Kollisionsfreie Mobilität

Ausarbeitung

Fabian Bruhns, Jan-Henrik Bruhn, Sven Mehlhop

26. Februar 2018

Inhaltsverzeichnis

1 Einleitung

Im Rahmen der Vorlesung **Formale Methoden eingebetteter Systeme: Modellbasierte Analyse** soll ein Semesterprojekt bearbeitet werden. Dieses Dokument ist die Ausarbeitung zu der ersten Wahlmöglichkeit zwischen zwei verschiedenen Projekten.

Ziel dieses Projekts ist es, einen Controller zu entwickeln, der einen Roboter so steuert, dass er eine Folge von Zielfeldern irgendwann erreicht und niemals einem auf dem Spielfeld befindlichen Kind zu nahe kommt. Dieser Controller soll auf Basis von spieltheoretischen Methoden synthetisiert werden wodurch dieses Projekt eine Echtweltanwendung in einem Eingebetteten System simuliert.

Als Erschwernis kommt weiterhin eine Tankladung hinzu, die mit jedem Schritt reduziert wird und an Tankstellen beladen werden kann. Zusätzlich soll der Roboter die Ziele in einer Endlosschleife durchlaufen. Dabei haben sowohl das Kind als auch der Roboter feste mögliche Bewegungsmuster auf dem Feld um ihre Aktionen zu verwirklichen. Wie dieses Problem für den Roboter gelöst wird, ist in dieser Ausarbeitung beschrieben.

2 System

2.1 Umgebungsvariablen

Roboter vs Kind Bewegungsmuster Spielfeld um Begrenzung und Mauern erweitert

2.2 GUI

Die Benutzeroberfläche zur Visualisierung des Problems und dem anschließenden Lösungsweg, wurde so gewählt das der Nutzer zu jedem Zeitpunkt die Schritte das Programmes nachvollziehen kann. Dazu wurde die GUI von der Logik getrennt und besteht als eigenständiger Teil im Projekt. Die GUI bezieht von einem Spielfeld wo sich die Objekte befinden. Dieses Spielfeld wird von der Prozesslogik verändert und von jener ausgelesen. Nach jeder Veränderung wird das Feld als Benutzeroberfläche neu gezeichnet.
Dem Benutzer ist es zu aller erst möglich das Spielfeld durch das Platzieren von Objekten zu gestalten. Dabei stehen dem Benutzer Wände, Roboter, Kind, Ladestationen und Zielfelder zur Verfügung. Der Roboter beschreibt das Feld auf dem der Roboter auf dem gekachelten Spielfeld startet. Dem entsprechend verhält sich das Kind zu seiner Figur. Die Batterien stellen Ladestationen für den Roboter da, welche er in einem Spielmodus besuchen muss, um seine Energie/Ladung wieder aufzufüllen. Die Zielfelder beschreiben die Felder die der Roboter in chronologischer Reihenfolge abarbeiten muss. Dabei wird das als nächstes angestrebte Feld grün markiert.
Ist das Spielfeld erstellt kann der Benutzer im folgenden die Bewegungsmöglichkeiten von Kind und Roboter frei wählen. Dabei sind auch Variationen möglich mit welchen der Roboter sein Ziel nicht erreichen kann, da die GUI mit Rückmeldungen an den Benutzer ausgestattet ist, welche dieses zur Laufzeit wiedergeben.
Um das Programm zu starten ist der Start-Button zu drücken. Dieser ermöglicht es dem Benutzer nach dem Laden der Logik das Programm zu Beschleunigen oder gänzlich

zu stoppen. Für das Beschleunigen steht ein Regler zur Verfügung, welcher die Dauer zwischen den Schritten beschränkt oder verlängert. Das Stoppen wird durch einen gleichnamigen Button realisiert, welcher das Programm wieder in den Bearbeitungszustand setzt.
In den Menüreitern können weitere Einstellungen getroffen werden, wie Sprache, oder ob die Tankfüllung beachtet werden soll. Im Programm werden dann dynamisch die Sprache angepasst und die Tankfüllung angezeigt (oben rechts im Feld).
Sollten Fragen zum Spiel aufkommen können diese jederzeit mit dem Hilfefenster versucht geklärt zu werden. Weiterhin ist es noch möglich die Spielfeldgröße zu verändern.

