UTILISING BTTRACE VISUALISER AND LTL FORMULAE PATTERNS FOR ANALYSING COUNTEREXAMPLE

Irene Ully Havsa

Formal Method in Software Engineering Laboratory, Faculty of Computer Science, Universitas Indonesia, Kampus Baru UI, Depok, 16424, Indonesia

Email: irene.ully@gmail.com

Abstract

The aim of this paper is to demonstrate the utilisation of a Behavior Tree trace visualiser called BTTrace and generalised LTL formulae patterns to help system analysts analyse counterexamples and generate valuable ones. Counterexample generated by SAL model checker from a Behavior Tree model and an LTL formulae is translated into a BTTrace file. This file is rendered by BTTrace to visualise the counterexample on Behavior Tree diagram in animated fashion. Generalised LTL formulae patterns are exploited using a particular technique to assist analyst on constructing new yet meaningful property formulas. These formulas are used to obtain different and valuable counterexamples for further analysis. It is shown that BTTrace and LTL formulae patterns give significant support for analysing counterexamples of Behavior Tree model.

Keywords: *Behavior Tree, LTL, counterexample, visualiser*

Abstrak

Tujuan dari makalah ini adalah untuk menunjukkan pemanfaatan dari visualisasi jejak *Behavior Tree* yang disebut BTTrace dan generalisasi pola formula LTL untuk membantu analis sistem menganalisis *counterexample* dan menghasilkan *counterexample* yang berharga. *Counterexample* dihasilkan oleh SAL *model checker* dari model *Behavior Tree* dan formula LTL diterjemahkan ke dalam sebuah file BTTrace. File ini kemudian di-*render* oleh BTTrace untuk memvisualisasikan *counterexample* diagram *Behavior Tree* dalam mode animasi. Pola formula LTL yang sudah digeneralisasi kemudian dieksploitasi dengan menggunakan teknik tertentu untuk membantu analis untuk membangun formula properti baru namun bermakna. Formula ini digunakan untuk mendapatkan *counterexample* yang berbeda dan berharga untuk analisa lebih lanjut. Pada makalah ini ditunjukkan bahwa BTTrace dan pola formula LTL memberikan dukungan yang signifikan untuk menganalisis *counterexample* dari model *Behavior Tree*.

Kata kunci: *Behavior Tree, LTL, counterexample, visualiser*

1. Introduction

This paper discuss a tool support and formulae patterns to assist system analyst on identifying system safety requirements, specifically on analysing counterexamples. The approach uses Behavior Tree (BT) notation [1] to model system requirements and SAL model checker[1] to process the verification.

Behavior Tree is a formal modelling language that has the strengh among other language on its graphical notation which has been shown to be easy to understand by people who are not formal method experts [2]. Furthermore, BT notation has the ability to capture functions, object states, and multi-threaded behavior in a single modelling language [3].

A BT model is constructed from system requirement description, usually from functional requirement. For verification purpose, this BT model will be translated into SAL model using the existing BT to SAL translator. The safety requirement/property will be delivered in a form of LTL formulae. To learn more about how to build BT model from system requirement and perform verification afterwards, please refer to [3,4,5,6].

LTL (Linear-Time Temporal Logic) provides temporal operator to express assertion about paths through the SAL model. G(P) means a proposition

[1] http://sal.csl.sri.com/

P holds globally (holds at each state), F(P) means that P holds in the future (eventually will be held), X(P) means P holds in the next step on the path, and P U Q means that P always holds until Q holds and Q does eventually hold. Formulae can be built using standard propositional connectives, for example: AND, OR, NOT, implies. For our experiment we always use implication connective since our focus is on modifying the antecedent to restrict the possible paths as explained later. In relation with Behavior Tree, atomic formulae correspond to statements about what state a component is in, the current value of an attribute, or whether a particular message is available or not.

Given a SAL model and an LTL formulae, SAL either returns proved (means the property holds on all paths), times out (runs out of computing resource), or returns a counterexample. A counterexample is a sequence of executed actions and resulting states which show the path where the property does not hold.

SAL only returns a single counterexample at a time and will returns the same counterexample in the next runs eventhough there is another counterexample(s). One counterexample is usually not enough to conclude a system behavior that lead to property violation. To find more counterexamples, the LTL formula should be modified to eliminate from consideration the particular condition that gave raise to the recent counterexample. A technique for this modification will be described later.

