Using the Soar Cognitive Architecture to Remove Screws from Different Laptop Models

Nicholas M. Difilippo
University of Rhode Island

Musa Jouaneh
University of Rhode Island, jouaneh@uri.edu

Follow this and additional works at: https://digitalcommons.uri.edu/mcise_facpubs

The University of Rhode Island Faculty have made this article openly available. Please let us know how Open Access to this research benefits you.

Terms of Use
This article is made available under the terms and conditions applicable towards Open Access Policy Articles, as set forth in our Terms of Use.

Citation/Publisher Attribution

This Article is brought to you for free and open access by the Mechanical, Industrial & Systems Engineering at DigitalCommons@URI. It has been accepted for inclusion in Mechanical, Industrial & Systems Engineering Faculty Publications by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons-group@uri.edu.
Using the Soar Cognitive Architecture to Remove Screws from Different Laptop Models

Keywords
Automated disassembly; cognitive architecture; computer vision; disassembly tooling; electronic waste (e-waste); robotic disassembly; Soar cognitive architecture

The University of Rhode Island Faculty have made this article openly available. Please let us know how Open Access to this research benefits you.

This is a pre-publication author manuscript of the final, published article.

Terms of Use
This article is made available under the terms and conditions applicable towards Open Access Policy Articles, as set forth in our Terms of Use.

This article is available at DigitalCommons@URI: https://digitalcommons.uri.edu/mcise_facpubs/427
Using the Soar Cognitive Architecture to Remove Screws from Different Laptop Models

Nicholas M. DiFilippo and Musa K. Jouaneh, Senior Member IEEE

Abstract—This paper investigates an approach that uses the cognitive architecture Soar to improve the performance of an automated robotic system, which uses a combination of vision and force sensing to remove screws from laptop cases. Soar’s long term memory module, semantic memory, was used to remember pieces of information regarding laptop models and screw holes. The system was trained with multiple laptop models and the method in which Soar was used to facilitate the removal of screws was varied to determine the best performance of the system. In all cases, Soar could determine the correct laptop model and in what orientation it was placed in the system. Soar was also used to remember what circle locations that were explored contained screws and what circles did not. Remembering the locations of the holes decreased a trial time by over 60%. The system performed the best when the number of training trials used to explore circle locations was limited, as this decreased the total trial time by over 10% for most of the laptop models and orientations.

Note to Practitioners: Although the amount of discarded electronic waste in the world is rapidly increasing, efficient methods that can handle this in an automated non-destructive fashion have not been developed. Screws are a common fastener used on electronic products such as laptops and must be removed during non-destructive methods. In this paper, we focus on using the cognitive architecture Soar to facilitate the disassembly sequence of removing these screws from the back of laptops. Soar is able to differentiate between different models of laptops and store the locations of screws for these models leading to an improvement of the disassembly time when the same laptop model is used. Currently, this work only uses one of Soar’s long-term memory modules (semantic memory) and a screwdriver tool. However, this work can be extended to use multiple tools by using different features available in Soar such as other long-term memory modules and sub-states.


I. INTRODUCTION

ELECTRONIC waste (e-waste) is a growing concern in the world as the number of electronics used in everyday life continues to increase while the lifespan of these electronics continues to shrink [1]. In developing countries, the most prevalent ways of discarding e-waste are through burning, burying, or dumping it in the sea [2]. In areas of developing countries such as Guiyu, China, where entire villages are heavily invested in the recycling of e-waste, recycling methods are crude and few preventative measures are taken to protect the workers from the hazardous materials they are disassembling. This is a concern since many of the heavy metals found in e-waste are toxic to both humans and the environment. Due to this, many studies have been conducted to measure the wellness of the people and the levels of heavy metals found in the region [3]–[5].

Methods used to disassemble e-waste are destructive, semi-destructive, and non-destructive disassembly. Destructive methods are used when the disassembly process aims to recover materials such as metals and plastics from the waste. Typically, the e-waste is fed into large shredders and then processed to retrieve different types of metals [6]. Semi-destructive methods of recovering e-waste involve cutting screws or wires in an attempt to disassemble the product, while non-destructive disassembly methods attempt to reverse engineer the assembly process. Non-destructive disassembly is an attractive option when the goal of the recycling operation is to be able to reuse the product or parts for different applications. Although manual non-destructive methods can yield the highest amount of recycled product, manual disassembly is time-consuming, and if proper working conditions are not met, potentially dangerous for the workers [7].

In order for a robot to make decisions based on an understanding of the parts and situations presented to it, some form of intelligence should be built into the robot’s control system. Bannat et al. [8] detailed the evolution of production, planning, and cognition systems and their growth in the future as they pertain to flexible, adaptive production systems for manufacturing. Stenmark et al. [9] presented a knowledge based method that leveraged cloud computing in order to extend the capabilities of a robotic system in a manufacturing setting. Kurup and Lebiere [10] showed that an algorithmic approach, where the user programs all possible scenarios that the robot can handle, is not efficient as the program will not be able to handle a scenario unless it was preprogrammed. Instead, an approach that uses cognition is preferred, as it corresponds more with the way that humans process and make decisions. Cognitive architectures make use of some form of high-level abstract reasoning to determine the next action to take and can incorporate different learning methods.

Popular cognitive architectures include Soar [11], ACT-R [12], and EPIC [13], [14]. Jones et al. [15] identified that ACT-R and Soar have constraints that are opposites of each
other in their respective low-level reasoning tasks. ACT-R only allows for one production rule to work at a time even if more rules are available. Therefore, multiple passes must be made in order to make decisions, increasing the computation time. Soar allows for multiple rule instantiations to fire at once. However, the computation time in Soar increases because only one type of operator rule is allowed to be selected at a time and the operators have to be proposed through different types of rules. Hanford [16] suggested that Soar is a better choice to use for robots since it does not limit the access to working memory and can use all knowledge to figure out how to proceed with a situation. Johnson [17] also compared the two architectures and stated the only similarity between the two is that they both organize control around a single-goal hierarchy. In conflict resolution, Soar tries to select an action based on all available knowledge, and if an impasse occurs, create a sub-goal to solve the problem while ACT-R will halt if it detects an impasse.

