Formal Verification of an Autonomous Wheel Loader by Model Checking

Rong Gu, Raluca Marinescu, Cristina Seceleanu, Kristina Lundqvist

6TH INTERNATIONAL CONFERENCE ON FORMAL METHODS IN SOFTWARE ENGINEERING
FormaliSE June, 2018
Use case: Autonomous Wheel Loader

Autonomous Wheel Loader (AWL)

(a) A heavy construction vehicle
(b) Transports material, loads and unloads at crushers
(c) No human operator on-board
(d) Works under any condition, e.g., dusty, raining, foggy, and dark environment
(e) Existing prototype has no intelligence (e.g. collision avoidance) and no dependability guarantees
(f) Path planning and replanning for autonomous path following and collision avoidance

Requirement

a) An AWL must calculate the initial path before it starts to move and avoid all kinds of obstacles dynamically as it moves.
b) Follow the planned path autonomously.
c) React to errors in the control system timely and correctly.
Use case: Autonomous Wheel Loader

• The architecture of the AWL’s control system

• Task allocation in the control system
Method: Formal Modelling and Verification

- **UPPAAL TA**: UPPAAL Timed Automata
- **TCTL**: Timed Computation Tree Logic
Preliminaries: Path-planning algorithm – A* algorithm

- A widely used algorithm for path finding and graph traversal.
- A* algorithm works in a grid.
- 2-dimensional array (int map[N][N]), 1: walkable, 0: ”not walkable”.
- A* algorithm is an extension of Dijkstra’s algorithm, finding the shortest path from A to B.

![Diagram of a grid with A* algorithm in action](image)

Figure 2. A* algorithm works in grid.
Preliminaries: Path-planning algorithm – A* algorithm

- \( F = G + H, \)
  - \( G: \) cost from start to current cell
  - \( H: \) estimated cost from current cell to destination
- Manhattan Distance: The simple sum of the horizontal and vertical distance ignoring the “unreachable” cells.

Figure 3. A* algorithm finds the shortest path.
Preliminaries: Collision-avoidance algorithm

Dipole Flow Field: Static Flow Field – avoid static obstacles

\[ F_a = \frac{k_a q_0 Q}{D^2} \quad F_r = \frac{k_r q_0 q_1}{d^2} \]
\[ F_{\text{flow}} = F_a + F_r \]

Figure 6. The representation of the static flow field (unity vectors), (A) the initial path with the configured static attractive field, (B) the static flow field with added repulsive force to the obstacles\(^1\).

**Preliminaries: Collision-avoidance algorithm**

**Dipole Flow Field:** Dynamic Dipole Field – avoid dynamic obstacles

- Every object is assumed to be a source of magnetic dipole field.
- The magnitude of the magnetic moment is proportional to the velocity.
- Two moving objects repulse each other when they are close enough.

\[
\vec{m} = k_m \vec{v}
\]

\[
\vec{F}_d = \frac{k_d}{d^5} \left[ (\vec{m}_0 \cdot \vec{r}) \times \vec{m}_i + (\vec{m}_i \cdot \vec{r}) \times \vec{m}_0 + (\vec{m}_0 \cdot \vec{m}_i) \times \vec{r} \right] - \frac{5 \cdot (\vec{m}_0 \cdot \vec{r}) \cdot (\vec{m}_i \cdot \vec{r})}{d^2} \times \vec{r},
\]

Preliminaries: Timed automata and UPPAAL

- Timed automata (TA): finite state machines with real-valued clocks
- UPPAAL: A TA-based toolbox for validation and verification of real-time systems.

Figure 8. A lamp example of a network of UPPAAL TA
**Preliminaries: Timed Computation Tree Logic**

Formalize the natural-language requirements to (Timed) Computation Tree Logic (TCTL) queries, which are in the form:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E\Diamond p$</td>
<td>There exists a path where $p$ eventually holds</td>
</tr>
<tr>
<td>$A\Box p$</td>
<td>For all paths, $p$ always holds</td>
</tr>
<tr>
<td>$A\Diamond p$</td>
<td>For all paths, $p$ will eventually hold</td>
</tr>
<tr>
<td>$p \rightarrow_{\leq T} q$</td>
<td>For all paths, if $p$ holds then $q$ will eventually hold within $T$ time units.</td>
</tr>
</tbody>
</table>

Figure 9: Different types of TCTL queries and their expressions in UPPAAL
Abstraction of map and movement