A plain counterexample is not enough for analysing error, we need to trace it back to a sequence of steps on the BT model. This sequence of steps illustrating a system behavior which violates the property. Eventhough in [7] this work is claimed as a simple matter, from our experience on working on a similar project, it is shown as a time-consuming activity, since the analyst need to find a corresponding BT node for each executed action in a counterexample. The problem is increased when dealing with several counterexamples or long counterexample(s). To overcome this problem, a trace visualiser named Behavior Tree Trace (BTTrace) [8] which implemented in TextBE (Textual Editor For Behavior Engineering) [9] is introduced.

TextBE is a textual editor aiming to support the construction of BT model. It is distributed as Eclipse plugin[2]. Textual representation of a BT model is stored in a single file with extension .bt, which will be rendered by TextBE into a static diagram. To enable visualisation in animated fashion on this diagram, we use an extension named BTTrace.

As a visualiser, BTTrace takes a file with extension .btt defining visualisation sequence. This file contains the execution order of nodes in the diagram using format as depicted in Figure 1.

BT filename.bt
TRACE [node1][node2] ….
LOOP [nodeA][nodeB] ….

Fig. 1. The format of BTTrace file.

The first line define a corresponding BT textual representation as a ”base” for visualisation. The second line define a sequence of nodes that will be visualised once. In BTTrace, each node in a BT diagram has an identity number. This numbering begin at 1, starts from the root node and continues through the diagram in preorder traversal manner. Therefore, [1] points to the first (root) node. The node number can also be in a form [a,b]. For example, [2,3] points to second node and gives a shadow to third node. We will discuss about the function of this format later. The third line (optional) define a sequence of nodes that will be visualised repeteadly to represent an infinite loop. Once a BTTrace file is loaded in Eclipse text editor, the visualisation will be executed automatically.

BTTrace shows the visualisation by high-lighting one node at a time, means that this node is executed at this step. The visualisation example is depicted in Figure 2. At one time step, the visualisation shows a scene as in Figure 2(a), then in the next time step it will change to Figure 2(b), Figure 2(c), and so on. In the BTTrace, this sequence will be represented as …[Btnode$_1$][Btnode$_2$][Btnode$_3$]… and so on. Similar animation will be applied on other BT diagrams.

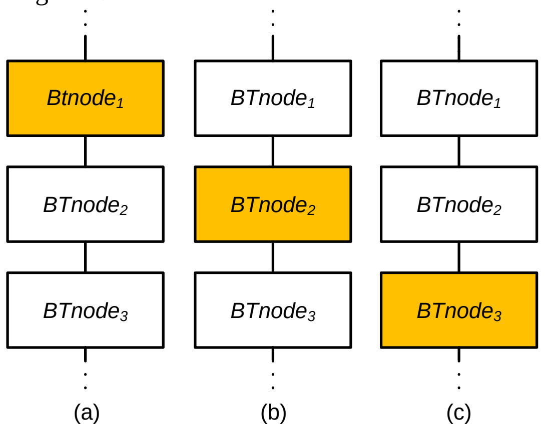

Fig. 2. BTTrace visualization example. (a), (b), and (c) are shown simultaneously.

[2] http://code.google.com/p/textbe/wiki/InstallingTextBE

In Behavior Tree, there are nodes which has a role as reference to another node, they are reversion, macro, and branch-kill node. These nodes are called reference node and the destination node is called target node. For marking a target node through the visualisation, BTTrace uses dim-light color as shown in Figure 3. Both of the reference node and the target node is lighted at the same time. We represent this visualisation in BTTrace file as [referenceNode,targetNode]. Branch-kill node is visualised in similar fashion with Figure 3(b).

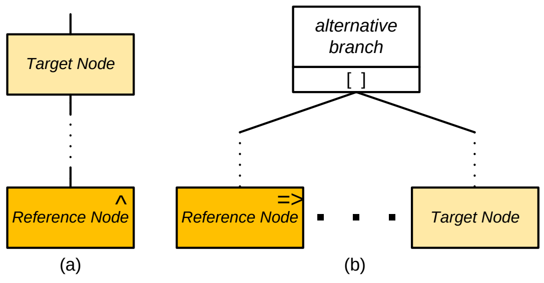

Fig. 3. High-lighting and dim-lighting when reaching (a) reversion node and (b) macro node.

