Automatic Scheduling of Robotic Tasks

Ethan Bledsoe and Yingxue Wang

Supervisor: Jacek Malec

Abstract

This report presents the results of our further development and maintenance of the existing MiniZinc robotics task scheduler model created by Tommy Kvant in his thesis *Task scheduling for dual-arm industrial robots through Constraint Programming,* 2015*.* During our testing, we found issues of the old model coping with the newest version of software as the 2.1.5 version of MiniZinc cannot handle empty array properly and will crash the program. We also made changes to the model's searching method to improve its efficiency and created a thorough instruction of how users can work with the scheduler. Different solvers, including Gecode, JaCop, and G12 were tested in solving the tasks. The effect of the domain filters introduced in the old model was also tested and showed they have a similar effect on the new model. In the end of the thesis, we discussed about the potential of adapting the model to domains other than scheduling assembly tasks, and we hope this study can be applied in various fields.

Table of Contents

1. Introduction

Following a trend of automated operations in industrial production, the automation of robotic task scheduling plays a more and more important role in modern society. A lot of work has thus been done, and one of the effective approaches is to solve the scheduling problem using a constraint-programming method. In this case, the user will need to prepare the required information for fulfilling the task of an assembly problem, which is usually presented as a data file, and input it to a constraint-programming based scheduler. The optimal schedule will then be solved by a Constraint Programming Solver(CPS), if the task is solvable. However, the scheduler model has not yet been fully tested, and there is no sufficient database for the robots to be able to do a variety of movements. In this project, a more thorough testing of an existing MiniZinc model was done, and a few modifications were made to optimize the scheduling process on both the user and the program aspects. Three solvers: Gecode, JaCoP and G12/FD were used to test the modified model.

1.1 Project Goal

The goal of the project is to further develop and solidify the existing MiniZinc scheduler model, then incorporate it with CPS solvers. Maintain the Github repository to allow engineers to have easy access of programing the dual-arm industrial robots.

1.2 Related Work

The most direct contribution, the existing model and many of the testing cases were created by Tommy Kvant in his master thesis, February, 2015, under the supervision of Jacek Malec and Maj Stenmark at Lund University.[1]

1.3 Report Structure

This report first introduces how to use the scheduler to solve assembly tasks, including how to create a data file using an .xml file and a time matrix using AssemblyConv.jar. Then it talks about the issues that we have encountered in testing the model and how we solved them through making changes on AssemblyConv.jar. Afterwards, the report analyses how different solvers behave with the newest version of the softwares, MiniZinc 2.1.5, and updates how the temporal and predecessor filters work regarding to different solvers. It also discusses the generality of the model on applying to different domains.

2. Workflow

To solve the scheduling problem of a specific task, an input containing all of the relevant information, such as steps, number of machines(robotic arms) available, the name of the components, required time, is required by the scheduler. The input can be presented in the form of a data file. The example of a data file can be found in the git repository of the project. As a data file has the format that is not really readable and programmable for a non-trained user who does not know about the programming language, Kvant developed an executable .jar file called AssemblyConv to automatically generate the data file. The .jar file takes in a time matrix(.csv) and a list of objects written in an .xml file, to convert them into a data file(.dzn) that suits the MiniZinc model(.mzn). We have created an annotated .xml file, which can act as a template, attached in the Appendix 7.1 to explain how to construct an .xml file.

One of the solvers can be chosen to solve for results that satisfy all of the constraints listed in the model, based on which problem types they are good at solving. If the problem is solvable, satisfied results will be printed until the solver finds the optimal one. The output was designed to include a numbered list of actions that the robot will perform, the start and end time, predecessors of the numbered tasks, etc. The detailed Output explanation is given at section 5.3.1.

3. Issues and Improvements on Existing Model Working with Various Test Cases

The model that we based our work off of can be found at *https://git.cs.lth.se/jacek/constraintsandrobots* within the Model6 folder that is located within the MiniZinc folder in the main directory. The file here is called model.mzn. This was the latest version of the model that was pushed by Tommy Kvant.

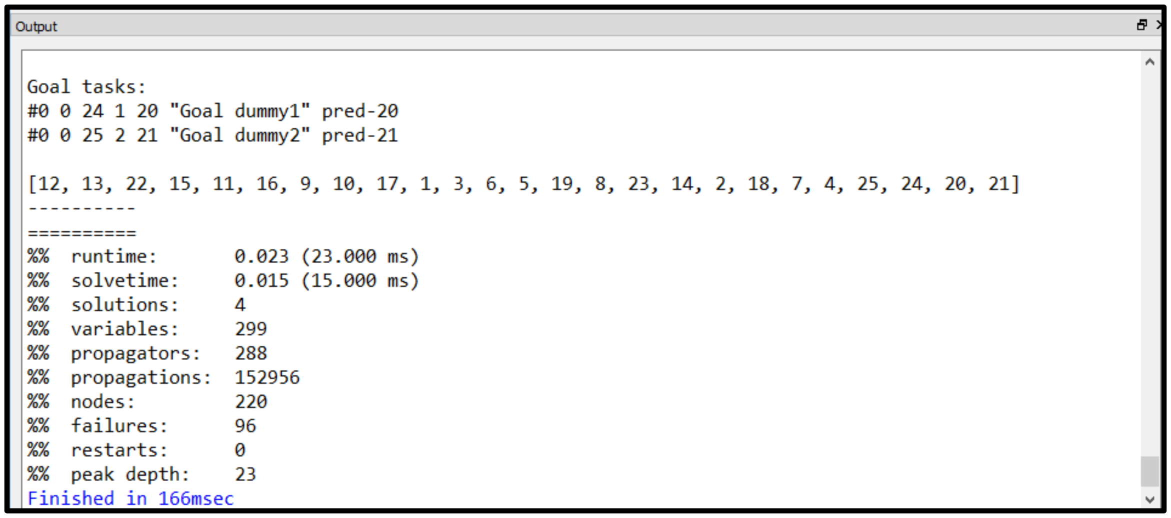
Likewise, the datafile we used for testing was also located inside of the Model6 subdirectory named XML. The file here is called xml_in_minizinc.dzn. This data file describes the case of a two armed robot constructing an emergency stop button, consisting of 5 primary components.

3.1 Model Changes

We ended up leaving many of the constraints the same as they were in the original model from Tommy Kvant, where we made changes with significant results is in the solution statement located on lines 653 through 659.

Since the solve statement of the model is one of the most vital portions we decided it would be a good place to look at for possible improvements. To do this we experimented with the arguments passed to int_search(), specifically the arguments that were previously all first_fail and indomain_median. After experimenting with different values for these arguments we discovered the model that consistently provided better results was one that included indomain_max on line 653 and indomain_min on line 654. The rest of the lines remained the same.

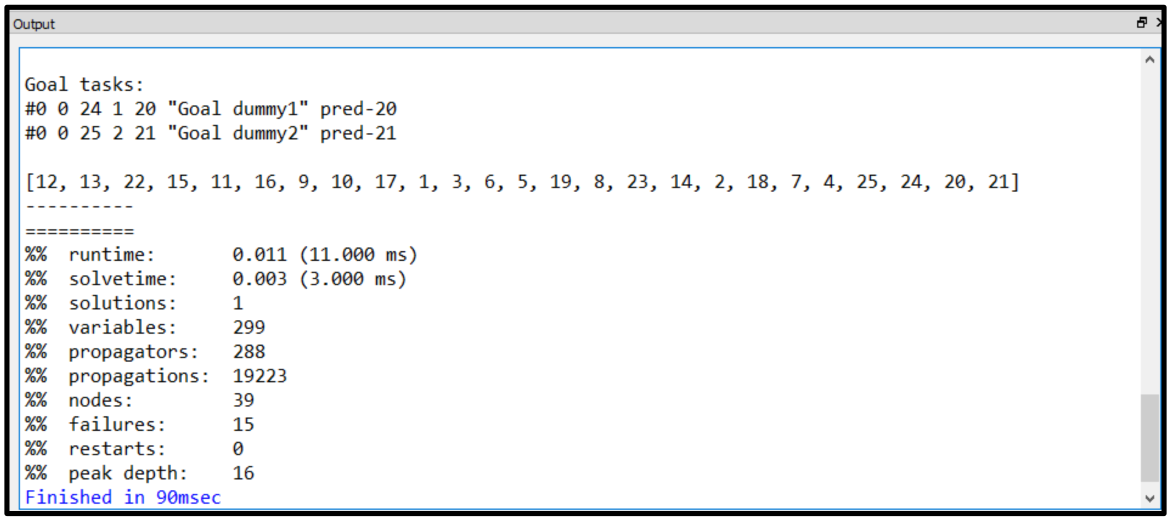
To examine the results from the changes, I ran the datafile provided by Tommy Kvant against both his model and the modified model using two different solvers, JaCoP and Gecode. In figure one and two below you can see the detailed results from the tests using Gecode from within the MiniZinc IDE.



```
Goal tasks:
#0 0 24 1 20 "Goal dummy1" pred-20
#0 0 25 2 21 "Goal dummy2" pred-21

[12, 13, 22, 15, 11, 16, 9, 10, 17, 1, 3, 6, 5, 19, 8, 23, 14, 2, 18, 7, 4, 25, 24, 20, 21]
----------
==========
%%  runtime:       0.023 (23.000 ms)
%%  solvetime:     0.015 (15.000 ms)
%%  solutions:     4
%%  variables:     299
%%  propagators:   288
%%  propagations:  152956
%%  nodes:         220
%%  failures:      96
%%  restarts:      0
%%  peak depth:    23
Finished in 166msec
```