3 Logik

probleme übersicht roboter probleme übersicht gruppe

3.1 Spielgraph

To review!
Da der Roboter seine Entscheidungen basierend auf den Enforce-Algorithmen treffen soll, müssen wir einen endlichen Spielgraphen erstellen. Ein solcher Graph besteht aus Zuständen, in denen der Spieler (in diesem Fall der Roboter) und Zuständen in denen die Umgebung (in diesem Fall das Kind) einen Zug macht. Ein Zug wird dabei von einer Kante im Graphen dargestellt.

In unserem Fall sind die möglichen Züge die jeweiligen Bewegungsmuster, die den Spielern mitgegeben werden. Demnach besteht unser Spielgraph $G = (Q, I, Q_0, Q_1, A_0, A_1, \Omega)$, orientiert an den Vorlesungsunterlagen, aus diesen Elementen:

- Q: Menge aller Zustände
- $I \subseteq Q_0$: Menge aller Startzustände
- Q_0 und Q_1: Teilmengen von Q
  - Q_0: Zustände in denen der Roboter am Zug ist
  - Q_1: Zustände in denen das Kind am Zug ist
- A_0: Züge des Roboters
- A_1: Züge des Kindes
- $\Omega \subseteq Q$: Zustände, in denen der Roboter auf dem Ziel steht

Die Zustände enthalten dabei die Positionsinformation des Roboters und des Kindes, und welcher Spieler am Zug ist.

Um einen Spielgraphen für unser Spielfeld zu erhalten generieren wir also, ausgehend von einem Initialzustand $Q0$, alle möglichen darauf folgenden Transitionen, abhängig davon, ob der daraus entstehende Folgezustand valide ist. Es werden dabei jeweils entweder

die Bewegungesmuster des Kindes oder die des Roboters gewählt. Die daraus folgenden Zustände werden in die Datenstruktur mit aufgenommen. Wenn jene zuvor noch nicht bekannt waren, werden auch von diesen Zuständen die Folgezustände generiert, bis keine neuen Zustände mehr generiert werden.

Da schon der Spielgraph illegale Zustände vermeiden muss, wird hier bereits beachtet, ob der Roboter mit dem Kind kollidiert oder nicht. Die offensichtliche, einfache Lösung, unabhängig davon wer am Zug ist illegale Folgezustände zu verbieten entspräche natürlich nicht der Aufgabenstellung, weil dann auch das Kind dem Roboter aktiv ausweichen würde. Es soll aber der Roboter dem Kind ausweichen, also ggf. auch vorhersehen ob er mit dem Kind kollidiert. Diese Funktionalität wird umgesetzt, indem für alle generierten Folgezustände des Roboters zusätzlich die darauf folgenden Möglichen Züge des Kindes berechnet werden. Enthält diese zweite Menge einen Zustand, in dem das Kind in den Roboter hineinspringt, wird der aktuelle Roboterzustand verworfen, da er zu einem illegalen Zug führen kann.

Wird die Graphengenerierung so umgesetzt erhalten wir bereits einen Roboter, der dem Kind aktiv ausweichen kann, egal welchen Zug das Kind macht. Dies gilt natürlich nur unter der Voraussetzung, dass die angegebenen Bewegungsmuster dies Zulassen. Kann das Kind bspw. sich frei bewegen ($A_1 = u, d, l, r$), der Roboter jedoch gar nicht ($A_0 = e$), so kann der Roboter offensichtlich auch keine Kollision vermeiden.