To visualise a counterexample, we need to create a BTTrace file representing the content of the counterexample. The technique to create this file automatically is described in the next section.

2. Counterexample Visualisation

A counterexample describes a system behavior as a sequence of steps. Each step contains the name of executed action followed by the value of variables after the execution. For example, Figure 4 shows a snippet of counterexample showing an execution of action which in SAL model has a label A25.

The implementation of translator from BT to SAL model that we use merge the first node (or atomic node) into initialisation step, then each other node is translated into SAL action one by one. The translated action has a label with format Ai, where i is a number given on each translation. For example, second node will be translated into action A1. The traversal process is in depth-first preorder manner, similar to the node numbering sequence in BTTrace. The difference is only in the first node, which does not counted in BT to SAL translation. Therefore, we only need to capture the action number and increment it by 1 to get the node number in BTTrace and then arrange it in BTTrace file.

```
….
Counterexample:
========================
Path
========================
….
Transition Information:
(module instance at [Context: model2, line(617),
column(13)]
  (label A25
    transition at [Context: model2, line(247),
column(4)]))
-----------------------
Step 6:
--- Input Variables (assignments) ---
extInMsg_dL_goes_down = true
….
```

Fig. 4. A snippet of a SAL counterexample.

Counterexample file only is not sufficient for generating visualisation file. For reference node (reversion, macro, and branch-kill), action label can only shows the reference node without telling the target node. To complete the information, we need to supply a file containing pairs of reference and target node number, which we called reference file. A snippet of an example of this file is as below:

```
….
20 16
25 16
32 26
….
```

Fig. 5. A snippet of a reference file.

The pair 20 16 means that the reference is a node which translated into action A20 and the target is node with action label A16. This file should contains all of the pairs in a BT model. We can omit this information by simply provide empty file, but the visualisation will not be smooth and confusing as the analyst might not be prepared for the movement of visualisation from one part to another part of the BT diagram.

Once the counterexample file and reference file is available, we can create a BTTrace file automatically. For generating BTTrace file, in Figure 6 we provide the algorithm.

```
store reference information from reference file into a set
start trace
FOR each line in counterexampleFile
  IF found keyword “label A”
    take the action number
    IF it is a reference node
        write the pair of reference node number and target node number
    ELSE
        write the node number
  ELSE IF found keyword "Begin of Cycle"
      start looping trace
```

Fig. 6. Algorithm for generating BTTrace file from SAL counterexample and reference file.

SAL counterexample always has a keyword “Counterexample:”, therefore in our implementation, a BTTrace file will be generated only if this word appear in the counterexample file. Otherwise, a BTTrace file is generated, but it is not a valid one since it only contains an error message. There are two common mistake that lead to generation of this invalid file, which is a proven message file or a SAL error message is mistaken as counterexample file.

An example of generated BTTrace file is depicted in Figure 7.

```
BT model2.bt
TRACE [1][2][3][17][18][19][26,17][18]
LOOP [19][26,17][18]
```

Fig. 7. An example of valid BTTrace file.

The “base” BT model is defined in a file named model2.bt. The visualisation starts by high-lighting node number 1 (root node), number 2, number 3, and so on. When it reach [26,17], node 26 will be high-lighted and node 17 will be dim-lighted. The trace ended when it reach node 18, but then followed by sequence of nodes in LOOP section which will be visualised repeatedly until the analyst stop the visualisation.

For a huge and complex BT model, analyst is likely to obtain long counterexamples. Interesting part of these counterexamples id usually only appears as a short subsequence in the middle. It would be more convenient for analyst to examine this part only rather than exploring the entire counterexample over and over again. BTTrace support this issue as it provide a flexibility to manipulate BTTrace file. At first, analyst watch the entire counterexample to determine which part is interesting. Then the analyst pick the corresponding subsequence from the BTTrace file, and remove the other subsequences. A new trace will be create, which should showing the interesting part of the counterexample, then the analyst can focus on this trace.

3. Generating Different Counterexamples

System verification process that use model checking always exploit counterexample. However, in the publication the generation of counterexample usually put in background and not explored. In this paper, we discuss in detail about this aspect in a form of general technique to ease the effort.