Figure 1: Gecode Output Model 6



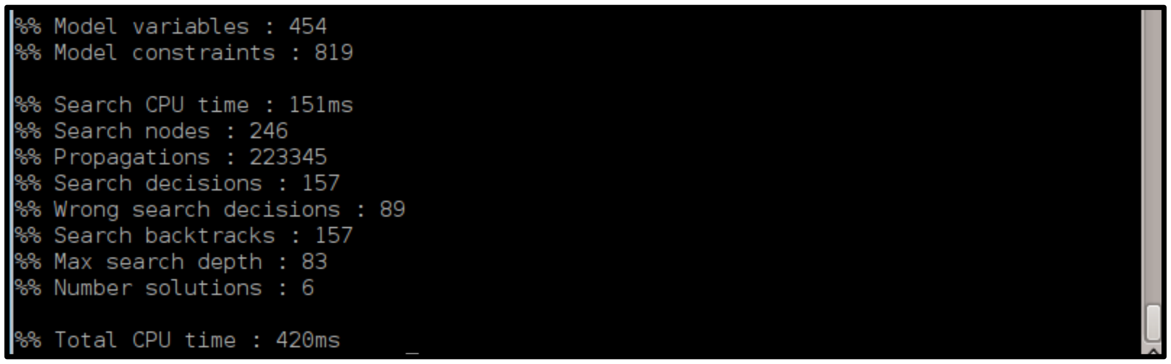
```
Goal tasks:
#0 0 24 1 20 "Goal dummy1" pred-20
#0 0 25 2 21 "Goal dummy2" pred-21

[12, 13, 22, 15, 11, 16, 9, 10, 17, 1, 3, 6, 5, 19, 8, 23, 14, 2, 18, 7, 4, 25, 24, 20, 21]
----------
==========
%%  runtime:       0.011 (11.000 ms)
%%  solvetime:     0.003 (3.000 ms)
%%  solutions:     1
%%  variables:     299
%%  propagators:   288
%%  propagations:  19223
%%  nodes:         39
%%  failures:      15
%%  restarts:      0
%%  peak depth:    16
Finished in 90msec
```

Figure 2: Gecode Output Model 7

By looking at the output we can see the differences in output. Given the new model the program finished running in 90 msec as opposed to 166 msec with the previous model. This is an increase of about 46%. Other factors to look at are the number of failures as well as propagations. Where the old model experienced 96 failures the new model only experienced 15. And when looking at the number of propagations the older model was at 152,956 compared to 19,223 for the new model.

Below are the results we received when running the two models using JaCoP through the terminal.

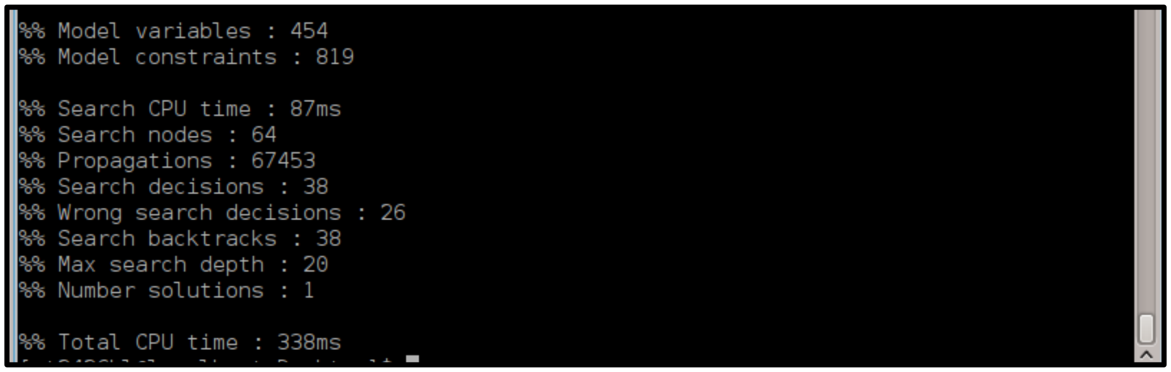


```
%% Model variables : 454
%% Model constraints : 819

%% Search CPU time : 151ms
%% Search nodes : 246
%% Propagations : 223345
%% Search decisions : 157
%% Wrong search decisions : 89
%% Search backtracks : 157
%% Max search depth : 83
%% Number solutions : 6

%% Total CPU time : 420ms
```

Figure 3: JaCoP output Model 6



```
%% Model variables : 454
%% Model constraints : 819

%% Search CPU time : 87ms
%% Search nodes : 64
%% Propagations : 67453
%% Search decisions : 38
%% Wrong search decisions : 26
%% Search backtracks : 38
%% Max search depth : 20
%% Number solutions : 1

%% Total CPU time : 338ms
```

Figure 4: JaCoP output Model 7

While the increase in speed wasn’t as dramatic while using JaCoP, the increase was still there. Going from 420 ms to 338 ms, an increase of about 20%. Similarly if we look at the values for wrong decisions we see a drop from 89 down to 26 and if we look at propagations a drop from 223,345 down to 67,453.

Lastly looking at both JaCoP and Gecode we see that the number of solutions found by the new model is only one, the optimal solution. Whereas in the previous model Gecode found four solutions, one of which being the optimal and JaCoP found 6 solutions, one of which being the optimal.

3.2 Generating data file with Assembly Conv

As mentioned earlier the current way to generate a MiniZinc datafile is to use AssemblyConv.jar using an XML file describing the tasks to be completed as well as a CSV file representing a time matrix. However, when we attempted to generate a working datafile file using AssemblyConv we ran into a few issues. This section will go into detail on what these issues were and how we modified the program to solve them.

For example the code in AssemblyConv was not entirely modified to generate a datafile consistent with the input expected by the model. It was also mentioned in Kvant’s thesis that version 2.0.1 of MiniZinc , one of the versions he used, handles empty arrays in a different way. It was able to deal with the case when there is an empty tray, thus accessing an empty array will not crash the program. However, when we were testing the same model using the new version 2.1.5, the problem occurred. To avoid future issues, we altered the program by fixing this error.

3.2.1 Changes made to Assembly Conv to solve the current issue

As previously mentioned when we attempted to generate additional examples using AssemblyConv.jar we ran into to several issues. The first one being that the original AssemblyConv.jar left in the repository still had code that included a variable called nbrCycles. As this was leftover from a previous model of Kvant’s it caused the current version to crash at runtime. To fix this error we went through all of the source code for AssemblyConv and removed any code related to calculating or printing cycles to the datafile. After fixing this we were able to replicate the main datafile based on an XML file and a CSV file provided in the repository.

Once we were able to replicate a working datafile using AssemblyConv we started to experiment with creating our own XML and CSV files. One such file was a smaller version of the original data file that describes the assembly of an emergency stop button, this new file simply describes assembling the top and button into a component referred to as top-button in the original datafile. When running AssemblyConv.jar on these new files it appeared to have generated a working data file, however when running the actual model we recieved an index out of bounds error. After some testing we discovered a couple sources for this error.

The first being that we had defined trays at the top of the XML but never used them in any of the tasks that followed. While this seems to be of little consequence to the actual process of planning, it resulted in the error being thrown by the model.

To fix the issue of having empty trays, we first try adding more constraints to the model to accommodate the scenario. For example, we try to write an “if-else” statement in MiniZinc language, to separate the empty trays from the others. However, there would need to be a series of modifications after this change, and an if-else statement does not handle the issue really well given the constraint programming method. We then tried to add a warning message to remind the user not to include the unused trays after the model takes in all of the data input. However, the best way to do this is to ask the user themselves to define all of the available trays as well as the number of used trays, if the numbers are different, it means there are empty trays that will crash the program. This is the same thing as just telling the user not to include the empty tray. Usually only when users are careless will they make this mistake, and this change will not account for that. Therefore, we abort the plan of making the change within the model.

After attempting to modify the model to prevent this error, we decided a better option might be to modify AssemblyConv from allowing a user to generate a datafile that caused these errors.

To account for the unused tray we modified AssemblyConv to keep track of all trays referenced in tasks and compared it with a list of all defined trays after the file was parsed, if these lists were not equal we throw an error letting the user know that their XML contains an unused tray.