Laird and Rosenbloom [18] used a robot called Robo-Soar [19], which was a PUMA arm that used a vision system (VS) to obtain the orientation and position of objects in a workspace. The goal of Robo-Soar was to align these objects until a light came on and then press a button. Hanford et al. [20], [21] used Soar to control mobile robots. With these robots, Soar used different sensor information to decide how the robot should navigate to a GPS location while avoiding obstacles. One drawback in these robotic applications is that Soars long-term memory is not used, so when the agent is initiated it cannot draw from its previous experiences. A Soar agent named Rosie learned to play new games and perform other tasks using mixed-initiative interactions through natural language [22], [23], and visual demonstrations [24]. Rosie used semantic memory when learning to play a game to store all action knowledge, failure conditions, and goal states in order to determine a legal play. Current work [25]–[28] involving Soar addresses making decisions based on natural language processing. It is developed to work in conjunction with Rosie and uses grammatical rules in order to turn sentences into operations that Rosie can use in order to pick up or move objects. Mininger and Laird [29] developed a mobile robot that uses Soar to perform tasks and search for an object even if it cannot see the object it is looking for.

Guo et al. [30] presented a dual-objective optimization model to determine disassembly sequences by maximizing profits and minimizing time while Tian et al. [31] described an AND/OR disassembly sequence that considered uncertain component quality and disassembly operational cost. Friedrich et al. [32] showed an approach to path planning using algorithms that reduce planning time for robots performing different maintenance tasks. Li et al. [33] described an approach to identify the profitability of recycling electric vehicle electric components using a robotic system and semi-destructive disassembly methods. The robotic system was composed of a robot arm with six degrees of freedom, and had specially designed tools for drilling, cutting, and gripping. A semi-destructive automated disassembly plan was designed and carried out from a manually non-destructive disassembly step. Torres et al. [34] designed a system to disassemble computers where a VS was used to recognize parts presented to it. A modeling system could then virtually create a disassembly plan. One drawback of this approach was that it was necessary to have a detailed model of the part in order to create a representation of the disassembly sequence and make decision of what move to perform next. Vongbunyong et al. [35]–[37] described a system to perform semi-destructive disassembly of LCD TVs where a cognitive robotic agent was used. This cognitive agent used the language IndiGolog in order to determine the next disassembly step. A VS was used to obtain information about the product being disassembled. The system used learning and revision in order to eliminate redundant moves performed during the disassembly operation.

Schumacher and Jouaneh [38] created a disassembly tool that utilized low-cost force-sensing resistors (FSRs) and was able to remove snap-fits and batteries from calculators. This snap-fit removal tool was placed on an EnCore 2s robotic system that used the Microsoft Kinect and a Visual Basic program for control [39]. This system needed a database that contained the preprogrammed locations of the batteries for different orientations of the calculator. The Kinect was used to make the system more robust against minor variations when the device was loaded in the workspace. DiFilippo and Jouaneh [40] extended the system to work with a variety of laptops that the system had never seen before. A webcam with an overview of the entire workspace was used to determine potential locations of screws on the laptop, and a sensor-equipped (SE) screwdriver tool was created that would explore those locations and determine if a screw was actually present. The drawback of this system, was that the system did not remember the locations of the screws. Therefore, for every trial, every potential screw location needed to be checked which lead to long trial times (40-60 minutes).

These issues described above are very important when considering the use of robots to perform disassembly of end-of-life (EOL) electronic products. Unlike assembly operations on a production line, where product features and locations are known in advance, a robotic disassembly system for EOL products needs to work with a large variety of products of unknown sizes and possibly, damaged or missing features.

This paper will present a novel method combining computer vision, automated robotic disassembly, and the Soar cognitive architecture to be used for removal of screws from different laptops. It will show how Soar can be used to create a disassembly plan by obtaining all the needed information from the laptop in an automated manner. No information about any of the laptops were given by a human operator or to the system beforehand. This disassembly plan will be created by using Soar to automatically locate and remove screws from the back of laptops, which would be the first step in any non-destructive disassembly process. Soar will be able to maintain a database of the different laptops and remember the screw locations of each to reduce the time of the disassembly operation on subsequent trials.

The rest of this paper is organized as follows; the next section gives a brief introduction to Soar and how Soar works. It also discusses the procedure used to train Soar’s semantic memory with the different laptop models and holes. Section 3 discusses the results obtained during the trials, and Section 4
A. Soar Cognitive Architecture

Soar is a cognitive architecture developed by John Laird, Allen Newell, and Paul Rosenbloom at Carnegie Mellon University in 1983 [41], and works by using productions to test, propose, and select operators in working memory to try to obtain a specified goal state. Working memory, which can be represented by a graph structure, is short-term memory and the way which Soar views the current configuration of the workspace. Information is stored as working memory elements (WMEs) that consist of an identifier, attribute (preceded by a ‘∧’), and value. This value can either be a terminal node consisting of a constant (string or numerical value) or a non-terminal node which links to another identifier.

The decision cycle Soar goes through is shown in Fig. 1 and consists of an input phase, state elaboration/propose operators phase, select operators phase, apply operator phase, and output phase. While in the input phase, Soar checks to see if any new information has been placed on its input-link through sensors or external environments. The state elaboration/propose operators phase is when Soar elaborates the WMEs and proposes different operators. The operators are fired and retracted in parallel, and the firing of one operator could lead to the firing or retracting of other operators. Once there are no more rules left to fire, the system has reached a state of quiescence and proceeds to the select operators phase, where Soar uses rule preferences to select one operator. In the apply operator phase, Soar will execute the selected operator as well as make changes to working memory and in the output phase, put any appropriate information on the output-link. The information placed on the output-link can be used by an external environment.

