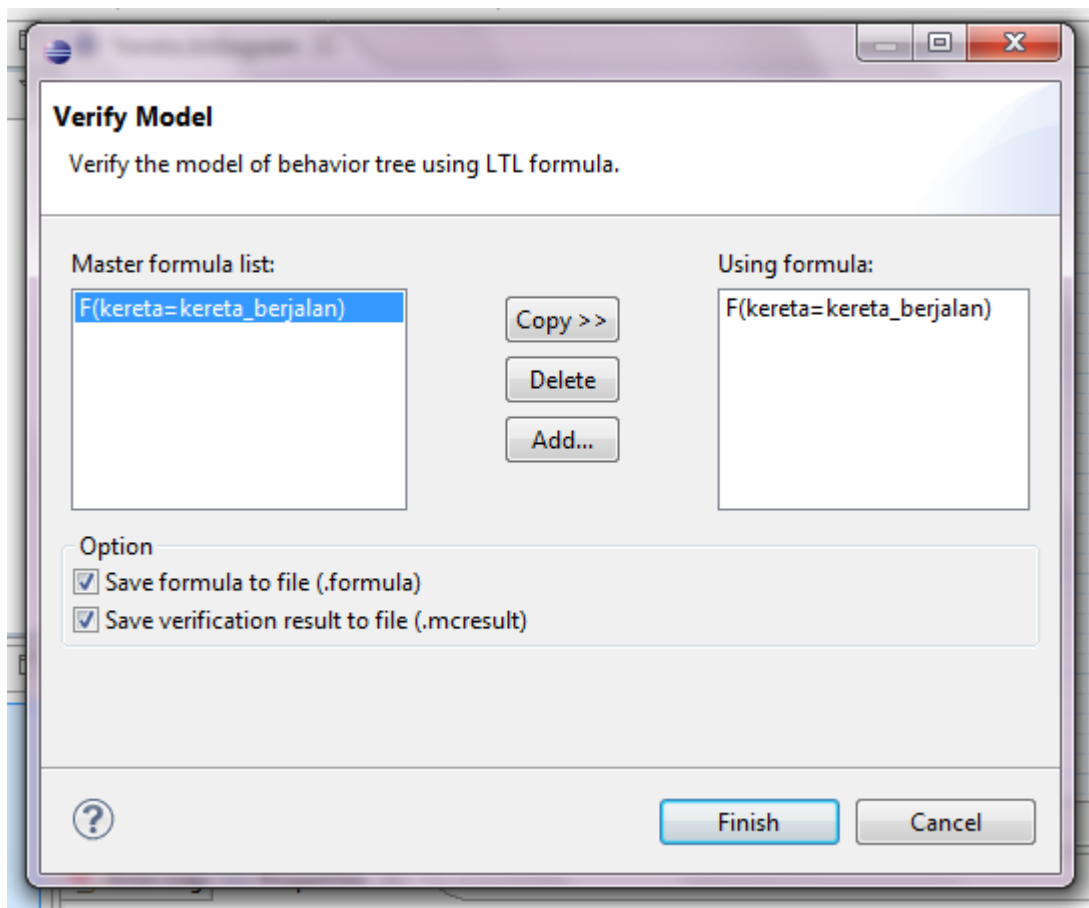


Figure 43 View of Verify Model Wizard

Part	Function
Master formula list:	
Current formula list:	
Copy >>	
Delete	
Edit...	
Add...	
Option	
Finish	
Cancel	