3.2 Enforce

Auf dem Graphen wird der Enforce-Wert für die Bewegungen des Roboters und des Kindes berechnet. Dabei treten die Aktionen der beiden Beteiligten immer in abwechselnder Reihenfolge ein. Zu erst kommt immer die Bewegung des Roboters, dann die des Kindes. Dieses wiederholt sich, bis der Roboter sein Ziel erreicht hat, oder dieses nicht mehr erreichbar ist, durch einen Fehlerzustand. Letzterer ist zu vermeiden und durch den Enforce-Wert möglichst zu vermeiden.
Für die Berechnung wird in zwei verschiedene Algorithmen unterschieden, zum einen Enforce und zum Anderen Enforce+ . Die beiden Algorithmen und ihre Umsetzung werden im folgenden diskutiert.
Der Enforce Algorithmus selbst ist das Grundgerüst, welches zu erst aufgebaut wird. Dabei wird vom Zielpunkt ausgegangen, denn der Roboter erreichen soll und Rückwärts der Enforce-Wert für die Knotenpunkte berechnet. Dabei erhält der Zielpunkt selbst denn Wert '0'. Von hier an wird eine Wiederholung eingeleitet die durchgeführt wird, bis der Startzustand erreicht ist, oder als nicht erreichbar gilt.
In der Wiederholung wird zuerst der Enforce Wert um einen hoch gezählt und alle Kanten die in einen Knoten gehen den man im vorherigen Schritt betrachtet hat zurückverfolgt. Dabei stößt man auf zwei Arten von Knoten, es wird zwischen einem Knoten unterschieden in dem der Roboter am Zug ist und Einem in dem das Kind am Zug ist. Dieses bildet die klassische zwei Spieler Partie ab.
Im Falle des Roboter-Knotenpunktes wird geschaut ob dieser schon einen Enforce-Wert hat, sollte dieses nicht so sein, wird der jetzige Wert eingetragen. Dieses bedeutet, da der Roboter immer den bestmöglichen Weg nimmt, dass es entweder schon einen vorhande-

nen besseren Weg gibt, oder jetzt einer gefunden wurde der zum Ziel führt in maximal Enforce-Wert Schritten. Dieses spiegelt sich in der Existenz eines möglichen Weges wieder, der für den Roboter wichtig ist.
Solch ein Verhalten ist bei den Knotenpunkten des Kindes nicht zu erwarten. Das Kind stellt die Umgebung da, welche bei jeder möglichen Bewegung dennoch den Roboter nicht vom erreichen des Ziels abhalten soll. Die Umgebung soll mit ihren Möglichkeiten den Roboter möglichst stark einschränken oder gänzlich verhindern, wobei der Roboter dennoch einen Weg finden soll. Dieses zeigt auf, dass wenn ein Knotenpunkt gefunden wird, jeder Nachfolger dieses Knoten schon einen Enforce-Wert haben muss, damit die Umgebung keine schlechte Alternative treffen kann. Somit ist bei einem betrachten eines solchen Knotenpunktes es möglich, dass entweder alle anderen Nachfolger schon betrachtet worden sind, dann wird der jetzige Enforce-Wert eingetragen, oder es sind nicht alle betrachtet worden, folglich wird kein Wert eingetragen. Es kann nicht sein, dass schon ein Wert vorhanden ist, da sonst der vorherige Knoten schon betrachtet worden wäre, welches nicht möglich ist. Durch diese Unterscheidung liegt in diesem Fall der Allquantor vor. Jede Aktion die die Umgebung wählen kann, muss dennoch zu einem Enforce-Wert führen. Dieses folgert, dass das Kind im schlimmsten Fall das Gewinnen des Roboters nur maximal herauszögern kann, jedoch nie gänzlich verhindern.
Damit erfüllt der Enforce Algorithmus die Aufgabenstellung, dass der Roboter zu einem Zeitpunkt in der Zukunft das Ziel erreichen wird. Jedoch ist noch nicht gegeben, dass er dieses auch mehrfach erreichen kann. Für diesen Teil ist der Enforce+ Algorithmus zuständig. Dieser erweitert den Enforce Algorithmus indem er nicht beim erreichen des Startknotens stoppt, sondern darüber hinaus noch das Zielfeld vom Zielfeld selbst. Dieses meint, dass es mindestens einen Zyklus geben muss vom Zielfeld zu sich selbst, welcher weiterhin mit Enforce-Werten gefüllt ist. Dieses bedeutet Enforce+ prüft wenn es an einen Vorgängerknoten geht, ob dieser schon einen Enforce-Wert hat und ob dieser Enforce-0 ist. Sollte dieser gefunden werden gibt es einen Zyklus. Enforce+ geht dabei wie Enforce selbst vor, nur hat es diese beiden Abbruchbedingungen, welche er finden muss. Weiterhin liegt die Schwierigkeit darin, ob der Zielknoten ein Kind-Knoten ist oder ein Roboter-Knoten. Denn vom Kindknoten gilt der Allquantor, welches bedeutet, dass es nicht zwingen nur einen Zyklus geben muss, sondern jeder der ausgehenden Pfade einen bilden muss. Sollte dieses nicht der Fall sein, kann die Umgebung, in diesem Falle das Kind, den Roboter vom erneuten gewinnen abhalten.
Somit kann Enforce+ den gesamten begehbaren Raum abdecken und sicher stellen, dass das Kind den Roboter niemals davon abhält das Ziel in der Zukunft erneut zu erreichen.