While working memory represents short term memory, Soar also has long-term memory modules to improve its performance in executing tasks. The three types of long-term memory modules native to Soar are procedural, episodic, and semantic memory. Procedural memory uses a reward function and reinforcement learning to change how an operator is selected. Semantic memory is where previous knowledge about the workspace is stored, and episodic memory is used to remember past experiences as snapshots of the workspace. Soar moves through these snapshots to attempt to find the best match [42], [43]. In order to communicate with various programming languages, Soar uses the Soar Markup Language (SML) which easily allows information to be taken off and put on the input and output links.

B. Workspace Setup

The robot used is an EnCore 2S robot (MRSI, North Billerica, MA) that was interfaced with a Galil DMC-2413 four-axis digital motion controller. The robot was enclosed in a wooden frame that held panels of black fabric which allowed for ambient fluorescent light to be eliminated and a uniform lighting to illuminate the workspace. A Matlab R2014a client and Python 2.7 server were used for communication and communicate via TCP/IP protocols. The Python server was also used to communicate with Soar via the SML, while Matlab was used to communicate with the positioning, vision, and tooling systems. The linear actuators (Midwest Motion) used for clamping were connected to an Arduino Uno and Motorshield while the SE screwdriver was connected to a second Arduino Uno, Motorshield, and a custom shield designed to interface with the SE screwdriver’s accelerometer. A Microsoft Kinect was mounted above the workspace to make depth measurements and a Microsoft HD-3000 webcam was placed over the workspace to locate screw holes on the laptop. A second Microsoft HD-3000 webcam was mounted on the head of the robot and was used to make the fine movement adjustments needed to center a screw. A complete overview of the connections of the different systems can be seen in Fig 2.

II. METHODS

A. Laptop Selection and Orientation Using Soar

A clamping system composed of two linear actuators was used to push the laptop against the walls at the edge of the workspace. By utilizing the potentiometer feedback of the linear actuators, and calibrating its output from the distance to the wall to the edge of the clamp, it was possible to obtain the length and width of the laptop once it was clamped in the workspace. The average thickness of the laptop was determined using a Microsoft Kinect sensor located above the workspace. From previous work [44], the Kinect was placed approximately 720 mm above the workspace plate. This distance was as close as the Kinect was able to be mounted
in order to see the entire workspace and not interfere with the overhead camera used to obtain screw coordinates.

Using the Kinect, a background image of the workspace was taken with the linear actuators retracted. In Fig. 3, the region enclosed in the box is the region of interest (ROI) that the pixel distances were calculated. This ROI extends past the railing on the left and bottom sides to accommodate laptops taller than the railing. Five readings were taken at each pixel to minimize noise and averaged using Matlab. Once a laptop was placed in the workspace, another measurement was taken with the Kinect. The Kinect gives a measurement with respect to how far an object is from it, so the distance will be greater when there is no laptop in the workspace. Since the only change in the ROI is the laptop, these two measurements can be subtracted from each other on a pixel-by-pixel basis. Fig. 4 shows how the average thickness of a laptop was obtained. The average thickness was used because the cover of laptops can slant, resulting in a non-uniform thickness across the width of the laptop. In this algorithm, the thickness difference is the absolute value of the difference between the background and laptop thickness. This difference is only counted towards the total thickness if it is between 10 and 100 mm. If the difference is less than 10 mm, the value is disregarded as noise and if it is more than 100 mm, one of the pixels had a depth the Kinect was unable to determine.

The geometries of many laptops are not rectangular prisms and contain curved features and contours. It is possible that when a laptop is placed in the workspace at one orientation, different parts of the laptop may become the points of contact between the clamp and the wall than when it is placed in a different orientation. To account for this, the length, width, and thickness are measured for a laptop when it is in both a “0-degree” and “180-degree” orientation. The current workspace setup only allows the laptop to be placed in these orientations rather than four orientations every 90 degrees.

Soar’s semantic memory can be used to store different laptop models and their parameters. Soar was trained with the dimensions of eight different laptop models and used to determine what model was placed in the workspace, and its orientation. This was repeated 10 times for each laptop with a random run order; five times, the model was placed in a “0-degree” orientation, and the other five times the model was rotated and placed in a “180-degree” orientation. For each trial, the length, width, and thickness of the laptop was recorded as well as the successful query number, and laptop identifier assigned by Soar. Fig. 5 shows the data structure used to store each of the laptops attributes and how it can be extended to hold m-number of attributes. In Soar, the “@” sign symbolizes a long-term identifier (LTI) which indicates that the specified identifier resides in the long-term memory module.

The logic used by Soar to determine if a laptop is already in the database is shown in Fig. 6. The first step is to obtain the length, width, and thickness of the laptop from the linear actuators and Kinect. Next, a query is performed on the database and these dimensional values are compared to the laptops “0-degree attributes to determine if the laptop model is already in the database. To perform a query in Soar using semantic memory, the command <s>"smem.command.query.<attribute><value> needs to be used (In Soar, dot (.) notation is a short-hand way of connecting multiple attributes which only link identifiers). If <attribute> is “name”, then the command translates to “on the main state conduct a semantic memory query that matches any LTI with any name”. This command needs to be augmented with a math-query to restrict the attributes being compared to within an upper and lower bound. The command for a math-query is “math-query.<attribute><condition><value>” where <attribute> is the attribute that you are comparing and <condition> can either be greater-or-equal, less-or-equal among a few others which are found in the Soar manual [45]. Examples of these commands within the Soar syntax can be found in Fig. 7.
If the first query is successful, an image template matching algorithm must be performed as it is not possible to tell what orientation the laptop is in since it is possible both orientations could have the same dimensions. If the query is unsuccessful, then the database is queried again and compared to the laptops attributes in the “180-degree” orientation. If the second query is successful, the laptop is in the “180-degree” orientation. If the query fails, the laptop needs to be added to the database, so the clamps will retract and the laptop is rotated to obtain the dimensional parameters of the other orientation.