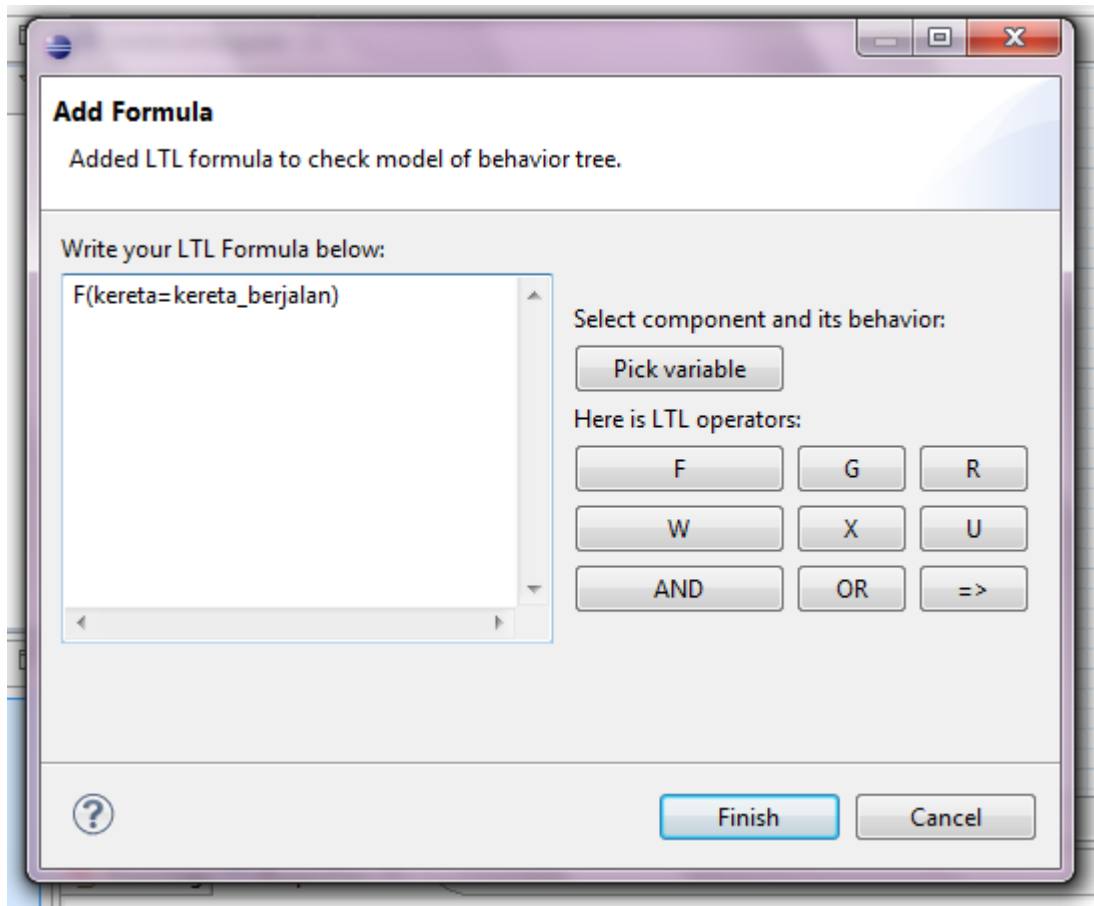


Figure 44 View of Add Formula Window

Part	Function
------	----------

Write your LTL Formula below:	
Select component and its behavior:	
Here is LTL operators:	
<input type="button" value="Finish"/>	
<input type="button" value="Cancel"/>	