3.3 Controller

To review!
Der Controller ist die Schnittstelle zwischen dem Graphen, inklusive der Enforce-Werte und der GUI. Er bezieht aus der GUI die Position des Kindes und des Roboters, sowie die Standorte der Mauern, Zielfelder und Batterien. Er wird mit Start des Programms aus der GUI-Umgebung aufgerufen. Daraufhin initialisiert er den Graphen und berechnet

mit hinzunahme der aktuellen Position des Kindes sowie des Roboters, den enforceten Graphen, abhängig vom aktuellen Ziel. Sollte festgestellt werden, dass es keine Lösung für das Problem gibt, terminiert das Programm. Sollte dies nicht der Fall sein wird der Enforce-Graph für den ersten und die fortlaufenden Schritte des Roboters genutzt. Die Schritte des Kindes wird auch aus diesem Graphen bezogen, dabei bekommt der Controller die möglichen Moves aus dem Graphen und wählt zufällig einen. Mit der Hinzunahme des Energie-Problems wird der Controller um die Aufgabe erweitert, das Energie-Level zu überprüfen. Sollte der Worst-Case-Pfad zum nächsten Zielfeld, zuzüglich des Weges zum nächstbesten Energiefeld, größer als das Energie Level des Roboter sein, bewegt sich der Roboter zum nächsten Batteriefeld. Sollte auch dies nicht möglich sein terminiert das Programm. Der Weg zum nächstbesten Batteriefeld, ist der geringste Weg mit dem geringsten Aufwand aus den Worst-Case-Pfaden aller Batteriefeld. Als Energiefeld gelten die Felder um die Batterie.

Zur Berechnung der Pfade wird also die WorstCasePath Metrik festgelegt. Diese berechnet sich aus der Summe der Anzahl von Bewegungen die entstehen, wenn der Roboter von seinem aktuellen Zustand in den Zielzustand navigiert. Dabei nimmt der Roboter immer den Weg der zum geringsten Enforce-Wert führt. Das Kind nimmt auf diesem Pfad allerdings immer die Transitionen, die zu dem Zustand mit dem höchsten Enforce-Wert führen. Die Summe der Anzahlen der Bewegungen, die der Roboter auf diesem Pfad macht, ist dann der WorstCasePath ($WCPath(target)$).

Der Ablauf des Controllers ist in Algorithmus 1 vereinfacht dargestellt.

3.4 Optimierung

Die Optimierung des Programmes wäre an verschiedenen Stellen möglich, jedoch aufgrund der Aufgabenstellung nicht implementierbar. Ein möglicher Aspekt wird im folgenden vorgestellt, der sich eigenen würde in Hinsicht auf die Aufgabenstellung. Dabei wird das Abfahren von der Ladestation betrachtet.
Im Ausgangszustand schaut der Roboter in jedem Schritt wie weit er es zum Ziel hat und wie lange er bis zur nächsten Ladestation brauchen würde. Sollte die Ladestation dabei im schlimmsten Falle gerade noch zu erreichen sein, fährt er zu dieser. Sonst versucht er immer zu aller erst das Ziel zu erreichen und dann sich aufzuladen.
Nimmt man auf dieser Basis eine Ladestation an die hinter dem Roboter liegt und zwei oder mehr ab zu fahrende Ziele vor dem Roboter an, so kann man die Optimierungsmöglichkeit festmachen. In der Annahme hat der Roboter nur noch 50% geladen und kann damit maximal das erste Ziel erreichen, ohne aufladen zu müssen. Sollte er diese Möglichkeit also wählen, fährt er zum ersten Ziel um von diesem den Weg zurück zu fahren um die Ladestation zu erreichen, um anschließend die weiteren Ziele abzuarbeiten.
Würde der Roboter an gleicher Stelle prüfen, ob es besser wäre für den Ladungsverbrauch und die Geschwindigkeit der Abarbeitung, dass er erst zur Ladestation fährt und dann alle Ziele auf einmal abarbeiten könnte, so würde er sich mindestens einmal die Strecke zwischen Ladestation und Zielen sparen.
Es ist schnell zu sehen, dass solch eine Optimierung mit Einbezug des Kindes zu deut-