The second error resulted from a lack of understanding when it came to fixtures and outputs. We accidentally placed a fixture and an output in the same task. While an obvious error, this wasn’t immediately apparent to us as we attempted to run our tests. To fix this, and prevent AssemblyConv from generating a datafile from bad XML, we introduced a variable to keep track of the current number of fixtures and outputs for a given task. If this number is greater than one we report an error to the user informing them of the mistake.

4. Evaluation of New Model Running with Different Solvers

The computer used to run the solver is installed Ubuntu 16.04.2, 160GB. An IDE is used to compile and solve the MiniZinc files for a clear presentation.

MiniZinc Version: MiniZincIDE-2.1.5-bundle-linux-x86_64/
The versions of each of the solvers are presented in the table below

Solver	Version
Gecode	5.1.0
JaCoP	4.5.0
G12	G12_fd

Table 1. Versions of Softwares used in the testing.

Two sample tasks with different level of complexity were tested and solved using the solvers listed above. The first one was created by Kvant in his thesis in 2015 which is an assembly of an emergency stop button, it has a higher complexity. The second task was created by Ethan, which is a simple task of assemble only two components together. The corresponding source code can be find at the project directory.

4.1 Running Results

For every solver, each task was run for 3 times, and the averages were taken to represent the regular circumstances. The raw data can be found at Appendix 7.1. All of the solvers below were tested with the final model that we have modified.

4.1.1 Task I. Assembly of the Emergency Stop Button

This task is defined by the data file xml_in_minizinc.dzn.

Gecode

Solver Runtime	7.531ms
Solve Time	2.755ms
Solutions	1

Variables	299
Propagators	288
Propagations	19791
Search Nodes	39
Failures	15
Restarts	0
Max Search Depth	16
Total Finishing Run Time	44ms

Table 2. Average performance of Gecode on the complex task.

JaCoP

Model Variables	256
Model Constraints	333
Search CPU Time	67ms
Search Nodes	41
Propagations	35930
Search Decisions	26
Wrong Search Decisions(Failures)	15
Search Backtracks	26
Max Search Depth	17
Number Solutions	1
Total CPU time	239ms
Total Finishing Run Time	338ms

Table 3. Average performance of JaCoP on the complex task.

G12

Number of Choice Point Explored	59
Total Finishing Run Time	127ms

Table 4. Average performance of G12 on the complex task.

4.1.2 Task II. Simple Assembly Test

This task is defined by the data file xml_samll_minizinc_test.dzn.

Gecode	
Solver Runtime	1.718ms
Solve Time	0.300ms
Solutions	1
Variables	81
Propagators	31
Propagations	223
Nodes	9
Failures	1
Restarts	0
Peak Depth	8
Total Finishing Run Time	39ms

Table 5. Average performance of Gecode on the simple task.

JaCoP	
Model Variables	69
Model Constraints	59
Search CPU Time	11ms
Search Nodes	8
Propagations	281
Search Decisions	8
Wrong Search Decisions	0
Search Backtracks	8
Max Search Depth	8
Number Solutions	1

Total CPU time	143ms
Total Finishing Run Time	234ms

Table 6. Average performance of JaCoP on the simple task.

G12	
Number of Choice Point Explored	40
Total Finishing Run Time	33 msec

Table 7. Average performance of G12 on the simple task.

4.2 Discussion

From the results obtained above, we can observe that, for the complex model of assembling the emergency stop button, Gecode is the most efficient solver, and for the simple task, G12 has the least runtime.

When solving the more complex task, Gecode and JaCoP have similar number of propagation nodes(39 and 41), same number of failure times(15 times), similar search depth(16 and 17), and they both successfully found the same optimal solution. The significant difference between these two solvers are the number of propagations made and the runtime. It is known that Gecode exploits the multiple cores of today's commodity hardware for parallel search [2], which largely improves the operation efficiency and allows Gecode to be very suitable for solving complex tasks.

However, when it comes down to the very simple tasks, Gecode’s performance is not as good as G12, an efficient finite domain solver. [3] The run time of G12 solver is largely affected by the complexity of the task. It can be seen as a good choice for simple tasks, but it runs slower for more difficult tasks.

JaCoP, in this testing model, performs relatively not ideal since it took the most amount of time among all 3 solvers tested. More tests are needed to find the properties of how JaCoP performs on this model.

Figure 5 is a model showing the trend of how the 3 solvers behave responding to changes in complexity. This graphical model could be useful in choosing the appropriate solver for given task complexity.

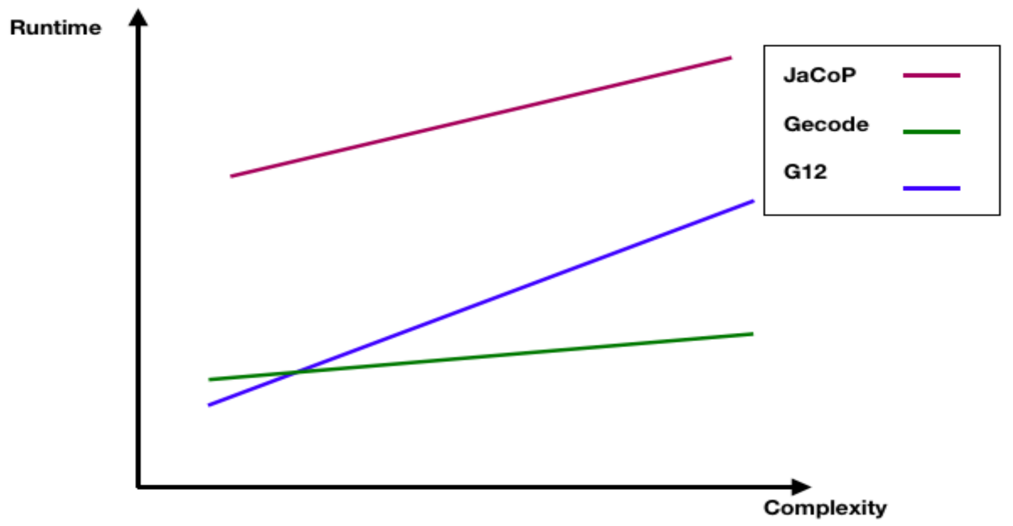


Figure 5. Run time versus task complexity of JaCop, Gecode, and G12.

5. Discussion

Given the objective of this project is to create an easy access for users to program the robots, it is also useful to discuss the scheduler's flexibility in adapting to other domains other than doing assembly tasks. This research can also be seen as an update and further study of Kvant's thesis, thus information regarding on how domain filters performs on the modified model with the updated software is worth discussing.

5.1 Adaptation Potential to Other Domains

The current model only allows the robot arms to perform 4 actions, "Taking", "Putting", "Moving", and "Mounting". These actions are almost sufficient for the machines to do simple tasks like assembling, taking and transporting objects when equipped with appropriate tools. In the future, this program can be adapted to many other domains with a little modification.

There are many ways to perform the adaptation, and we think it is necessary for the changes to be made in two aspects.

I. Generation of Data File

As the current way of generating a data file being converting an .xml file and a time matrix using AssemblyConv.jar, it is necessary to modify from the source code of how AssemblyConv parses the xml file, and how it arranges the new elements. For example, if one wants to add other actions such as “photo” or “peeling”, it is necessary to define the new element in the “objects” package, modify the parser, and allow it to print out the in the right format of an MiniZinc data file.

II. MiniZinc Model

Once the model was modified or new elements were added, there should be new constraints specified. As most of the constraints comes from common senses and simple logic as the current model does, they could be implemented in a similar fashion. Therefore, once we simply replicate the same constraints for the new elements, and specify special requirements, the model should be completely suitable for the new domain.

5.2 Filters

In Kvant’s model, he applied two filters to narrow down the searching domain to cut down the search time. A “temporal filter” limits the range of the variables dealing with time, for example, “one machine arm can only do one action at a time”, or “all of the actions have to end before the maximum end time”. A “predecessor filter” applies many logical conditions in common senses, such as “a putting task cannot happen before a taking task”, or “a component cannot be used before it is created”, to constrain the sequence of tasks. These two filters are expected to increase the search efficiency and may have a different effect when working with different solvers. In the end, we observed a similar result as Kvant did, using the newest version 2.1.5 of MiniZinc and other solvers.

For all 3 solvers, the temporal filter plays an important role: if it is disabled, the program will not finish running in a reasonable time. However, the predecessor filter does not have a significant impact on runtime, it even slows down the program when solving using Gecode and G12.