When a laptop is added to memory, a template image must be created for both orientations of the laptop. The steps taken to find and create the template image are shown in Fig. 8. Fig. 8a) shows the image of the the laptop in the workspace subtracted from the image of the workspace without the laptop. When the images are subtracted from each other, since the only difference in these images is the laptop, only the laptop is left and the rest of the image is black. Fig. 8b) shows the image after a Prewitt edge detection filter is performed, and Fig. 8c) shows the dilation of the lines. In Fig. 8d), the contours are closed over the laptops area, and the size and location of the laptop in the image is determined. Since the location of the laptop is now known, the laptops image can be extracted from the main image and saved as a template for both orientations of the laptop. These template file names are added to semantic memory as attributes of that laptop. Initially, an approach that used the same template image for both orientations was attempted. The template image would be rotated 180 degrees when performing the matching step to decrease trial time and database size. However, this approach did not work as the template image taken was sensitive to how the lighting hits the laptop and created enough discrepancies between the two orientations.

The template matching algorithm used is the sum of absolute difference (SAD) shown in (1) [46]. In this equation, \( I_1 \) is the template image, \( I_2 \) is the target image, and \( x \) and \( y \) are the size of the template image. In SAD, the template is moved over the target image, the pixel values are subtracted pixel-by-pixel, and the sum of these values give the score for the current region. The best match becomes the region with the lowest score.

\[
SAD(x, y) = \sum_{i=0}^{m_{rows}} \sum_{j=0}^{n_{cols}} |I_1(i, j) - I_2(x + i, y + j)| \tag{1}
\]

B. Laptop Screw Removal Using Soar

Soar’s semantic memory was used to evaluate how much the system’s performance would improve while removing screws from a laptop. Previously [40], every time the system ran a test, every circle location obtained from the overhead camera had to be explored to determine if a screw was present or not. Using semantic memory will cut down on the trial times as circles that have had their location explored will not have to be explored again. Soar will be able to ignore circles that do not have a screw in them, and only focus on the circles that have not yet been explored, or have had screws found.

To test how well Soar improves the system’s performance while removing screws from laptops, the disassembly program was run on a laptop until the testing time of a trial converged. The testing was performed using two methods. The first method allowed for circles to be identified by the overhead camera during every trial. The second method only allowed circles to be identified during the first training run, and the locations of the screws were retrieved from memory after the first trial. The eight laptop models were placed in the workspace a total of 10 times each. For five of the trials the laptop was placed in the “0-degree” configuration and
for the other five trials, the laptop was placed in the “180-degree” configuration. For all of these trials, circles were identified with the overhead camera, and the length of a test as well as the number of screws that were correctly determined and removed were recorded. An additional four trials were performed where Soar did not identify circles from the overhead camera and just removed screws by retrieving their locations from semantic memory.

Due to how semantic memory is constructed within the Soar framework, it is not possible to use dot notation to explore levels that are not directly under the LTI as it leads to increasingly complex queries. Instead, this work proposes that each hole can be treated as its own LTI and linked to a specific laptop by having an attribute that consists of the laptops ‘name value. By linking the LTIs of the laptop model and holes in the laptop via the attribute ‘name, when queried all subsequent queries will only select LTIs that pertain to the laptop in the workspace. This is shown in Fig. 9 where the laptops have LTIs of @L1 and @L2 and the holes have LTIs of @H1-@H5. The LTI @L1 has an attribute ‘name laptop1. When this laptop is chosen as the laptop in the workspace, only the holes that also have an attribute ‘name laptop1 will be returned. In Fig. 9, LTIs that are linked between the holes and laptops are visually shown with the same shade in their identifier.

It is easier to bring LTI into working memory to compare their contents than continually querying and comparing with semantic memory. The general logic flow used with Soar is shown in Fig. 10. When the program is started, the database will be checked for a laptop. While the details of checking for a laptop are shown in Fig. 6, the result is the laptop is already in the database, or is not and needs to be added. Next, the locations of the circles that were found by the overhead camera, as well as the current position of the end effector are put on the input-link and transferred to Soar’s working memory. Then all of the holes in Soar’s long term memory are queried and brought to working memory one-by-one until the query fails. When the query fails, it means that there are no more holes in semantic memory left to bring to working memory. If laptop is not already in the database, the failure will occur after the first attempt. Next, a circle from the input link is picked and its x,y coordinates are compared to the upper and lower bounds of all of the holes in memory. In case of multiple matches, preference is given to circles that are closer to the current position of the robot, and circles that contain a screw. Once a hole has been chosen there are three possibilities:

1) **The circle location is not already in the database and must be explored**- When a circle is explored, it uses the same procedure described in [40]. This previous work introduced a robotic system that consisted of two different cameras; an overhead camera that identified potential locations of screws, and a camera mounted on the robot that would travel to each of the potential hole locations and attempt to center the screw hole. Once a hole was centered, the SE screwdriver would move to that spot and try to remove the screw. The work explored varying different parameters (camera brightness, lighting method, and laptop cases) in order to accurately locate screw holes. Darker laptop cases required a higher camera brightness to create a high contrast with the
hole making it easier to find using the image segmentation algorithm. A localized lighting source underneath a camera that moved with the head of the robot was also desirable as it would eliminate shadows on the laptop caused by lighting the area from above.