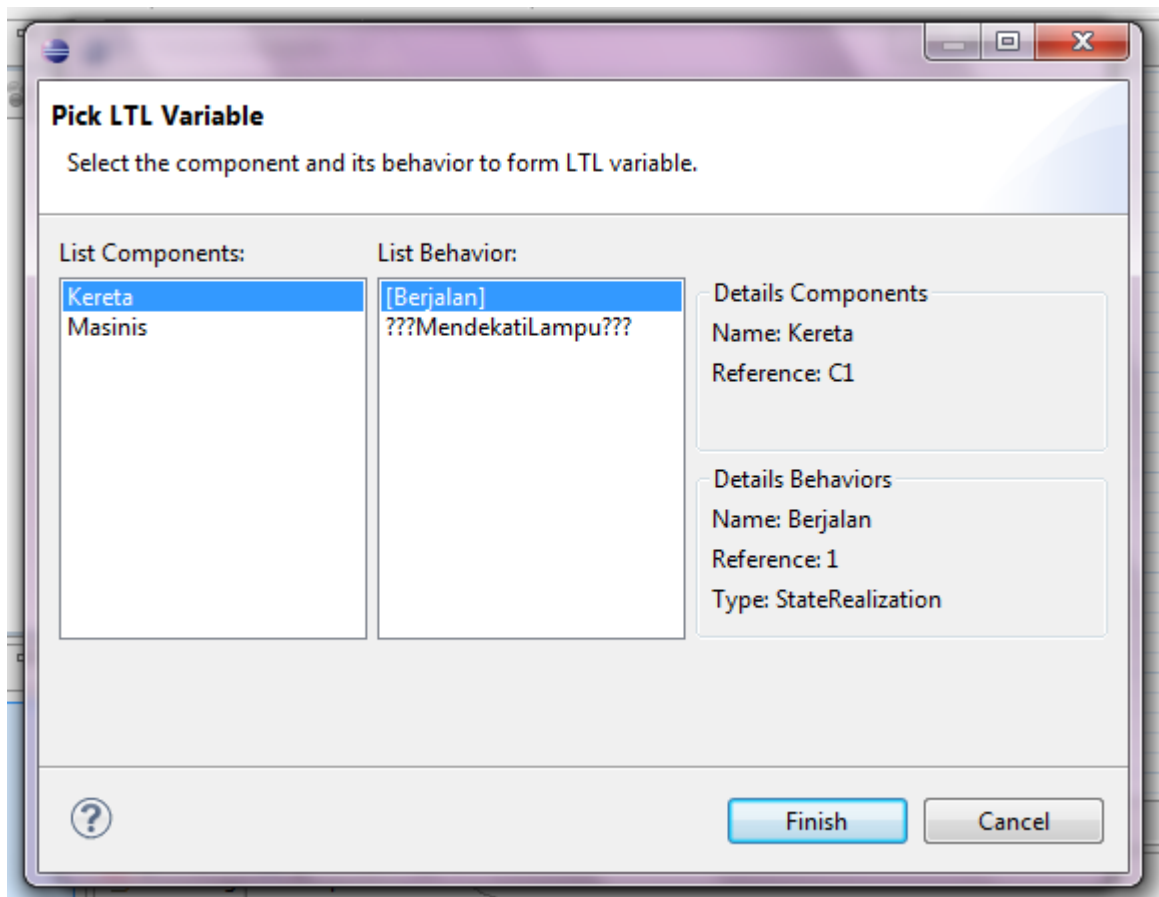


Figure 45 View of Pick LTL Variable Window

Part	Function
List Components:	
List Behavior:	
Details Components	
Details Behaviors	
<input type="button" value="Finish"/>	
<input type="button" value="Cancel"/>	

Debugging Tools

User can also debug an already created BT diagram by clicking Debug BT Diagram button in the toolbar.

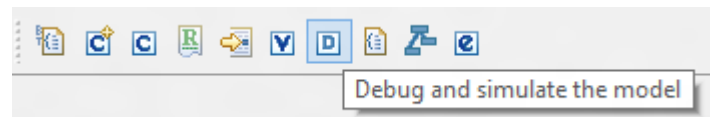


Figure 46 Debug and simulate BT icon in the toolbar

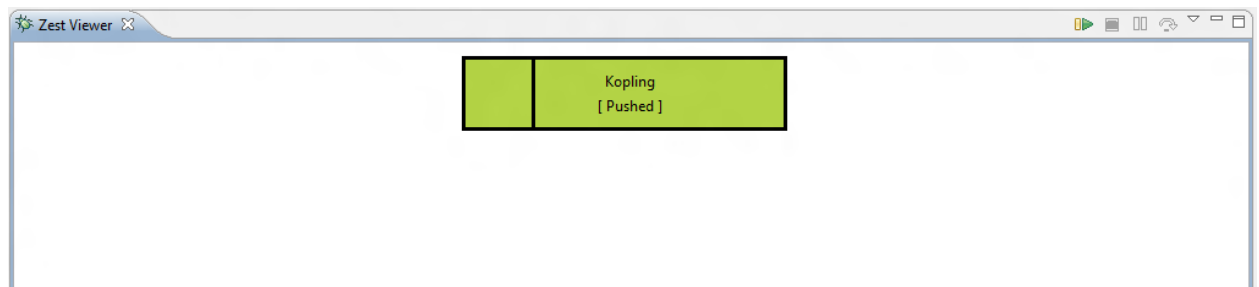


Figure 47 View of debugger and simulation tool

[illegible]

Figure 48 state viewer for Debugger

IV. Conclusion and Recommendation

4.1 Conclusion

In this project we have developed an integrated development tool for Behavior Tree called GraphBT. One of the most important features provided in the tool is the diagram and textual editor for creating Behavior Tree. The tool is delivered as Eclipse plug-in and can be directly downloaded from the Update Site.