5.3 Future Work

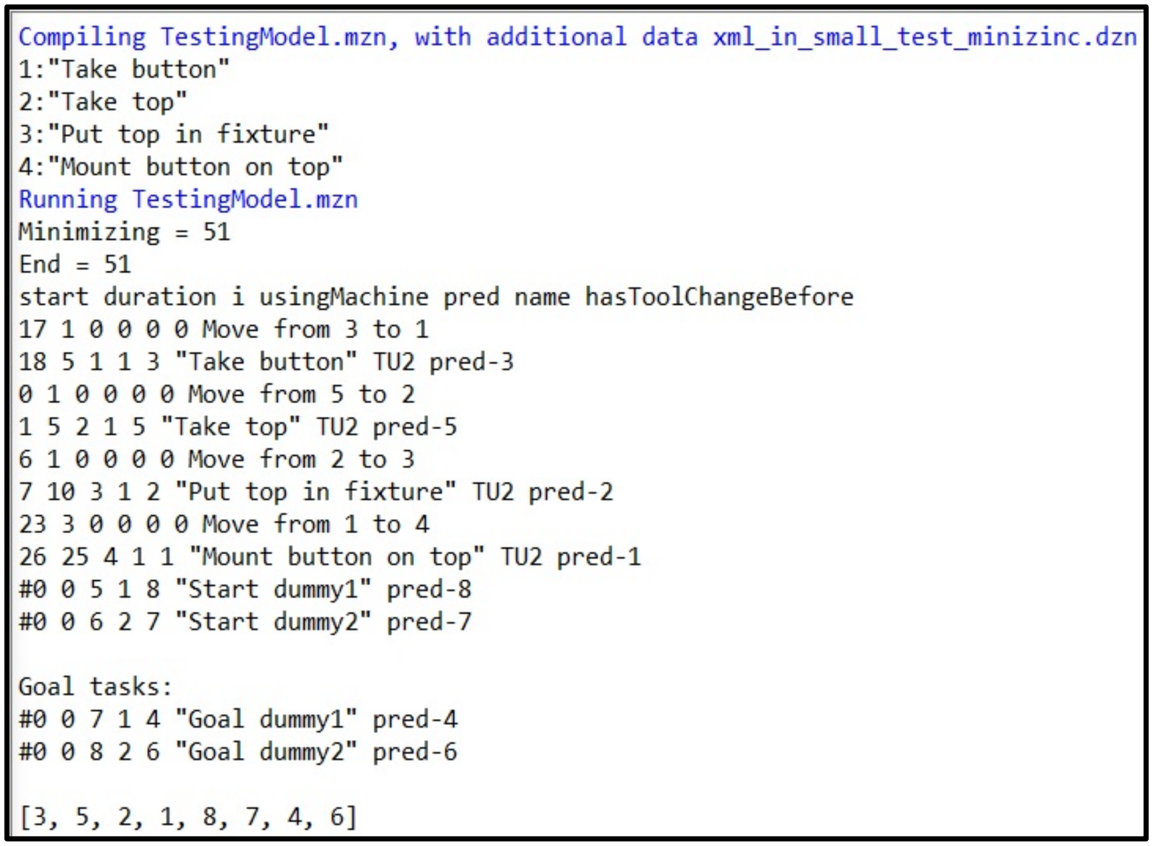
In this section we will discuss a couple ways that the project can continue to be improved and applied in the future.

5.3.1 Output

One area for potential improvement in this project is with output and how it is parsed. Currently the output is in the format of “start duration i usingMachine pred name hasToolChangeBefore” where:

- start : The point in time that the task begins.
- duration: The amount of time units it takes to finish the task.
- i: The task ID
- usingMachine: The ID of the machine that carries out the task.
- pred: The ID of the task executed prior to this task
- name: The string associated with the task declared in the XML by the user.
- hasToolChangeBefore: The tool used and predecessor.

While this gets the majority of the information across it can be somewhat difficult for a human to parse. Below you can see a full example of the output in figure 6. The output is generated by xml_in_small_test_minizinc.dzn, which can be found in the Model 7 directory of the git repository under DataFile.



```
Compiling TestingModel.mzn, with additional data xml_in_small_test_minizinc.dzn
1:"Take button"
2:"Take top"
3:"Put top in fixture"
4:"Mount button on top"
Running TestingModel.mzn
Minimizing = 51
End = 51
start duration i usingMachine pred name hasToolChangeBefore
17 1 0 0 0 0 Move from 3 to 1
18 5 1 1 3 "Take button" TU2 pred-3
0 1 0 0 0 0 Move from 5 to 2
1 5 2 1 5 "Take top" TU2 pred-5
6 1 0 0 0 0 Move from 2 to 3
7 10 3 1 2 "Put top in fixture" TU2 pred-2
23 3 0 0 0 0 Move from 1 to 4
26 25 4 1 1 "Mount button on top" TU2 pred-1
#0 0 5 1 8 "Start dummy1" pred-8
#0 0 6 2 7 "Start dummy2" pred-7

Goal tasks:
#0 0 7 1 4 "Goal dummy1" pred-4
#0 0 8 2 6 "Goal dummy2" pred-6

[3, 5, 2, 1, 8, 7, 4, 6]
```

Figure 6: Output using xml_in_small_test_minizinc.dzn

The first four lines of the output that are in black tell the user the name of the tasks along with their task ID. In this case “Take Button” has an i value of 1, “Take top” has an i value of 2 and so on. After this the output tells you the number of time units it will take to complete all of the tasks, before moving onto the format previously mentioned. Currently the output is printed out in an order that is not consistent with the order tasks occur. While not difficult for a computer to parse back into the correct order it may be slightly harder for a human to understand just by looking at it. By looking at the output we can see the start time of 0 occurs three lines in and the output continues in order until line seven where it jumps back up to line 1 being the next task. Another aspect of the output is the "Start" and "Goal" dummies. Each arm has a corresponding start and goal dummy that represent when an arm starts and stops being in use. Lastly, the array located on the bottom of the output is an ordered list of predicates.

5.3.2 Robot Integration

One long term goal for this project is the integration of the scheduler with a multi-armed robot. At it's current state there is still a lot of work to be done with project to get to this point. One of the major obstacles in regards to this goal is the fact that getting a robot to execute a task is not currently as simple as simply calling a function written in code. If this were the case it would be possible to simply have the output be parsed and have the task names correlate to such functions. As things are now, programming such things require more effort from the operator, since working in a real world environment introduces new constraints. For example much of the programming is done using a lead-through technique that would require a user to train the robot for each task that a scheduler schedules.

6. Conclusion

This study allows the readers to have a better and clearer understanding of the process for transcribing a robotic task into data that can be used by the scheduler in order to output an optimized schedule. A thorough explanation of output was given, including how to write an XML file, time matrix and how to use them to generate the MiniZinc data files.The MiniZinc scheduler model was made more robust through the improvement made to the model as well as AssemblyConv. This includes the handling of more errors and giving feedback to the user on how to fix their XML. Such as printing warning message when there are misuses of the definition of an "Output" and "Fixtures" or having unused trays. To conclude, this report explains the work we have done, updates the filters' and solvers' performance with MiniZinc version 2.1.5, and suggests a potential of adapting it to other domains as well as possible future works.

7. Appendix

Files and raw data mentioned in the text are presented in this section.

7.1 Annotated .xml file

This file can also be found at the project directory.

```
<?xml version="1.0" encoding="UTF-8"?>

<!--start-flag of the section, in this case the section is named as
"Assembly"-->
<Assembly>

<!--Define the Output id. There must have and only allow one output in
a model-->
      <Output id="Output"/>

<!--Define all of the Trays needed. Note: It is not allowed to include
Trays that are not used, otherwise the program will generate an error.
Each of the Tray ID has to be unique, otherwise the program will
generate an error.-->
      <Tray id="name_of_the_first_tray"/>
      ...
      <Tray id="name_of_the_last_tray"/>


<!--Define all of the Fixtures needed. Note: It is not allowed to
include Fixtures that are not used, otherwise the program will generate
an error. Each of the Fixture ID has to be unique, otherwise the
program will generate an error.-->
      <Fixture id="name_of_the_first_fixture"/>
<!-- ...    -->
      <Fixture id="name_of_the_last_fixture"/>


<!-- Define all of the components under the Component tag. Include both
primitive components(basic units) as well as the subcomponents(sub-
assemblies) that are involved in the task, since the sub-assemblies are
also treated as components in the model-->
      <Component id="name_of_the_first_Component"/>
<!-- ...    -->
      <Component id="name_of_the_last_Component"/>


<!--Define all of the Subcomponents that are expected to appear during
the task. This segment of code describes what components make up a
subcomponent using the Subcomponents tag, which can be both regular
components or sub-assemblies.
An example of definition of a Subcomponent called "top_bottom_base"
that consists subcomponent "top_bottom" and component "base" is defined
as follow.-->
      <Subcomponents id="top_bottom_base">
            <Component id="top_bottom"/>
            <Component id="base"/>
      </Subcomponents>

<!--Define all of the tools used in the task under the tag Tool. Note
the each of the Tool id has to be unique.-->
      <Tool id="tool_1"/>
<!-- ...    -->
      <Tool id="tool_n"/>

<!--Define all of the machines available in the task under the tag
Machine. Note the each of the Machine id has to be unique. In this
case, the two robotic arms are treated as two independent machines
named "m1", and "m2"-->
      <Machine id="m1"/>
      <Machine id="m2"/>
<!--Define all of the individual Tasks and their task ids, durations
Followed by each one's required elements, such as Fixture/Tray,
relevant Components/Subcomponents(must have), ToolNeeded(must have),
Action(one of "Taking", "Putting", "Mounting", "Moving"),
ComponentCreated(if there is a new subcomponent created), Output(if it
is the final task that will generate an output)
%Example:              -->
      <Task id="Mount button on top" Duration="25">
            <Fixture id="Front fixture"/>
            <Component id="Top"/>
            <Component id="Button"/>
            <ComponentCreated id="Top-Button"/>
            <ToolNeeded id="tool1"/>
            <Action id="Mounting"/>
      </Task>

<!--Define special requirements, such as OrderedGroup(A group of
Actions that have to happened in specific orders),
ConcurrentGroup(Actions that have to happen at the same time)
%Example:         -->
      <OrderedGroup>
            <Task id="task_name_1"/>
            <Task id="task_name_2"/>
<!-- ...    -->
      </OrderedGroup>

      <ConcurrentGroup>
            <Task id="Mount ring on top-button, hold"/>
            <Task id="Mount ring on top-button, mount"/>
      </ConcurrentGroup>


<!-- Define out-of-range tasks for each machine.  TasksOutOfRange
happens when a task is unable to be reached or fulfilled by a specific
machine, which means that task can only be performed by another
machine.
%Example of TasksOutOfRange list for machine m1 and m2           -->
      <TasksOutOfRange id="m1">
            <Task id="task_name_1"/>
            ...
            <Task id="task_name_n"/>
      </TasksOutOfRange>
      <TasksOutOfRange id="m2">
            <Task id="task_name_1"/>
            ...
            <Task id="task_name_m"/>
      </TasksOutOfRange>

<!--Define the duration of tool change between each tools
%Example:                  -->
      <ToolChangeDurations>
            <Change FromToolId="tool1" ToToolId="tool2" Duration="60"/>
            <Change FromToolId="tool2" ToToolId="tool1" Duration="60"/>
      </ToolChangeDurations>

<!--end-flag of the section->
</Assembly>
```