The VS uses image segmentation in order to isolate a screw hole by systematically varying the standard deviation (std) of the 5x5 Gaussian blur and the sensitivity of the Prewitt edge detection. The above procedure was modified with minor variations in order to speed up the test segments such as the size of the image that the algorithm is performed on. Before the initial movement adjustment to center the screw, the image used to center a screw is smaller (350 x 400 px compared to 600 x 800 px). If the robot makes a movement adjustment, the image becomes smaller (300 x 300 px) as the screw should be close to the center of the webcam. Since the image is smaller, the algorithm no longer attempts to close contours, but does toggle the localized lighting module; First by checking the hole with the light on and then with the light off. The different lighting methods combined with both the laptop color and the depth of the screw holes can result in higher contrast levels. If a screw is removed, the program remembers the computer vision parameters used to find the hole and the coordinates at which the hole is at. The program also transposes the coordinates of the hole so it can be found on the laptop’s other orientation. After a screw has been centered, the SE screwdriver is used to check the area for a screw. Attributes are added to the LTI of the hole to save the information determined by the computer vision, robot, and screwdriver and is placed in Soar’s semantic memory. If a screw was not found, this circle location is placed in memory as not containing a screw.

2) The circle is already in the database and contains no screw or is not a screw hole- When the circle from the input-link matches the circle in working memory and there is no screw, the robot is told to wait and Soar proceeds to its next decision cycle.

3) The circle is already in the database and contains a screw- When the circle from the input-link matches a circle in working memory and there is a screw at that location, the robot is moved to the x,y location and attempts to center, and finally removes the screw. In order to speed up the computer vision while it is locating and centering a screw, the ROI is reduced (300 x 300 px) and the computer vision parameters used are retrieved from the holes semantic memory attributes. When Soar makes a decision to explore the circle location with the SE screwdriver, it is assigning a reward based on the presence of a screw. This current model assumes that a score of a 1.0 indicates a screw is present, a 0.0 indicates a screw is not present, and each hole starts with a score of 0.5. If a screw is not found, that location is marked with a no when the LTI is modified (-0.5) and if a screw is found, that location is marked with a yes (+0.5). However, this current model is greedy and means that if a hole was incorrectly found or not found, it will be placed in memory as having a screw or not having a screw and will not change.

III. RESULTS AND DISCUSSION

A. Laptop Thickness

The average thicknesses of the laptops, as calculated by the Kinect, are shown in Fig. 11. The triangle that is pointed upward shows the maximum thickness measured on the laptop and the triangle that is pointed downward shows the minimum thickness measured on the laptop. The average thickness computed by the Kinect is shown as a circle with error bars indicating one std. The results show the Kinect is able to compute an average thickness that falls between the minimum and maximum thickness. The measurements taken by the Kinect are also repeatable as the highest standard deviation was 0.324 mm for Laptop 3. Laptop 4 was equipped with an extended life battery which is why the maximum thickness is so high. The average thickness of the laptop is much closer to the minimum thickness of the laptop because the extended battery only covered a small area on the back of the laptop.

The results when Soar was used to determine the orientation and model of a laptop are shown in Table I. The “Orientation” column identifies whether the laptop was placed in a “0-degree” or a “180-degree” orientation. The “Query” column displays whether the first or second query was the successful query when the laptop was compared in semantic memory. If query 1 was the successful query, the template matching procedure had to be implemented since the program would be unable to determine what orientation the laptop was in. The highest std of length, width, and thickness is 0.321 mm thus, four stds is 1.3 mm. This can be seen with laptop models 3, 7, and 8, as all the size parameters are less than 1.3 mm of the corresponding size parameter in the other orientation. For all of the other laptop models that were determined by query 1 when the laptop was placed in the “0-degree” orientation, and query 2 when the laptop was placed in the “180-degree” configuration, at least one of their size parameters was higher than 1.3 mm of the corresponding size parameter in the other orientation. However, if query 2 was the successful query, then by the process of elimination, the laptop was in the “180-degree” configuration. In this experiment, Soar was able to determine the correct laptop and orientation 80 out of 80 times. The “Length”, “Width”, and “Thickness” columns are the experimental results obtained from the two linear actuators, and the Kinect. The ± values in these columns represent one std of the measured quantity. All measurements have low stds demonstrating that the measurement methods are repeatable.

B. Holes

The number of trials it takes for the time of a disassembly routine to converge is shown in Fig. ?? The first disassembly trial for a laptop always takes the longest time because all the circles located by the overhead camera need to be explored. For all the different laptops, the time has decreased by 60% or more by the second trial. The system performs better when the circles are only explored during the initial trial rather than every trial. With this method, the system reaches its best performance time during the second trial and the test times stay consistent throughout the rest of the trials. If the circles are located and explored every trial, then a comparable trial
The visual results of the circles that the system explored for the 2nd run shown in Fig. ?? with the ‘○’ data marker. In Fig. 13, a ‘▲’ means the VS was unable to determine a screw, a ’X’ means the SE screwdriver was unable to find and remove a screw, and a ’●’ means that the screw was successfully removed. As the number of trials progress, the number of circles that are explored continue to decrease until none or only a few circles are explored during a trial.

The results showing correct number of holes found by the overhead camera and number of screws that were successfully removed by the SE screwdriver are shown in Table II. These results are an improvement over the previous findings [40] as the parameters governing the Hough circle transform such as circle radius size were relaxed. The results are separated by the orientation the laptop was placed in the workspace. The column “New Circles Explored”, lists the number of new circles that were found and explored with the robot camera and SE screwdriver during a particular trial. As more trials were performed for each laptop, the number of holes needed to be explored decreased, until a trial would only need to explore a few holes. This decrease was due to the fact screws that had been previously found and removed were saved in semantic memory and still removed without the need to be explored. The column “Possible Screws to Remove” lists how many screws could be found on the laptop during that trial after adjusting for if a screw had been missed and that location incorrectly been placed in memory as not containing a screw. The column “Holes Found By Vision” lists the number of the screw holes successfully found by the VS. Finally, the column “Screws Removed” indicates the number of screws found by the computer system that were successfully removed.