4.2 Recommendation

During the implementation phase, there are several known bugs that is still left unresolved and some small details that is still not yet implemented, such as zoom feature using mouse scroll. Several explanation for this is because the lack of coordination and teamwork in the team. In the future, we suggest that building a good team chemistry and discipline should be prioritized to avoid these problems to be occurred again.

V. Application Features

5.1 Conclusion

In this project we have developed an integrated development tool for Behavior Tree called GraphBT. One of the most important features provided in the tool is the diagram and textual editor for creating Behavior Tree. The tool is delivered as Eclipse plug-in and can be directly downloaded from the Update Site.

5.2 Recommendation

During the implementation phase, there are several known bugs that is still left unresolved and some small details that is still not yet implemented, such as zoom feature using mouse scroll. Several explanation for this is because the lack of coordination and teamwork in the team. In the future, we suggest that building a good team chemistry and discipline should be prioritized to avoid these problems to be occurred again.

VI. Miscellaneous Implementation

Get XML representation of .BT file

To get the xml representation of BT file, we use the ATL plugin to parse the given BT file (its semantic already defined by TextBE plugin).

```
1085- /**
1086-  * getXMLFromBT is used to get the file representation of XML
1087-  * representation of a BT file. The implementation used the existing
1088-  * model that already defined in TextBE plugin.
1089-  * @param file instance of BT file
1090-  * @return File instance of the XML file, null is no XML file produced
1091-  */
1092- public static File getXMLFromBT(IFile file){
1093-     IInjector injector = null;
1094-     IExtractor extractor = null;
1095-     IReferenceModel inMetamodel;
1096-     IReferenceModel outMetamodel;
1097-     File target = null;
1098-     ModelFactory factory = null;
1099-     try {
1100-         injector = CoreService.getInjector("EMF"); //$NON-NLS-1$
1101-         extractor = CoreService.getExtractor("EMF"); //$NON-NLS-1$
1102-
1103-         factory = CoreService.createModelFactory("EMF");
1104-     } catch (ATLCoreException e) {
1105-         e.printStackTrace();
1106-     }
1107-     //IPath path = (IPath) f.getLocation();
1108-     IModel outputModel = null;
1109-     // Defaults
1110-     try {
```

Figure 49 Screenshot of Generate XML representation code

GraphBT Tree Layout

The bt diagram is using a tree layout to arrange the nodes. Each block node is drawn from the middle of the subtree's width and level's height. The computing process is done by method getWidth and getHeight in behaviortree.GraphBTUtil.java.

Get BEModel

BEModel is stored in a persistent representation so it should implements an EMF object. The BEModel is located in Diagram resource. By getting the BEModel, we will have access to the drawn diagram, such as components, requirements, behavior, etc. The implementation is located in method getBEModel in behaviortree.GraphBTUtil.java.

VII. Sample Project

7.1 Conclusion

In this project we have developed an integrated development tool for Behavior Tree called GraphBT. One of the most important features provided in the tool is the diagram and textual editor for creating Behavior Tree. The tool is delivered as Eclipse plug-in and can be directly downloaded from the Update Site.

7.2 Recommendation

During the implementation phase, there are several known bugs that is still left unresolved and some small details that is still not yet implemented, such as zoom feature using mouse scroll. Several explanation for this is because the lack of coordination and teamwork in the team. In the future, we suggest that building a good team chemistry and discipline should be prioritized to avoid these problems to be occurred again.