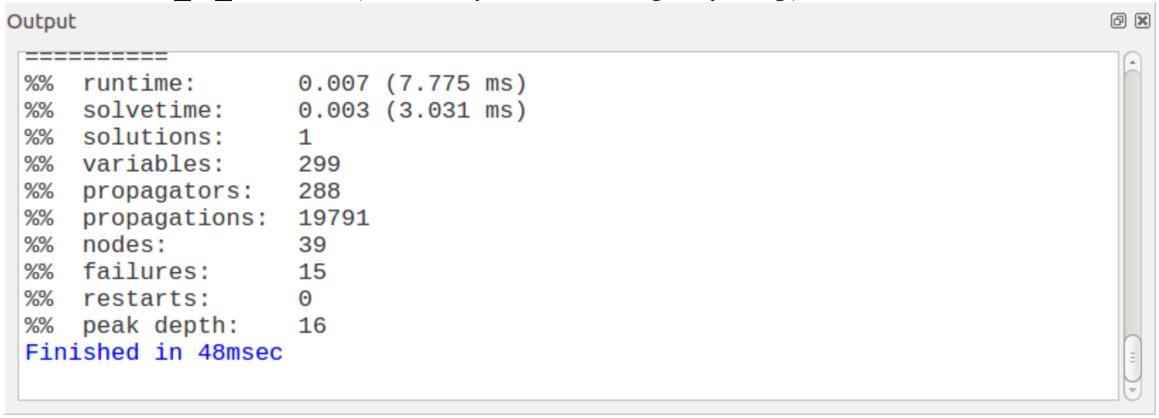
7.2 Raw data of Solver Testing

All of the solvers below were tested with the final model that we have modified
Each ran for 3 times and the averages were taken.

MiniZinc Model: MiniZincIDE-2.1.5-bundle-linux-x86_64/

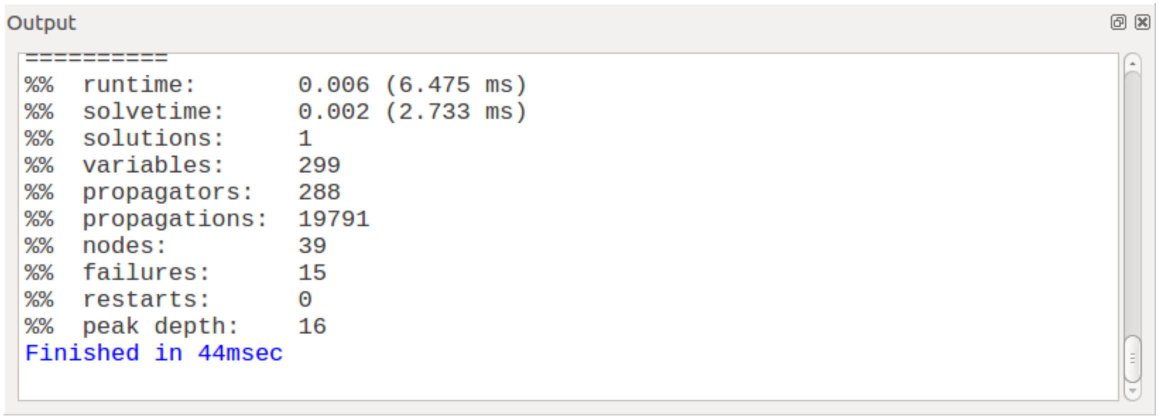
I. Gecode

i. xml_in_minizinc (Assembly of the Emergency Stop)



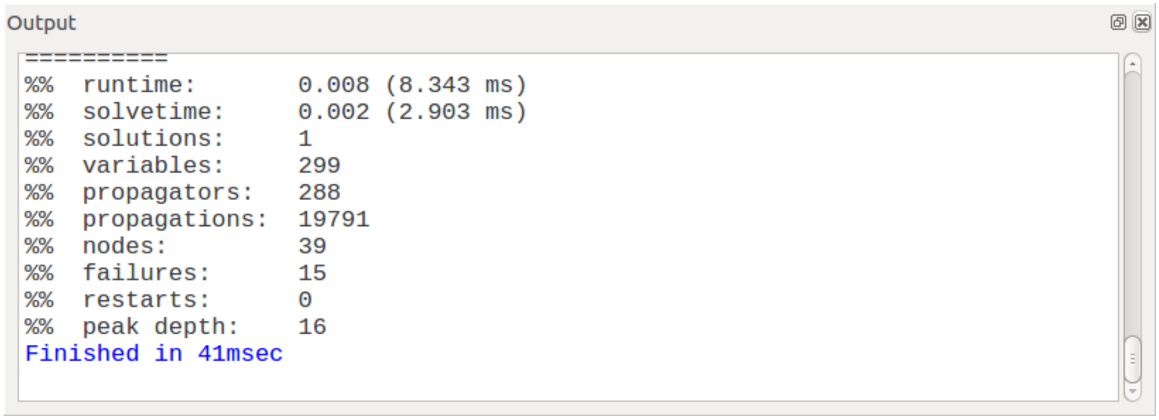
```
==========
%%   runtime:       0.007 (7.775 ms)
%%   solvetime:     0.003 (3.031 ms)
%%   solutions:     1
%%   variables:     299
%%   propagators:   288
%%   propagations:  19791
%%   nodes:         39
%%   failures:      15
%%   restarts:      0
%%   peak depth:    16
Finished in 48msec
```

Figure 7. Run 1 results of Gecode on the complex task.


Output

```
==========
%%  runtime:       0.006 (6.475 ms)
%%  solvetime:     0.002 (2.733 ms)
%%  solutions:     1
%%  variables:     299
%%  propagators:   288
%%  propagations:  19791
%%  nodes:         39
%%  failures:      15
%%  restarts:      0
%%  peak depth:    16
Finished in 44msec
```

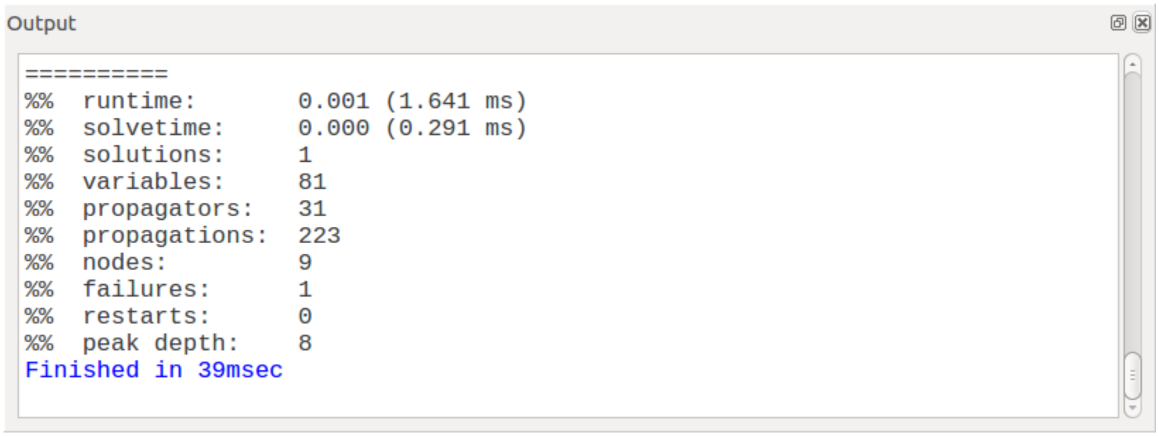
Figure 8. Run 2 results of Gecode on the complex task.