Italicized entries indicate that a screw on the laptop was removed by the SE screwdriver are shown in Table II. These results are an improvement over the previous findings [40] as the parameters governing the Hough circle transform such as circle radius size were relaxed. The results are separated by the orientation the laptop was placed in the workspace. The column “New Circles Explored”, lists the number of new circles that were found and explored with the robot camera and SE screwdriver during a particular trial. As more trials were performed for each laptop, the number of holes needed to be explored decreased, until a trial would only need to explore a few holes. This decrease was due to the fact screws that had been previously found and removed were saved in semantic memory and still removed without the need to be explored. The column “Possible Screws to Remove” lists how many screws could be found on the laptop during that trial after adjusting for if a screw had been missed and that location incorrectly been placed in memory as not containing a screw. The column “Holes Found By Vision” lists the number of the screw holes successfully found by the VS. Finally, the column “Screws Removed” indicates the number of screws found by the computer system that were successfully removed.

Italicized entries indicate that a screw on the laptop was

<table>
<thead>
<tr>
<th>Laptop</th>
<th>Orientation (Degree)</th>
<th>Query</th>
<th>Soar Identifier</th>
<th>Success Rate (%)</th>
<th>Length (mm)</th>
<th>Width (mm)</th>
<th>Thickness (mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>@L14</td>
<td>100</td>
<td>307.48 ± 0.012</td>
<td>266.11 ± 0.036</td>
<td>43.06 ± 0.157</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>2</td>
<td>@L14</td>
<td>100</td>
<td>307.32 ± 0.019</td>
<td>261.13 ± 0.228</td>
<td>43.46 ± 0.045</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>@L28</td>
<td>100</td>
<td>332.41 ± 0.010</td>
<td>274.11 ± 0.006</td>
<td>31.60 ± 0.047</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>2</td>
<td>@L28</td>
<td>100</td>
<td>332.43 ± 0.038</td>
<td>270.23 ± 0.321</td>
<td>31.87 ± 0.094</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>@L42</td>
<td>100</td>
<td>354.80 ± 0.010</td>
<td>254.12 ± 0.012</td>
<td>34.04 ± 0.137</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>1</td>
<td>@L42</td>
<td>100</td>
<td>355.16 ± 0.010</td>
<td>253.86 ± 0.063</td>
<td>33.42 ± 0.027</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>@L56</td>
<td>100</td>
<td>343.39 ± 0.022</td>
<td>279.80 ± 0.207</td>
<td>35.24 ± 0.010</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>2</td>
<td>@L56</td>
<td>100</td>
<td>343.26 ± 0.004</td>
<td>283.38 ± 0.022</td>
<td>35.32 ± 0.009</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>@L70</td>
<td>100</td>
<td>290.72 ± 0.086</td>
<td>242.51 ± 0.016</td>
<td>30.06 ± 0.154</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>2</td>
<td>@L70</td>
<td>100</td>
<td>289.93 ± 0.128</td>
<td>245.16 ± 0.021</td>
<td>30.32 ± 0.041</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>@L84</td>
<td>100</td>
<td>359.49 ± 0.074</td>
<td>264.43 ± 0.031</td>
<td>35.45 ± 0.032</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>2</td>
<td>@L84</td>
<td>100</td>
<td>359.07 ± 0.020</td>
<td>261.75 ± 0.013</td>
<td>35.52 ± 0.139</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>@L98</td>
<td>100</td>
<td>292.65 ± 0.124</td>
<td>234.74 ± 0.021</td>
<td>32.15 ± 0.196</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>1</td>
<td>@L98</td>
<td>100</td>
<td>292.86 ± 0.006</td>
<td>234.91 ± 0.007</td>
<td>32.11 ± 0.222</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>@L112</td>
<td>100</td>
<td>356.11 ± 0.024</td>
<td>262.36 ± 0.022</td>
<td>34.69 ± 0.221</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>1</td>
<td>@L112</td>
<td>100</td>
<td>356.41 ± 0.062</td>
<td>262.04 ± 0.011</td>
<td>35.07 ± 0.171</td>
</tr>
</tbody>
</table>

Fig. 11. Average, minimum and maximum thickness of a laptop

Fig. 12. Testing time to remove screws from a laptop.
Fig. 13. Visual results for the convergence test for laptop model 1. The numbers over the laptop indicate the corresponding trial number in Fig. ?? (2nd run). The ‘•’ indicates that a screw was found and successfully removed, a ‘X’ indicates the screwdriver did not find a screw, and the ‘▲’ indicates that the computer vision did not find a screw.

missed the first time a screw hole was discovered either by a) the VS not being able to determine the hole or b) the SE screwdriver not being able to remove the screw. The number in the parenthesis denotes the number of screws missed and incorrectly placed in semantic memory. The column the parenthesis are in indicates whether the VS (“Holes Found By Vision”) or the SE screwdriver (“Screws Removed”) made the error. This is also reflected in the “Possible screws to remove” column. For example, looking at laptop 2, there are 14 circles that contain screws and during the first trial, two of these holes were not found by the VS and placed in memory as not containing a screw. During the next trial, since those two circles did not contain a screw, there were only 12 possible screws that the system would be able to remove. Adding up the errors in both orientations, six errors occurred via the VS where the robot camera was unable to find the screw and six errors occurred via the SE screwdriver. Of the six errors with the SE screwdriver, four occurred because the screwdriver was not able to mate with the screw due to a physical obstruction caused by a feature on the laptop or clamp.