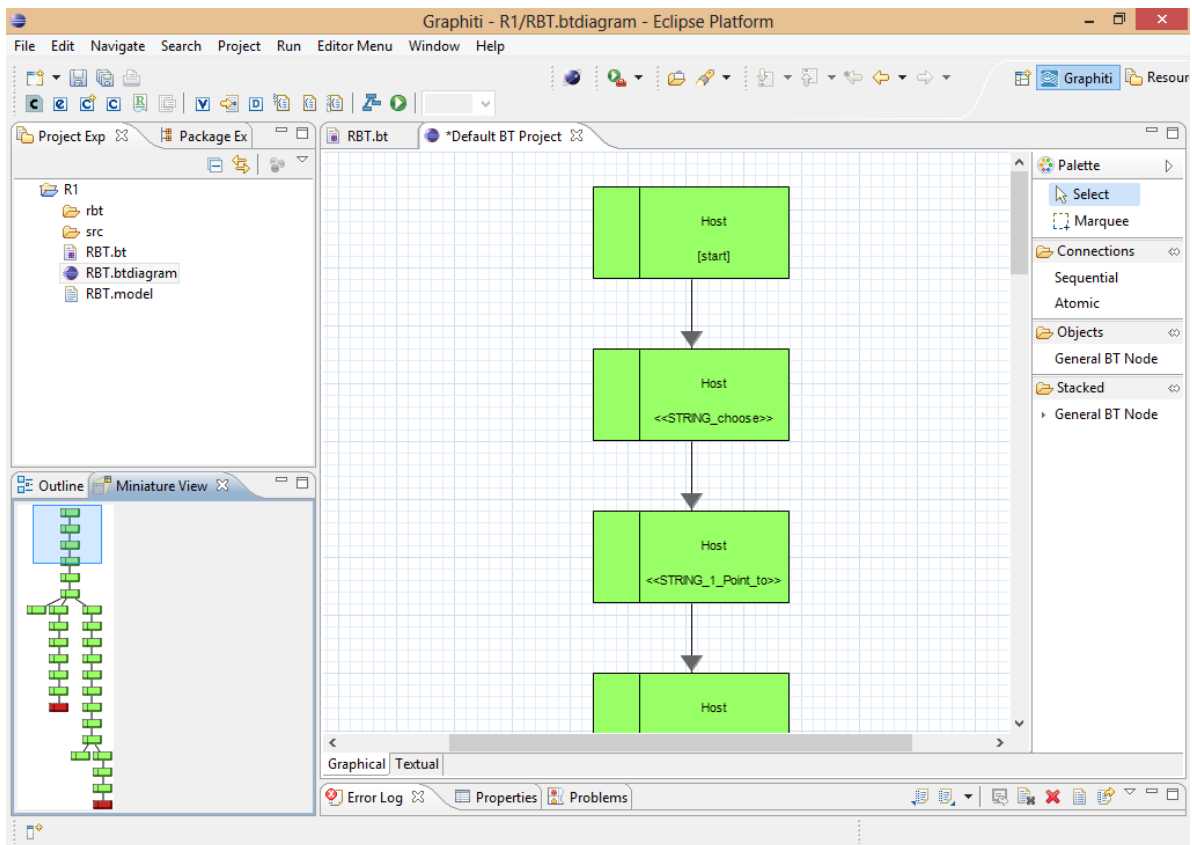
Appendix

Appendix 1. TextBE Meta Model

Metamodel is used to describe semantic of an object. TextBE metamodel uses EMF framework.

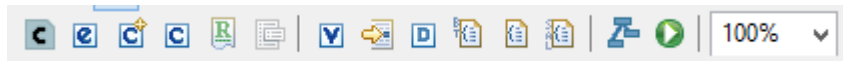
The metamodel can be accessed [here](#).