Output

```
==========
%%  runtime:       0.008 (8.343 ms)
%%  solvetime:     0.002 (2.903 ms)
%%  solutions:     1
%%  variables:     299
%%  propagators:   288
%%  propagations:  19791
%%  nodes:         39
%%  failures:      15
%%  restarts:      0
%%  peak depth:    16
Finished in 41msec
```

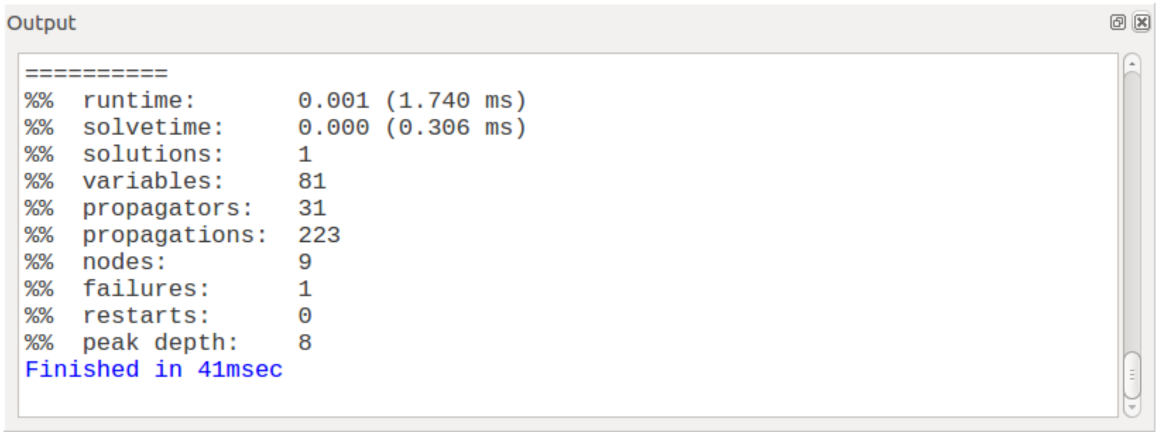
Figure 9. Run 3 results of Gecode on the complex task.

ii. xml_small_minizinc_test


Output

```
==========
%%  runtime:       0.001 (1.641 ms)
%%  solvetime:     0.000 (0.291 ms)
%%  solutions:     1
%%  variables:     81
%%  propagators:   31
%%  propagations:  223
%%  nodes:         9
%%  failures:      1
%%  restarts:      0
%%  peak depth:    8
Finished in 39msec
```

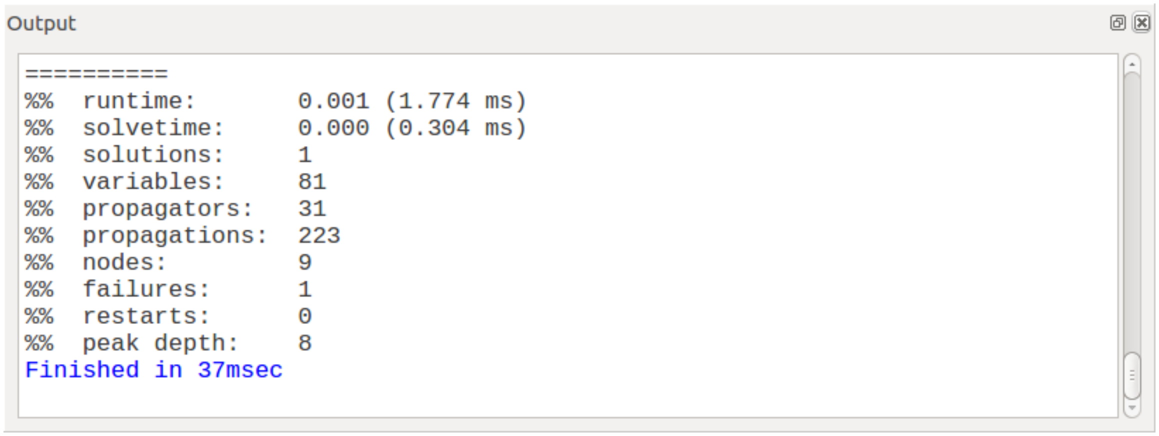
Figure 10. Run 1 results of Gecode on the simple task.



Output

```
==========
%%  runtime:       0.001 (1.740 ms)
%%  solvetime:     0.000 (0.306 ms)
%%  solutions:     1
%%  variables:     81
%%  propagators:   31
%%  propagations:  223
%%  nodes:         9
%%  failures:      1
%%  restarts:      0
%%  peak depth:    8
Finished in 41msec
```

Figure 11. Run 2 results of Gecode on the simple task.



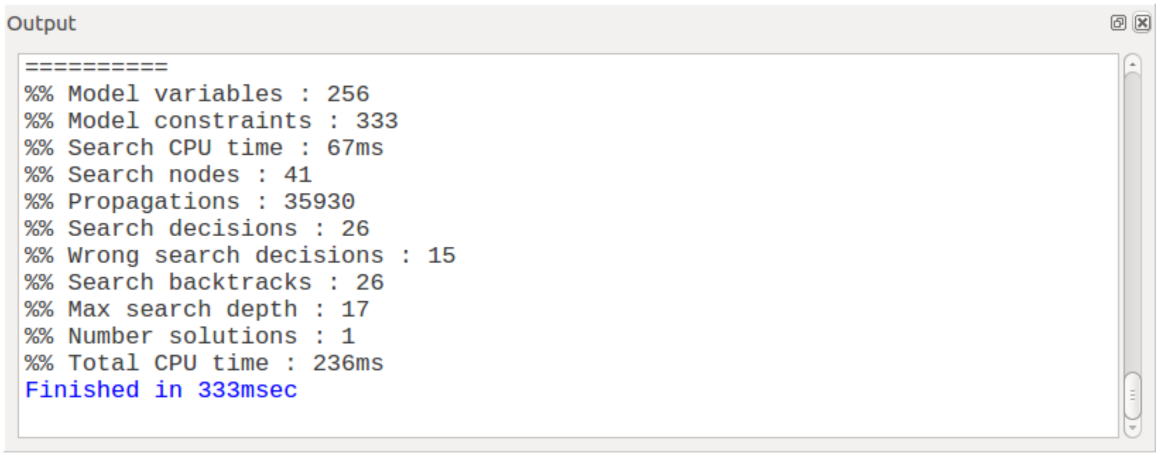
Output

```
==========
%%  runtime:       0.001 (1.774 ms)
%%  solvetime:     0.000 (0.304 ms)
%%  solutions:     1
%%  variables:     81
%%  propagators:   31
%%  propagations:  223
%%  nodes:         9
%%  failures:      1
%%  restarts:      0
%%  peak depth:    8
Finished in 37msec
```

Figure 12. Run 3 results of Gecode on the simple task.

II. JaCoP

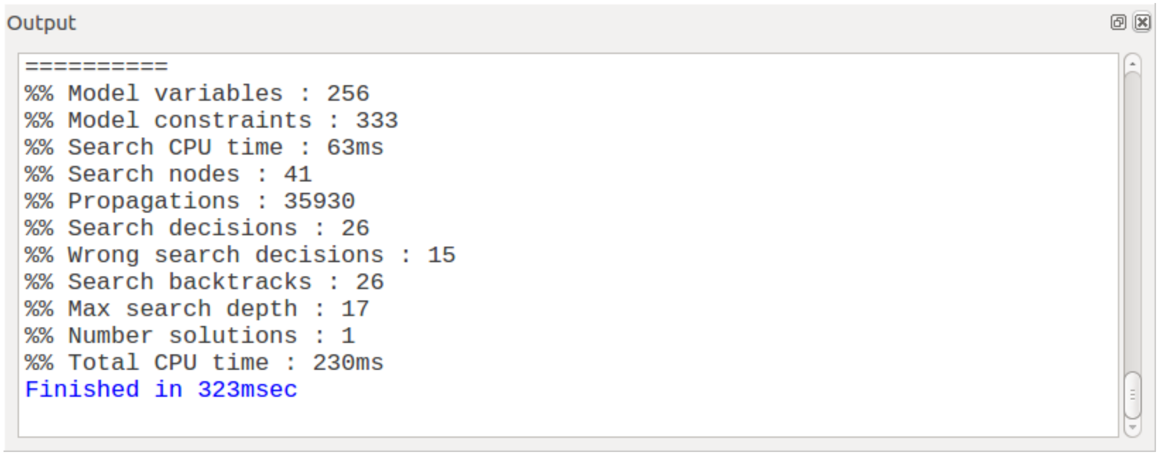
i. xml_in_minizinc (Assembly of the Emergency Stop)