- **Map Abstraction**

\[ f_1 : \mathbb{R}_+^2 \rightarrow \mathbb{Z}_+^2 \quad f(x, y) = (z_x, z_y) \]
\[
x - \frac{\epsilon}{2} \leq z_x \leq x + \frac{\epsilon}{2}, \text{ and } y - \frac{\epsilon}{2} \leq z_y \leq y + \frac{\epsilon}{2}\]

- **Movements Abstraction**

\[
p = (z_{x_0}, z_{y_0})(z_{x_1}, z_{y_1}) \cdots (z_{x_{n-1}}, z_{y_{n-1}})(z_{x_n}, z_{y_n}) \]
\[
\begin{cases}
z_{x_i} = z_{x_{i-1}} \pm \nu, \text{ where } x_i \geq 1 \\
z_{y_i} = z_{y_{i-1}} \pm \nu, \text{ where } y_i \geq 1
\end{cases}
\]
Formal model of tasks and algorithms

Figure: Model the tasks as TA and algorithms as C-code functions in TA.
Mapping activity diagrams to TA

(A) TA of Main Task

(B) TA of Execution Function

Timed Automata (TA) in UPPAAL
Overview of the system model

Figure 11: Task allocation in the control system

Figure 12: Task allocation for moving obstacle
Modeling the path-planning and collision-avoidance algorithms

```c
void AStart()
{
    Point ts, ps;
    int i, j;
    bool findEnd=false;
    insert(open, start);
    while ((open.listLen != 0) && (!findEnd))
    {
        // A* algorithm logic here
    }
```

C-code functions

TA
Data communication between tasks: global variables, clocks/channels:

a) Crucial signals: channels, e.g., freeze, fail, etc.

b) Asynchronous signals: clocks, e.g., \(w_{\text{task1\_trigger}} \leq w_{\text{task1\_threshold}}\).

Figure 13: a TA of a task waiting for data “wheel loader’s position” from another task
Initial Path Computation: during initialization, an AWL must compute an initial path to the destination, which ought to avoid all the static obstacles identified in the quarry.

Query

Q1.0: E<> mainTask.Wait
Q1.1: A<> mainTask.Wait imply lenOfPathStack > 0
Q1.2: E<> currentPosition == pile and destination == crusher
Q1.3: (currentPosition == pile and destination == crusher) -> currentPosition == crusher
Q1.4: E<> currentPosition == crusher and destination == pile
Q1.5: (currentPosition == crusher and destination == pile) -> currentPosition == pile
Q1.6: A[] forall(i:int[0,9]) currentPosition != staticObstacle[i]
**Obstacle Avoidance**: AWLs must avoid static and dynamic objects around them in due time before returning to the initial path.

Formalize requirement:

Q2.0: $A[\cdot] \text{currentPosition} \neq \text{currentObstacle}$

Q1.3: $(\text{currentPosition} = \text{pile} \text{ and } \text{destination} = \text{crusher}) \rightarrow \text{currentPosition} = \text{crusher}$

Q1.4: $E<> \text{currentPosition} = \text{crusher} \text{ and } \text{destination} = \text{pile}$

Q1.5: $(\text{currentPosition} = \text{crusher} \text{ and } \text{destination} = \text{pile}) \rightarrow \text{currentPosition} = \text{pile}$
Mode Switch Mode A: if the information of obstacles cannot be reported to the control unit, which is very dangerous, AWL must freeze its motion within 20 time units.

Formalize requirement

Q3.1: E<> errorStart == true

Q3.2: error_start==true -> (SYSTEM_ERROR==true and reaction_time<=20)
End-to-end Deadline: to guarantee a certain productivity, AWLs must finish one cruise within 2200 time units.

Formalize requirement:

\[ Q4.0: (\text{currentPosition}==\text{pile and destination}==\text{crusher}) \rightarrow (\text{currentPosition}==\text{pile and destination}==\text{pile and gClock} \leq 2200) \]
## Table 1: Verification queries and results