Due to the setup of the tools on the robot, the entire laptop cannot be reached by the robot when it is placed in the workspace. The reachable region of the workspace is shown as the rectangle that is on the laptops in Fig. 13. This means that the entire laptop is not explored when the laptop is introduced in its initial “180-degree” orientation resulting in a higher number of circles that need to be explored during the first trial for both orientations of the laptop.

The results of the tests broken up by different testing time categories are shown in Fig. 14 and Fig. 15. The time categories are: parameter time, computer vision time, robot movement time, screwdriver removal time, and other time. The parameter time for these tests was defined as the amount of time it takes the system to clamp the laptop in the workspace, take dimensional measurements, and find circles with the overhead camera. The parameter time always took the most time when a laptop was placed in the workspace and was not in memory. When a laptop needed to be added to memory, the laptop would then have to be unclamped, rotated, and re-clamped. These trials are always the first trial when the laptop was placed in the “180-degree” orientation. When a laptop was placed in the “0-degree” orientation and a test was run the parameter time is always less, even during the first trial, because the laptop had already been added to memory.

The robot movement time was the time it took the robot to move from circle to circle and the computer vision time was the amount of time that that system spent identifying and centering screw holes. Both the robot movement time and the computer vision time were the greatest in the first trial because the system had to explore every hole. The correlation between these two time variables has an r-score $>0.97$ for every laptop model indicating a strong positive correlation. The screw removal time was the time the disassembly sequence was actively trying to remove the screw with the SE screwdriver. The other time was the amount of time waiting for Soar to determine the next hole to remove, or waiting for a response between Matlab and the hardware during a communication timeout. When the VS identified a large cluster of holes, or when there were a large number of holes in the database, the computational time Soar required to make a hole selection increased.

Fig. 14 shows the results for when circles were found and explored every trial, and Fig. 15 shows the results when the circles were only explored during the initial trial. It should
be noted that Fig. 15 only has four trials instead of five trials since this figure reports the results for “Method 2” and the fastest time was reached by the second trial. Fig. 15 shows improvement over the trials in Fig. 14 as the time of a trial is faster and more consistent. The computer vision time for this figure, after the first trial, is only the amount of time that the system spends trying to center a screw while the computer vision time for Fig. 14 is spent both exploring and centering screws. The results in Table III show the performance between the two methods used to remove the screws from laptops. In this table the method where circles are located during every trial is referred to as “Method 1” and the method where circles are only found during the first trial is referred to as “Method 2”. For every laptop model, finding circles with “Method 2” decreased the overall time of a trial, and for 15 out of 16 laptops and orientations decreased the trial run time by over 10%. The column “Difference” shows that that for some laptops, it is possible for this to save over a minute of testing time during a trial. The average time that was reported for each orientation and method was calculated by averaging the time of the trials with less than five new circles explored from Table II.

### Table II

<table>
<thead>
<tr>
<th>Laptop</th>
<th>180-degree Orientation</th>
<th>0-degree Orientation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>New Circles Explored</td>
<td>Possible Screws to Remove</td>
</tr>
<tr>
<td>1</td>
<td>1 46 6 6 6 6</td>
<td>2 11 7 7 7 7</td>
</tr>
<tr>
<td>2</td>
<td>1 46 6 6 6 6</td>
<td>2 11 7 7 7 7</td>
</tr>
<tr>
<td>3</td>
<td>0 6 6 6 6 6</td>
<td>4 2 7 7 7 7</td>
</tr>
<tr>
<td>4</td>
<td>0 6 6 6 6 6</td>
<td>4 2 7 7 7 7</td>
</tr>
<tr>
<td>5</td>
<td>0 6 6 6 6 6</td>
<td>4 2 7 7 7 7</td>
</tr>
</tbody>
</table>

**Legend:**
- **Laptop:** The laptop model being tested.
- **Trial:** The trial number.
- **New Circles Explored:** The number of new circles explored.
- **Possible Screws to Remove:** The number of possible screws to remove.
- **Holes Found by Vision:** The number of holes found by vision.
- **Screws Removed:** The number of screws removed.

Note: The table shows results for testing laptops in two orientations: 180-degree and 0-degree, and compares the performance of two methods: "Method 1" and "Method 2". The results indicate that "Method 2" generally outperforms "Method 1" in terms of reducing the time required to complete the task.
TABLE III
RESULTS SHOWING IMPROVEMENT FROM WHEN CIRCLES WERE FOUND AND EXPLORED DURING EVERY TRIAL (METHOD 1) AND WHEN CIRCLES WERE ONLY FOUND DURING 1ST TRIAL (METHOD 2).

<table>
<thead>
<tr>
<th>Laptop</th>
<th>180-degree Orientation</th>
<th>0-degree Orientation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method 1</td>
<td>Method 2</td>
</tr>
<tr>
<td>1</td>
<td>208.2 ± 23.2</td>
<td>164.3 ± 5.8</td>
</tr>
<tr>
<td>2</td>
<td>377.3 ± 37.8</td>
<td>289.3 ± 5.1</td>
</tr>
<tr>
<td>3</td>
<td>554.6 ± 77.0</td>
<td>297.0 ± 4.5</td>
</tr>
<tr>
<td>4</td>
<td>361.1 ± 19.7</td>
<td>346.3 ± 8.6</td>
</tr>
<tr>
<td>5</td>
<td>328.7 ± 15.9</td>
<td>230.9 ± 4.5</td>
</tr>
<tr>
<td>6</td>
<td>414.9 ± 26.8</td>
<td>275.6 ± 14.6</td>
</tr>
<tr>
<td>7</td>
<td>276.3 ± 3.9</td>
<td>216.6 ± 5.3</td>
</tr>
<tr>
<td>8</td>
<td>402.3 ± 62.0</td>
<td>361.0 ± 2.5</td>
</tr>
</tbody>
</table>