Output

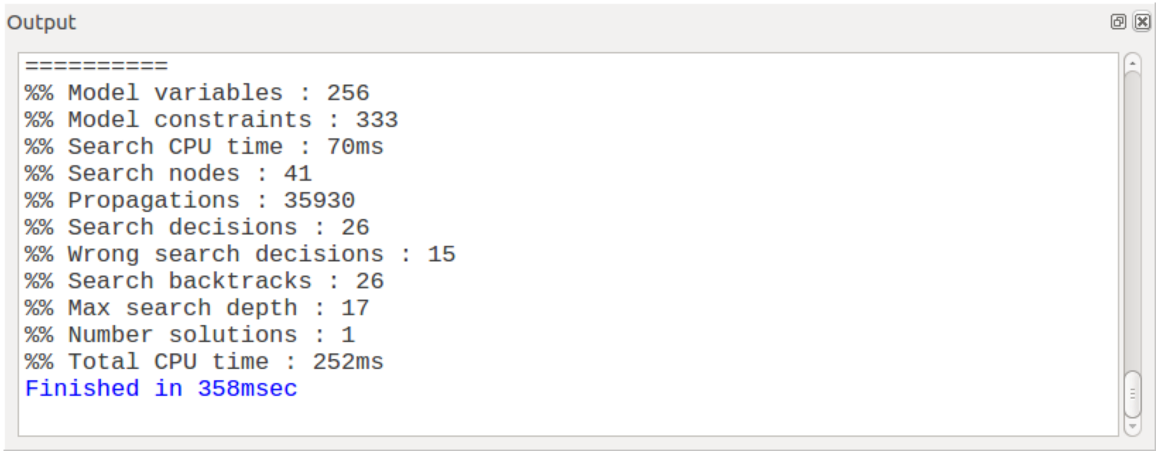
```
==========
%% Model variables : 256
%% Model constraints : 333
%% Search CPU time : 67ms
%% Search nodes : 41
%% Propagations : 35930
%% Search decisions : 26
%% Wrong search decisions : 15
%% Search backtracks : 26
%% Max search depth : 17
%% Number solutions : 1
%% Total CPU time : 236ms
Finished in 333msec
```

Figure 13. Run 1 results of JaCoP on the complex task.



```
==========
%% Model variables : 256
%% Model constraints : 333
%% Search CPU time : 63ms
%% Search nodes : 41
%% Propagations : 35930
%% Search decisions : 26
%% Wrong search decisions : 15
%% Search backtracks : 26
%% Max search depth : 17
%% Number solutions : 1
%% Total CPU time : 230ms
Finished in 323msec
```

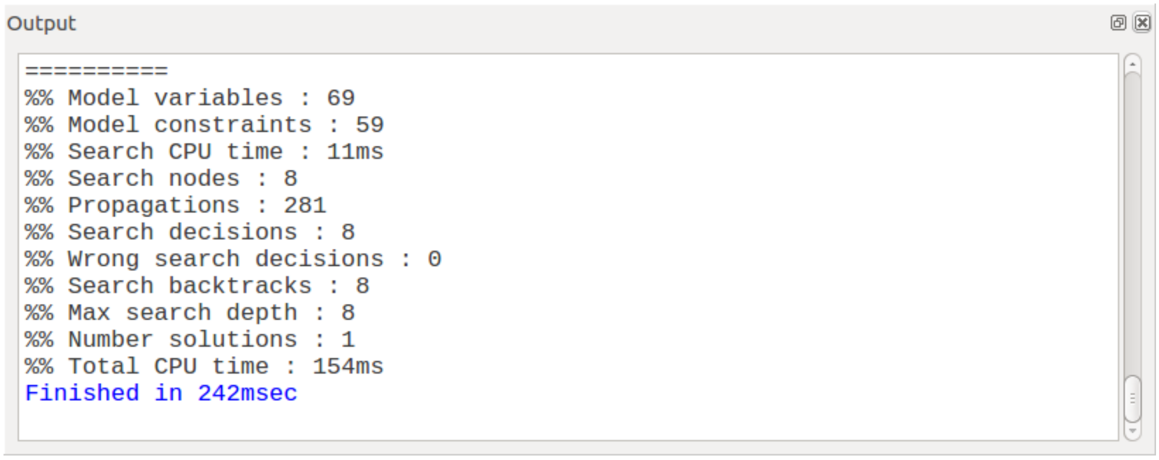
Figure 14. Run 2 results of JaCoP on the complex task.



```
==========
%% Model variables : 256
%% Model constraints : 333
%% Search CPU time : 70ms
%% Search nodes : 41
%% Propagations : 35930
%% Search decisions : 26
%% Wrong search decisions : 15
%% Search backtracks : 26
%% Max search depth : 17
%% Number solutions : 1
%% Total CPU time : 252ms
Finished in 358msec
```

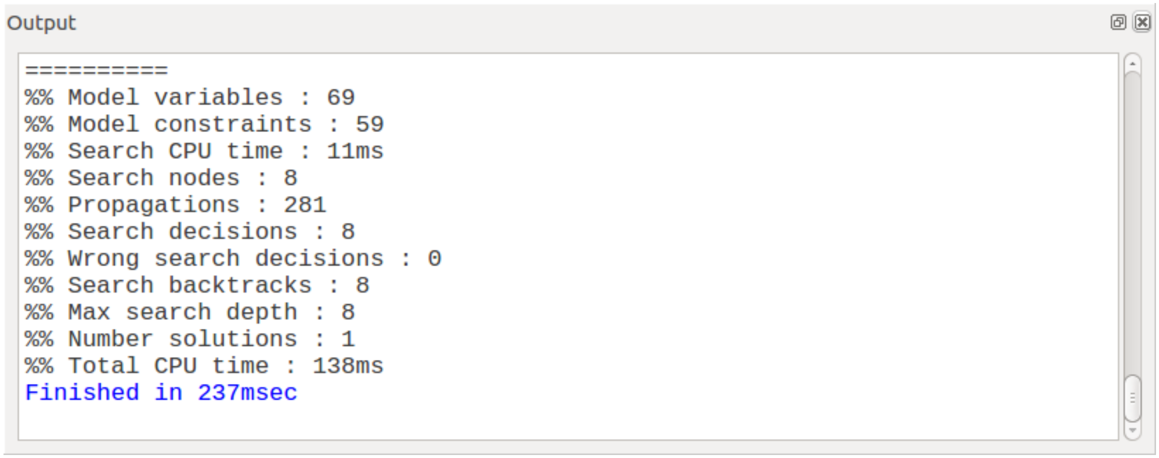
Figure 15. Run 3 results of JaCoP on the complex task.

ii. xml_small_minizinc_test



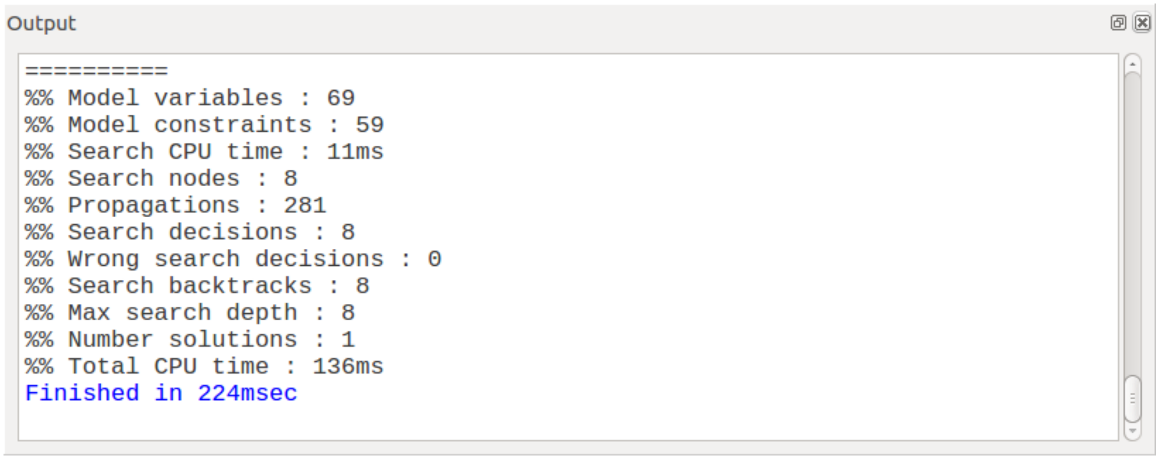
```
==========
%% Model variables : 69
%% Model constraints : 59
%% Search CPU time : 11ms
%% Search nodes : 8
%% Propagations : 281
%% Search decisions : 8
%% Wrong search decisions : 0
%% Search backtracks : 8
%% Max search depth : 8
%% Number solutions : 1
%% Total CPU time : 154ms
Finished in 242msec
```

Figure 16. Run 1 results of JaCoP on the simple task.



```
Output
==========
%% Model variables : 69
%% Model constraints : 59
%% Search CPU time : 11ms
%% Search nodes : 8
%% Propagations : 281
%% Search decisions : 8
%% Wrong search decisions : 0
%% Search backtracks : 8
%% Max search depth : 8
%% Number solutions : 1
%% Total CPU time : 138ms
Finished in 237msec
```

Figure 17. Run 2 results of JaCoP on the simple task.