lichen Veränderungen führen kann, da der maximale Weg oft nicht der genommene ist. Dieses beruht auf der Zufälligkeit des Kindes. Weiterhin ist in der Aufgabenstellung vermerkt, dass der Roboter sein Ziel in der Zukunft irgendwann erreichen muss, somit auch zeitliche Umwege vollständig akzeptabel sind. Dieses Beispiel verdeutlicht nur, dass deutliche Optimierungen möglich wären, besonders in Hinsicht auf die Abarbeitungszeit. Eine weitere mögliche Optimierung liegt in der Berechnung des Kindes. Auch wenn dieses zufällig sich bewegt, sind die Bewegungen dennoch von Anfang an gegeben. Somit kommt es häufig vor, dass das Kind sich nur in einem Teil des Raumes aufhält, oder durch Ladestationen und Wände gar dazu gezwungen wird. Wenn der Roboter dieses mit einbezieht, könnte er den längsten Weg in den Kind-freien Zonen deutlich weiter senken.
Weitere Optimierungsmöglichkeiten sollen in diesem Exkurs nicht weiter behandelt werden.

4 Abschluss

4.1 Ausblick-Muller

Der gedächtnislose Enforce-Algorithmus ist in der Lage mit wenig Speicherplatz eine mögliche Lösung zu finden und diese zu erzwingen, gegen die Umgebung. Schwachstellen am Enforce sind jedoch zu erkennen, wenn eine 'entweder-oder' Bedingung gestellt wird. Enforce steuert immer auf ein Ziel hinaus und sollte dieses nicht zu erreichen sein, wird dieses gemeldet. Jedoch stellt dieses oft nicht die Realität da, in welcher es Alternativen gibt. Der Muller-Algorithmus löst dieses auf Kosten der Gedächtnislosigkeit. So kann er gegen die Umgebung bei mehreren möglichen Zielen eines davon auswählen und verfolgen, falls dieses erreichbar ist. Muller merkt sich woher er kommt und kann somit anhand der Ziele erkennen wohin er als nächstes will. Diese Erweiterung ist in realen System deutlich wünschenswert, wenn diese möglichst agil handeln sollen.

4.2 Aufgetretene Fehler

Die Bearbeitung dieser Aufgabe lief natürlich nicht komplett ohne Probleme von statten.

Durch Verständnisschwierigkeiten hatten wir anfangs das Problem, dass die enforcten Graphen nicht immer den bestmöglichen Weg berechnet haben, was aus einer falschen Enforce-Implementierung resultierte. Zudem hatten wir eingangs den Fehler gemacht, dass auch das Kind dem Roboter ausgewichen ist. Zwar nicht aktiv, aber es ist, wenn der Roboter in der Nähe war, nicht auf das Roboterfeld gegangen. Auch dieses wurde korrigiert, sodass nun der Roboter gar nicht erst solche Zustände generiert, in denen das Kind auf ein Feld gehen kann, welches der Roboter gerade okkupiert.

Eine weitere Herausforderung war, eine geeignete Metrik zur Weglängenberechnung bei der Entscheidung zu finden, wann man zur Batterie und wann zu einem Ziel navigiert. Zunächst hatten wir vermutet, dass man den längsten Pfad aus dem Graphen nehmen muss, was natürlich weder in polynomieller Zeit lösbar, noch für unseren Fall die richtige Lösung ist. Die nun verwendete $WCPath$-Metrik ist um einiges besser, da sie die Enforce

Werte aus dem Graphen verwendet und auf Basis derer die effektive Logik des Roboters utilisiert.

4.3 Ergebnisse

Die Bearbeitung dieses Projekts zeigt durchaus die Stärken von spieltheoretischen Datenstrukturen und Algorithmen auch in nicht-spielorientierten Szenarien auf. Gerade in der Welt der Eingebetteten Systeme sind diese durchaus praktikabel, da es oft darum geht, mit der Außenwelt zu interagieren.