Using the results of trials where the circles were only explored for the first trial, data revealing the best run conditions could be calculated and is displayed in Table IV. All of this data is broken up by the orientation that the laptop was placed in. The column labeled “Average Screw Removal Time” is the average disassembly trial time for three runs after the training run. The “Screw Removal Time Limit” is the fastest time that the laptop could be disassembled while being adjusted for the screws that the system missed. In order to calculate this limit, the average parameter time was used for each model of laptop. This accounts for the fact that laptops could have different dimensions and so the clamping time could vary. The robot movement time was calculated by measuring the distance between holes that were closest to each other on the back of the laptop, for all of the holes in the reachable workspace. Additional distances that the laptop moved such as switching from the robot camera to the SE screwdriver as well as moving up and down were also added in to the total distance. This total distance was divided by the speed the robot was traveling at to obtain the optimal robot movement time. The fastest computer vision time that was produced by a trial was 6.5 seconds per hole. This time was used for all laptops and multiplied by the number of screws to get an optimal computer vision time. Finally, for every laptop, the fastest screwdriver removal time was used which was between 7.0 and 8.2 seconds. The screw removal times for individual laptops were used because the screws used on the different laptops could vary in type and size.

The backs of different laptop models have different number of screws so it is tough to compare the total disassembly times of laptop models. If the disassembly time is normalized to a “per screw” basis, it is possible to compare how well the system was able to remove the screws across different laptop models. The column “Screwdriver Removal Time Per Screw” shows the average time the SE screwdriver was trying to remove an individual screw per test, and for all of the tests, this time was between 7.0 and 9.6 seconds per screw. The column “Average Screw Removal Time Per Screw”, takes into account the robot movement and computer vision time as well as the screw removal time. These time categories are added together and divided by the number of screws in the reachable workspace. This means the average time to travel to, center, and remove a screw takes between 22.8 and 31.0 seconds per screw for the different laptops.

IV. CONCLUSION
This paper presented the results of incorporating the cognitive architecture Soar to an automated robotic cell capable
of determining the locations of screw holes on a laptop. The locations, and number of screw holes were not preprogrammed and were discovered by combining force sensing from a SE screwdriver and vision sensing from multiple webcams. Soar’s semantic memory module was used in order to remember the location of circles that contained screws and the locations that did not. These results were verified experimentally by using Soar to facilitate the screw removal process for eight different laptop models. The different laptops and potential screw holes were all given their own LTI and linked together by a laptop name identifier value.

The system presented in this paper performs the fastest times when the trials for exploring circles are limited to one training run (Method 2). While exploring circles in every trial (Method 1) gives the system the greatest chance to find a screw that may have been missed, it results in longer run times. The number of trials that the system performs while exploring screws can be adjusted so there is a tradeoff between the fastest trial times and its accuracy in finding screws. The reward factor the system currently uses is greedy and makes a decision if a screw is in a location after only one trial by giving a reward factor of +0.5 if a screw is found or -0.5 if
a screw is not found. These reward scores could be modified to change the system's behavior during training. The lower the reward, the better chance of eliminating putting an incorrect result in memory since the number of trials before a location was saved in semantic memory as containing a screw or not, which would improve the confidence all the screws were been found. Soar is a good cognitive system to choose for work like this as it:

1) Has a good framework (SML) in place to easily accept input from sensors and different languages and give output commands as well as allowing for easy incorporation in various programming languages.

2) Has long term memory modules well suited for disassembly applications. The semantic memory, is well suited for remembering information about specific LTIs such as laptop models and holes. One limitation with semantic memory is if the database to be queried starts to grow large (around 150 holes), Soar will begin to take more time (seconds instead of milliseconds) to make a decision about what hole is next due to all of the proposal and elaboration rules between the different hole identifiers. This delay only occurs when Soar looks for new circles during every trial (Method 1) and does not occur when Soar is only trained during the first trial as only the holes that have screws are proposed (Method 2). In the future, other long term memory modules, such as episodic memory could be leveraged to remember the unique disassembly steps and order for the different models of laptops.

3) Utilizes subgoal/substates to expand the search state. Although subgoal/substates were not used in this research, these allow Soar to continue to search for a solution if an impasse is found. These substates could be used to control the different systems connected to the robot, not just selecting the next hole to remove a screw from. For example, different disassembly tooling would use substates to help organize and manage the different goals and operators that Soar would be able to choose from.

In conclusion, Soar in conjunction with its long term semantic memory module, helps with the screw removal process as it was able to remember locations of the screws and decrease a trial time by over 60% for the laptop models. This work can be extended to the creation of multiple end effector tools to further expand the disassembly sequence as well as modifying parameters on the vision system to work with laptops in a variety of conditions. In addition the expansion of the type of Soar’s long term memory modules will allow for different and unique disassembly plans to be constructed and followed for various models of a laptop.

References


---

**Nicholas M. DiFilippo** received his MS degree in mechanical engineering from the University of Rhode Island, USA in Summer 2012 and his Ph.D. degree in mechanical engineering in Fall 2016 at the same institute. His research interests include vision systems, robot motion planning, mechatronics, and perception in robotics.

---

**Musa K. Jouaneh** (M, SM) is Professor of Mechanical Engineering and the director of the Mechatronics laboratory at the University of Rhode Island (URI). He received his B.S. degree in Mechanical Engineering in 1984 from the University of Louisiana at Lafayette, and his Master and Doctorate degrees in Mechanical Engineering from the University of California at Berkeley in 1986 and 1989 respectively. His research interests include mechatronics and robotics with particular interest in motion control systems. He has served as a consultant to many companies in the Northeast and has received two College of Engineering Faculty Excellence Awards and the URI Foundation Teaching Excellence Award. He is the author or co-author of over 65 publications including two US patents. He is also the author of two published textbooks on Mechatronics. Dr. Jouaneh is a Fellow member of ASME, and a member of ASEE.