After obtaining a counterexample, we often need to obtain other counterexamples to support our analysis. To generate different counterexample, we need to modify the current LTL formulae to eliminate a particular case that invoke the counterexample. The process is illustrated in Figure 8. This figure is a modification and more detail version of a diagram from the presentation of [6].

There are various types of case that we can eliminate depend on the model and the system as a whole. We have found several types that will generally appear in most BT model.

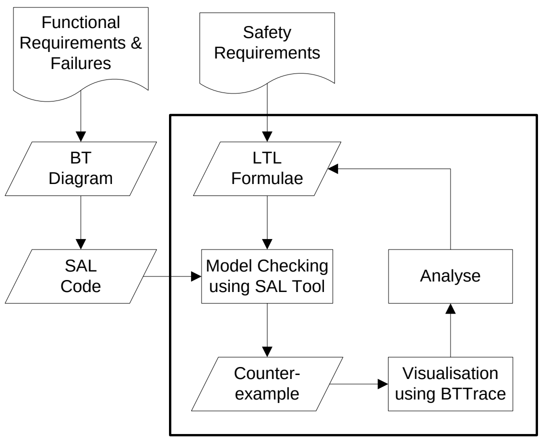
Functional Requirements & Failures
Safety Requirements
BT Diagram
LTL Formulae
SAL Code
Model Checking using SAL Tool
Analyse
Counter-example
Visualisation using BTTrace

Fig. 8. Experiment flow diagram. The core experiment is inside the box.

External input and event in BT notation described the environment and operator behavior which a system does not has control over it. System designer and analyst naturally expect the external to behave as it should be, but in many case counterexamples show the opposite. The first case is an external input/event *E* should not occur when a component *C* is in state *s*. Then if the opposite condition appears in a counterexample and we want to see what will happen if we eliminate this possibility, the analyst can use this pattern:

G(NOT(C=s AND E=true)) => initial formulae

To prevent more than one external input/event, i.e. E_1 and E_2, to occur, we can use this pattern:

G(NOT(C=s AND (E1=true OR E2=true))) => initial formulae

On the opposite, pattern below is suitable if we want to explore a path that event *E* is always available whenever component *C* is in state *s*, and the event will be executed whenever possible.

G((C=s) => (E=true)) => initial formulae

The analyst can use a combination of patterns above to control the environment or user behavior and creating a "perfect" condition. For each external input/event controlled, we can derived several conclusion and deliverables. An external input represents operator behavior could be a starting point to develop a standard operator procedure which all operators should follow. A path leading to execution of external input representing unexpected user action should be addressed by introducing an extension to handle this behavior. Furthermore, if the external input or event represents a component failure, which cannot be predicted at all, we should addressed it by enhance the design with a back up system to substitute the component.

Environment and user behavior are the most important thing to concern about. Another thing that can be explored is the different case on system. For a system that has several modes, options, or choices it is a good idea to explore each possibilities. The result can then be analysed to find a more specific case that lead to property violation, or construct general case from all possibilities.

There are two types of system mode determination, the first one is determined once in initialisation phase and will remain the same through entire execution, the second one is determined in initialisation phase and can be changed in the middle of execution. To explore each case, we need to generate counterexample for one case at a time.

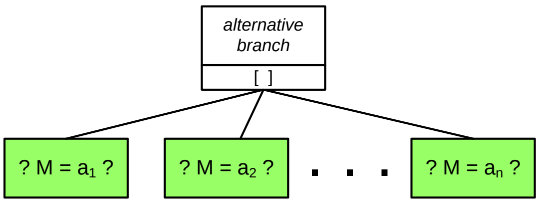

Fig. 9. Alternative branch with selection node as 'guard' for each branch.