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Query</th>
<th>Result</th>
<th>States explored</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial path computation</td>
<td>Q1.0: E&lt;&gt; mainTask.Wait</td>
<td>Pass</td>
<td>2</td>
<td>110 ms</td>
</tr>
<tr>
<td></td>
<td>Q1.1: A&lt;&gt; mainTask.Wait imply lenOfPathStack &gt; 0</td>
<td>Pass</td>
<td>8780</td>
<td>484 ms</td>
</tr>
<tr>
<td></td>
<td>Q1.2: E&lt;&gt; currentPosition == pile and destination == crusher</td>
<td>Pass</td>
<td>1</td>
<td>0 ms</td>
</tr>
<tr>
<td></td>
<td>Q1.3: (currentPosition == pile and destination == crusher) -&gt; currentPosition == crusher</td>
<td>Pass</td>
<td>14191</td>
<td>1125 ms</td>
</tr>
<tr>
<td></td>
<td>Q1.4: E&lt;&gt; currentPosition == crusher and destination == pile</td>
<td>Pass</td>
<td>2339</td>
<td>297 ms</td>
</tr>
<tr>
<td></td>
<td>Q1.5: (currentPosition == crusher and destination == pile) -&gt; currentPosition == pile</td>
<td>Pass</td>
<td>14204</td>
<td>782 ms</td>
</tr>
<tr>
<td></td>
<td>Q1.6: A[] forall(int[0,9]) currentPosition != staticObstacle[i]</td>
<td>Pass</td>
<td>8780</td>
<td>485 ms</td>
</tr>
<tr>
<td>Obstacle avoidance</td>
<td>Q2.0: A[] currentPosition != currentObstacle</td>
<td>Pass</td>
<td>125941</td>
<td>6297 ms</td>
</tr>
<tr>
<td></td>
<td>Q1.3: (currentPosition == pile and destination == crusher) -&gt; currentPosition == crusher</td>
<td>Pass</td>
<td>227646</td>
<td>13969 ms</td>
</tr>
<tr>
<td></td>
<td>Q1.4: E&lt;&gt; currentPosition == crusher and destination == pile</td>
<td>Pass</td>
<td>2678</td>
<td>375 ms</td>
</tr>
<tr>
<td></td>
<td>Q1.5: (currentPosition == crusher and destination == pile) -&gt; currentPosition == pile</td>
<td>Pass</td>
<td>192406</td>
<td>10656 ms</td>
</tr>
<tr>
<td>Mode switch: error A</td>
<td>Q3.1: E&lt;&gt; errorStart == true</td>
<td>Pass</td>
<td>30</td>
<td>234 ms</td>
</tr>
<tr>
<td></td>
<td>Q3.2: error_start==true -&gt; (SYSTEM_ERROR==true and reaction_time&lt;=20)</td>
<td>Pass</td>
<td>91</td>
<td>250 ms</td>
</tr>
<tr>
<td>Mode switch: error B</td>
<td>Q3.1: E&lt;&gt; errorStart == true</td>
<td>Pass</td>
<td>29</td>
<td>234 ms</td>
</tr>
<tr>
<td></td>
<td>Q3.2: error_start==true -&gt; (SYSTEM_ERROR==true and reaction_time&lt;=15)</td>
<td>Pass</td>
<td>320</td>
<td>266 ms</td>
</tr>
<tr>
<td>End-to-end deadline</td>
<td>Q4.0: (currentPosition==pile and destination==crusher) -&gt; (currentPosition==pile and destination==pile and gClock &lt;= 2200)</td>
<td>Pass</td>
<td>590326</td>
<td>36641 ms</td>
</tr>
</tbody>
</table>

**PASS** 36641 ms
What else did we observe?

Figure 14: A livelock scenario
Conclusion

- We have created a formal model of an industrial prototype of an AWL and its working environment
  - a discrete map and a TA for a moving obstacle
  - 11 TA for algorithms and tasks in the control system
  - encoded computations in the C-code functions of TA
- We have verified the system model and the algorithms against AWL’s requirements
  - functional requirements
  - timing requirements
- Counter-examples found by exhaustive verification are helpful for future optimization of system design and algorithms.
Lessons learned and future work

- Lack of floating-point value support in UPPAAL
  - More accurate path-planning and collision-avoidance algorithms need real numbers
  - UPPAAL model only supports integers
- Limitations of Dipole Flow Field algorithm applied in collision avoidance
- Hierarchical verification model/method is needed for more complex system model
  - Discrete model and exhaustive verification: decision-making component
  - Continuous model and statistical verification: real-valued map and dynamics, any-angle path, etc.
Thank you for listening!

Rong Gu (rong.gu@mdh.se)
Mapping activity diagrams to TA

(A) TA of Main Task
(B) TA of Execution Function

Timed Automata (TA) in UPPAAL