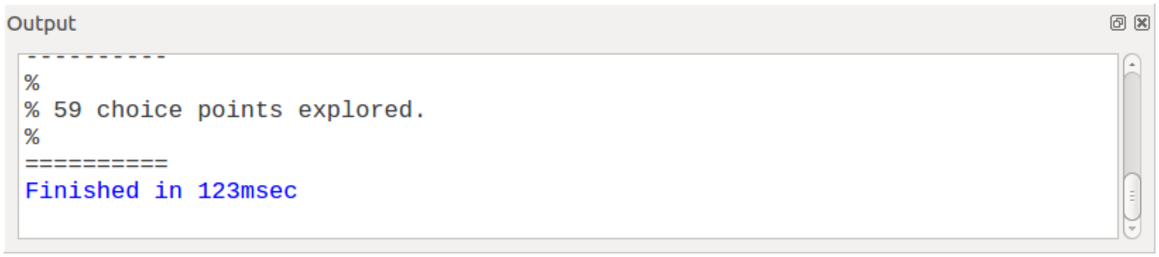


```
Output
==========
%% Model variables : 69
%% Model constraints : 59
%% Search CPU time : 11ms
%% Search nodes : 8
%% Propagations : 281
%% Search decisions : 8
%% Wrong search decisions : 0
%% Search backtracks : 8
%% Max search depth : 8
%% Number solutions : 1
%% Total CPU time : 136ms
Finished in 224msec
```

Figure 18. Run 3 results of JaCoP on the simple task.

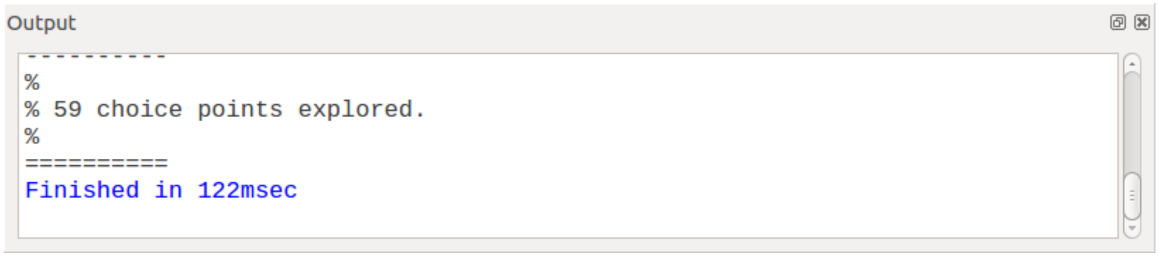
III. G12

i. xml_in_minizinc (Assembly of the Emergency Stop)



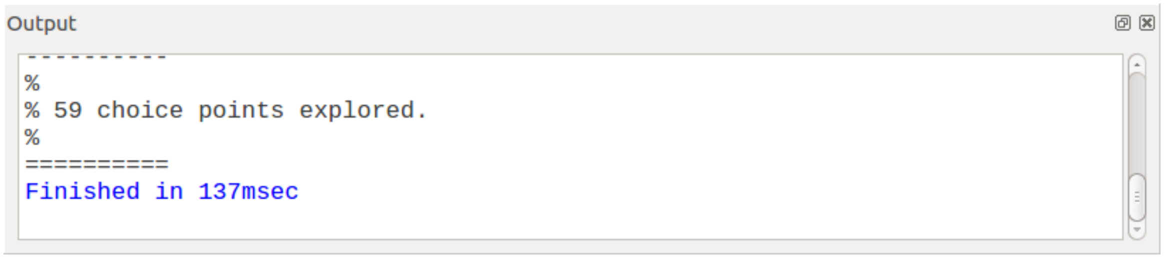
```
Output
----------
%
% 59 choice points explored.
%
==========
Finished in 123msec
```

Figure 19. Run 1 results of G12 on the complex task.



```
----------
%
% 59 choice points explored.
%
==========
Finished in 122msec
```

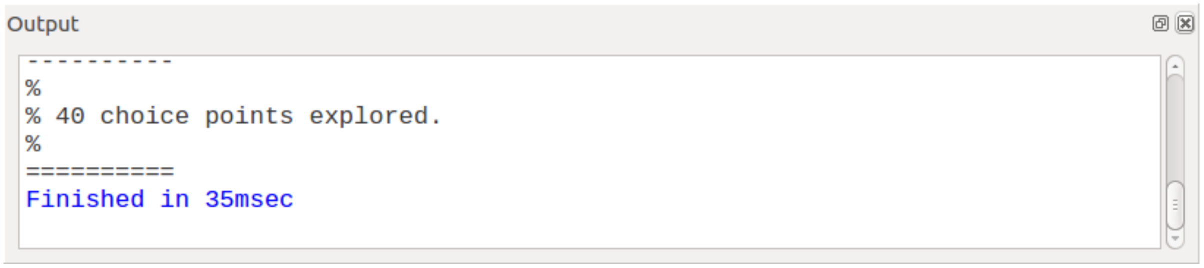
Figure 20. Run 2 results of G12 on the complex task.



```
----------
%
% 59 choice points explored.
%
==========
Finished in 137msec
```

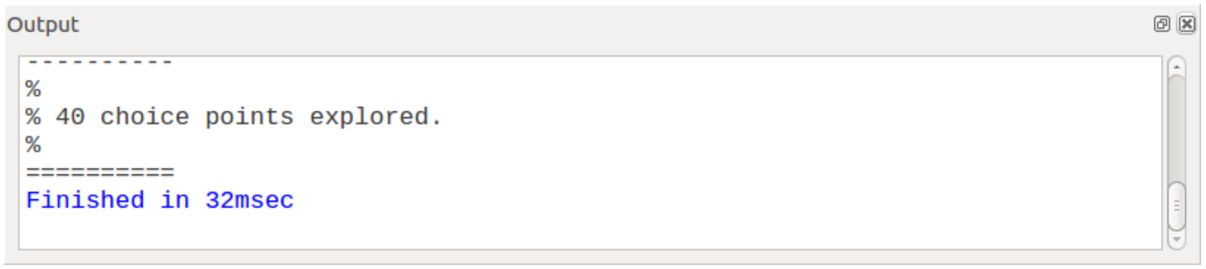
Figure 21. Run 3 results of G12 on the complex task.

ii. xml_small_minizinc_test



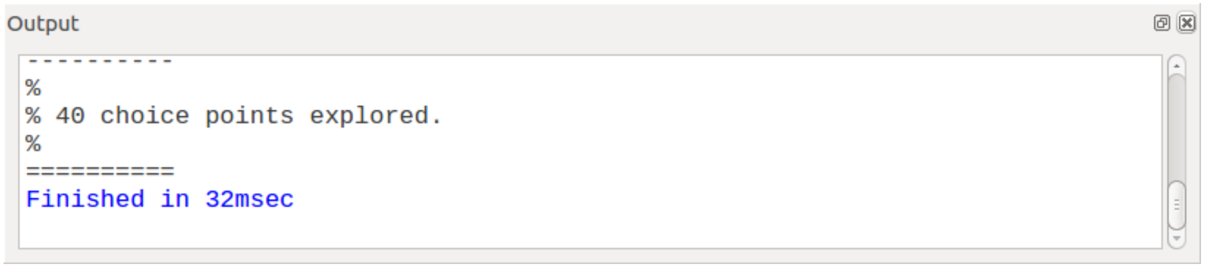
```
----------
%
% 40 choice points explored.
%
==========
Finished in 35msec
```

Figure 22. Run 1 results of G12 on the simple task.



```
----------
%
% 40 choice points explored.
%
==========
Finished in 32msec
```

Figure 23. Run 2 results of G12 on the simple task.


Output

```
----------
%
% 40 choice points explored.
%
==========
Finished in 32msec
```

Figure 24. Run 3 results of G12 on the simple task.

8. References

[1]Task scheduling for dual-arm industrial robots through Constraint Programming.(2015). Tommy Kvant. Retrieved 30 June 2017, from https://git.cs.lth.se/jacek/constraintsandrobots.git

[2]GECODE - An open, free, efficient constraint solving toolkit. (2017). *Gecode.org*. Retrieved 30 June 2017, from http://www.gecode.org

[3]G12 MiniZinc Distribution - Downloads. (2017). *Minizinc.org*. Retrieved 30 June 2017, from http://www.minizinc.org/g12distrib.html