Appendix 2. GraphBT Look



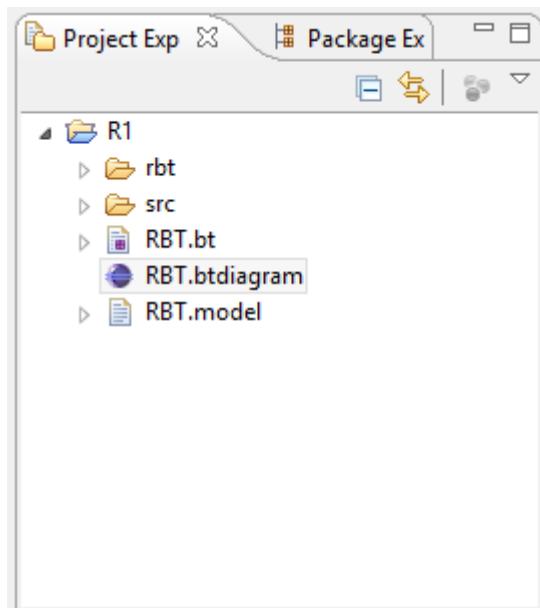
GraphBT Parts

1. Toolbar Icons



Toolbar Icon	Function
	Clear the current diagram
	Show "Manage components" wizard
	Display debug
	Extract diagram from a *.bt file
	Generate .bt file based on the diagram representation
	Generate Java source code based on the diagram representation
	Generate .sal file based on the diagram representation
	Apply layout to the tree in the diagram
	Show "Manage library" wizard
	Show "Create new component" wizard
	Show "Manage requirements" wizard
	Run generated java code
	Validate the model
	Percentage of zoom level, 100% is the default zoom level

2. Project Explorer

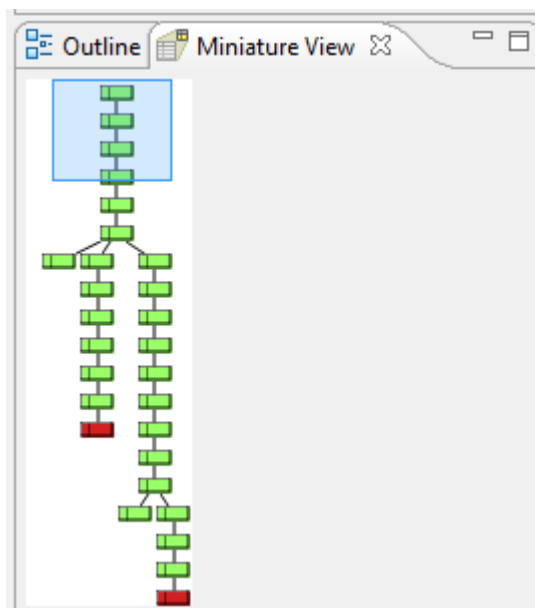


Project explorer part shows the list of active projects in Eclipse workspace. GraphBT project structure is shown in the side figure.

Notes:

Element Name	Definition
R1	GraphBT project name
rbt	Source of .bt file for reference only
src	Directory to store java source code
RBT.bt	TextBE .bt representation of the diagram
RBT.diagram	The diagram
RBT.model	The diagram model which is used as persistent objek of the business model

3. Miniature View



Miniature view part displays the whole diagram in its miniature form. The current displayed diagram is show in the blue rectangle.