Cases in BT notation is illustrated as *alternative branch*, which each child branch is guarded by *selection node* as depicted in Figure 9. Therefore to chose particular branch, we "force" the model checker to traverse the selected case. For a system that choice is only made in initialisation, we can use this pattern:

$(M=a_i)$ => initial formulae

If the choice eventually changes but analyst want to explore one particular case only, we make sure that the choice will be the same on the entire execution by this pattern:

$G(M=a_i)$ => initial formulae

If the analyst want to explore the behavior of particular sequence of modes change, i.e. mode a_1 then a_2, the analyst use this pattern:

$(M=a_1)$ AND $F(M=a_2)$ => initial formulae

Sometimes analyst need to explore what if particular condition is remain the same until a mode is chosen. To express that, use this pattern:

$U(C=s, M=a_i)$ => initial formulae

The analyst can combine these pattern to check specific case, until the most specific one. But to keep in mind, analyst should check that a particular case is really reachable by model checking an LTL formula with this pattern:

$G(NOT(M=a_i))$

This formula means that the mode a_i will never be chosen in any path. Different with checking a property, this time we expect to get counterexample. A counterexample means there is at least one path which a_1 is chosen. On the other hand, if SAL returens proven then this formula is satisfied, which tells that mode a_i is not possible to occur.

4. Results and Discussion

In our experience, analysis effort is decreased significantly after utilising BTTrace. The most important part is analyst can "watch" visualisation of counterexample trace by just doing several simple steps. The visualisation can be repeated several times, which helps analyst to learn the behavior faster.

We find some weaknesses in BTTrace. For large diagrams, we need to use large screen to recognize each node easily. In current implementation, BTTrace also does not give mark on nodes that have been traversed. For a BT model that has several pararel branch, it is hard to remember how far is the progress of each branch which makes analyst easy to lose track. We plan to add this feature on the next development.

The patterns for generating more counterexample are generalised version of a real LTL formulas that were used in a research on Aerial Fire-fighting Management System case study [7]. The technique is proved to be effective on finding various interesting counterexamples for safety property evaluation analysis. However, this technique still need to be evaluated on several other case studies.

5. Conclusion

The utilisation of BTTrace as counterexample visualiser and generalised LTL formulae patterns significantly increase the eficiency of counterexamples analysis. BTTrace brings a huge support on examining each counterexample with its graphical and animated fashion, and also its flexibility that allow analyst to examine only a small part of a counterexample. On the other hand, LTL formulae patterns assist analyst on constructing new property formulae for generating different valuable counterexample.

Acknowledgement

This research is part of a research internship held at School of ITEE The University of Queensland on February - April 2012, supervised by Prof. Peter Lindsay and Mr. Sentot Kromodimoeljo. This internship is funded by a grant for a research titled *Assessment of the Applicability of Behavior Tree Notation for Large System* with PIC Prof. Belawati H. Widjaja.

Reference

[1] R.G. Dromey, "From Requirements to Design: Formalizing the Key Steps" *In Proceeding of 1st IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pp. 2-13, 2003.

[2] D. Powell, "Behaviour Engineering - A Scalable Modelling and Analysis Method" *In Proceeding of IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pp. 31-40, 2010.

[3] P.A. Lindsay, "Behaviour Trees: From Systems Engineering To Software Engineering" *In Proceeding of IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pp. 21–30, 2010.

[4] T. Myers, R.G. Dromey, "From Requirements to Embedded Software - Formalising the Key Steps" *in Proceedings of the 2009 Australian Software Engineering Conference (ASWEC'09)*, pp 23-33, 2009.

[5] P. Lindsay, K. Winter and N. Yatapanage, "Safety Assessment Using Behavior Trees and Model Checking" *in 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2010)*, 2010.

[6] L. Grunske, P.A. Lindsay, N. Yatapanage, K. Winter, "An Automated Failure Mode And Effect Analysis Based On High-Level Design Specification With Behaviour Trees" *In Proceeding of International Conference on Integrated Formal Methods (IFM)*, LNCS vol. 3771, pp. 129–149, 2005.

[7] P. A. Lindsay, K. Winter, S. Kromodimoeljo, "Model-based Safety Risk Assesment using Behaviour Trees" *In The System Engineering, Test and Evaluation Conference*, 2012.

[8] F. Dolot, "Design And Implementation Simulation Language For Requirement Engineering In Form Of Behavior Tree Animation in TextBE (Textual Editor For Behavior Engineering), " B.S Thesis, Faculty of Computer Sciences, Universitas Indonesia, Indonesia, 2011.

[9] T. J. Myers, "TextBE : A Textual Editor for Behavior Engineering" *In Improving Systems and Software Engineering Conference*, 2011.