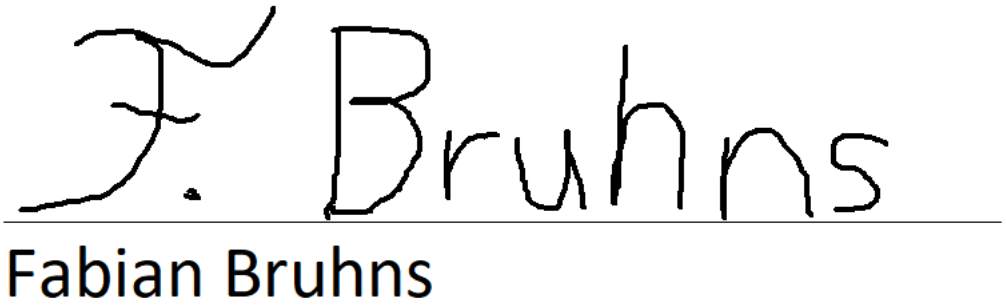
Natürlich ist die Darstellung von Szenarien der echten Welt mit Hilfe eines Spielgraphen weniger einfach als dieses idealisierte Beispiel, da bspw. Bewegungen von Robotern bzw. nicht Schrittweise passieren wie in diesem Fall. Allerdings ist ein Spielgraph, vor allem zusammen mit den entsprechenden Lösungsstrategien wie z.B. dem Enforce-Algorithmus eine durchaus gut anwendbare Möglichkeit, Controller für Eingebettete Systeme zu synthetisieren. Die benötigte Rechenleistung hält sich relativ niedrig, da die benötigten Graphen und enforce-Werte schon vor dem deployment des Systems berechnet werden können. Natürlich geschieht dies auf Kosten von benötigtem Speicherplatz, welcher aber heutzutage im Verhältnis zu Rechenleistung günstig ist.

5 Arbeitsverteilung

Teilaufgabe	Name	Anteil
1. Graphischer Editor mit Eingabe der Definitionen	Fabian	90%
	Jan-Henrik	5%
	Sven	5%

6 Erklärung

Hiermit versichern wir, dass wir diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Arbeit in der aufgeführten Arbeitsverteilung im Abschnitt 5 erledigt.



Algorithm 1 Pseudocode des Controllers

```
1: procedure findClosestBattery(graphs, position)
       return Feld um Batterien, das am nächsten an der gegebenen Position ist
2: end procedure
3:
4: procedure findReachableBatteries(robotPosition, childPosition)
       return Batterien, die vom Roboter erreichbar sind
5: end procedure
6:
7: procedure isSolveable(graph, robotPosition, childPosition)
8:     state ← graph.findState(robotPosition, childPosition)
       return state.enforceValue! = −1
9: end procedure
10:
11: procedure Controller(gameField, targetFields, batteryFields)
12:     graph ← generateGraph(gameField)                ▷ Spielgraphen generieren
13:     batteryGraphs ← enforceGraphs(graph, batteryFields)   ▷ Graphen für 9
    Felder um die Batterien enforcen, um diese als Ziel zu berechnen/verwenden
14:
15:     while true do
16:         for all targetField in targetFields do
17:             enforcedGraph ← enforceGraph(graph, targetField.position)
                                                ▷ Spielgraph auf aktuelles Ziel enforcen
18:             activeGraph ← enforcedGraph
19:
20:             while targetField.position ≠ robot.position ∧
    isSolveable(enforcedGraph, robot.position, child.position) do
21:                 robotToTargetCost ← enforcedGraph.WCPath(robot.position, child.position)
22:                 targetToBatteryCost ← findClosestBattery(targetField.position, batteryGraphs)
23:
24:                 if (targetToBatteryCost + robotToTargetCost) > robot.energy then
25:                     reachableBatteries ← findReachableBatteries(robot.position, child.position)
26:                     targetBattery ← findClosestBattery(reachableBatteries, targetField.position)
27:                     activeGraph ← targetBattery.graph
28:                 end if
29:
30:                 robot.makeMove(activeGraph.getBestRobotMove(robot.position, child.position))
31:                 child.makeMove(activeGraph.getRandomChildMove(robot.position, child.position))
32:             end while
33:         end for
34:     end while
35: end procedure
```