4. Diagram Editor

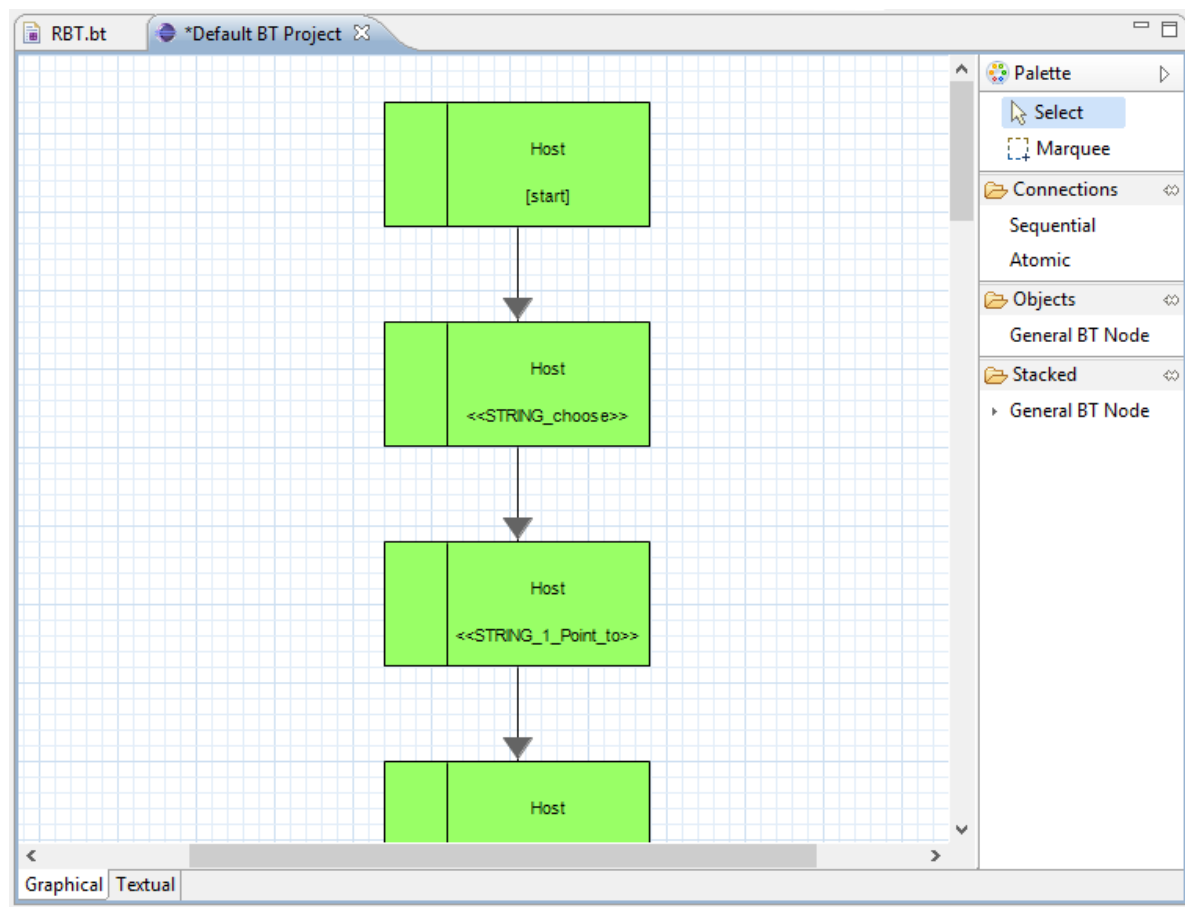
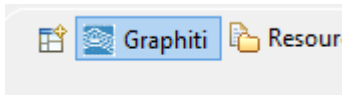



Diagram Editor is used to manipulate the BT diagram. Diagram Editor consists of many accessible functions.

Part	Definition
*Default BT Project	Project name
Select	Change the cursor in select mode, so it can pick an element in the editor
Marquee	Select multiple elements in the editor
Sequential	Add "Sequential" connection
Atomic	Add "Atomic" connection
General BT Node	Add "General BT Node" pictogram element
Stacked	Display the last function used [Deprecated]
General BT Node	
Graphical	Change mode to graphical editor
Textual	Display the BT text representation

5. Perspective



Perspective gives different view of workbench window. It arranges the required “View” and location as defined in each Perspective. To change the current perspective, click the  button. Choose Graphiti perspective if it is not already chosen.

Appendix 3. GraphBT Meta model

GraphBT used different meta model from the TextBE. It because the model can not be modified freely. The meta model used is similar but with different way of access. With the customized meta model, developer is easy to exploit things so the experience in using graphical editor richer. The metamodel can be accessed [here](#). The generated code documentation can be accessed in <http://mrzon.github.com/GraphBT/doc/> in package `behaviortree` and `behaviortree.impl`.

Appendix 4. Guide for developer

GraphBT tool is built as an open source project. Everybody is free to contribute for the plugin. The project is maintained using [GitHub](#). Before start contributing, the prerequisite to do can be accessed [here](#). The [documentation](#) of the tool is still on progress and will be evolved as long as the code is improved. Also the wiki [page](#) is containing important information to gaining more knowledge of the tool.