

ספר הפרויקט

הקדמה

בפרויקט מימשנו את אלגוריתם BITE (Bar-Ilan Teamwork Engine) שהומצא ע"י פרופ' גל קמינקא. אלגוריתם זה מהווה ארכיטקטורה להרצת משימות במערכות מרובות רובוטים בצורה סינכרונית ועם שיתוף מידע ושיתוף פעולה בין הרובוטים.

להלן האלגוריתם כפי שמוצג במאמר:

Require: Plan $P = \langle B, H, N, b_0 \rangle$
Require: Knowledgebase W
Require: Robots R
Require: Allocation Procedure $ALLOCATE$
Require: Voting Procedure $VOTE$
Require: Condition Testing Procedure $TEST$
Require: Belief Update Procedure $RECEIVE$
Require: Belief Revision Procedure $FUSE$
Require: Start Execution Procedure $S.START$
Require: Stop Execution Procedure $S.STOP$
Require: Inform Procedure $INFORM$

```

1:  $S \leftarrow \emptyset$ 
2:  $b \leftarrow b_0$ 
3:  $T \leftarrow \emptyset$ 
4:  $PUSH(S, \langle b, R \rangle)$ 
5: while  $\exists n$ , where  $(b, n) \in H$  do
6:    $A \leftarrow \{n | (b, n) \in H\}$ 
7:    $C \leftarrow \{a | a \in A, TEST(preconds(a), W)\}$ 
8:   for all  $c \in C$  do
9:      $INFORM(preconds(c), R)$ 
10:   $\langle b, R \rangle \leftarrow ALLOCATE(C, P, S, W, R)$ 
11:   $PUSH(S, \langle b, R \rangle)$ 

12: for all  $\langle t, R \rangle \in T - S$  do
13:   if  $t$  is running  $S.STOP(t, R)$ 

14: for all  $\langle s, R \rangle \in S$  do
15:   if  $s$  not running  $S.START(s, R)$ 

16:  $T \leftarrow \emptyset$ 

17:  $E \leftarrow \emptyset$ 
18: while  $E \neq \emptyset$  do
19:    $\langle K, R \rangle \leftarrow RECEIVE(W)$ 
20:    $W \leftarrow FUSE(W, \langle K, R \rangle)$ 
21:   for all  $\langle s, R \rangle \in S$  do
22:     if  $\exists k$  such that  $k$  used in  $s$  then
23:        $INFORM(k, R)$ 
24:    $E \leftarrow \{a | a \in S, TEST(termconds(a), W)\}$ 

25: while  $E \neq \emptyset$  do
26:    $\langle e, R \rangle \leftarrow Pop(S)$ 
27:    $T \leftarrow T \cup \{\langle e, R \rangle\}$ 
28:   if  $e \in E$  then
29:      $INFORM(termconds(e), R)$ 
30:    $E \leftarrow E - \{e\}$ 

31:  $A \leftarrow \{n | (e, n) \in N\}$ 
32:  $C \leftarrow \{a | a \in A, TEST(preconds(a), W)\}$ 
33: if  $C \neq \emptyset$  then
34:   for all  $c \in C$  do
35:      $INFORM(preconds(c), R)$ 
36:    $\langle b, R \rangle \leftarrow VOTE(C, P, S, W, R)$ 
37:   Goto 3
38:  $\langle b, R \rangle \leftarrow PEEK(S)$ 
39: if  $b \neq \emptyset$  then
40:   Goto 17
41: Halt.

```

► T holds a list of behaviors to stop
 ► Entire team (initial R) runs b_0
 ► children of b
 ► Inform all robots associated with b that preconditions of child are true
 ► Allocate C behaviors, get new R (subteam)
 ► Mark down new subteam members associated with new b
 ► Plans on the stop list, that's not on S
 ► Start execution of all behaviors in S
 ► Stopped everything that needed. Reset.
 ► Get updates from others, not just from UPDATE
 ► If new knowledge, mark down also its source, not just REVISE
 ► Inform subteam R associated with behavior s of any relevant knowledge
 ► Check termination conditions
 ► Not just pop, also determine which robots are affected
 ► Inform all of them that termination conditions for e are true
 ► e is the top-most terminated behavior
 ► There are potential followers to e
 ► Inform all robots associated with e that the preconditions for specific sequential behaviors are true
 ► Voters are those associated with the terminating behavior; R remains constant
 ► Not potential followers, continue with parent; get parent's subteam R

מדריך למתכנת

מבחינת המימוש של האלגוריתם, הפרדנו אותו לשני חלקים עיקריים. החלק הראשון אחראי להריץ את המשימות, לסנכרן בין הרובוטים ולסיים משימות בעת הצורך. זה החלק שמרכיב את עיקר האלגוריתם. החלק השני אחראי על ניהול מאגר המידע של הרובוט ושיתוף מידע בין הרובוטים. חלק זה מנהל מאגר מידע של איזה רובוט מריץ איזה משימות, ולפי הצורך, בעת קבלת פריט מידע חדש, מעדכן את הרובוטים הרלוונטיים בפריט מידע זה.

שני החלקים הנ"ל רצים במקביל (בשני Processes נפרדים) כך שהבקרה על המידע של הרובוט (המגיע מחיישנים ומדיווחים של רובוטים אחרים) והשיתוף עם הרובוטים האחרים קורים תוך כדי הניהול התקין של המשימות. מבחינת הפסאודו קוד של האלגוריתם ניתן לומר שהחלק השני הוא השורות 19-23, והחלק הראשון הוא כל השאר.

כפי שניתן לראות, האלגוריתם עצמו מורכב ממספר חלקים:

- שורות 1-11: הרכבת המחסנית של ה-Behaviors (פירוק של המשימה בצורה היררכית לתתי משימות)
- שורות 12-16: סיום המשימות שלא נמצאות בענף שנבחר והתחלת המשימות שבמחסנית.
- שורות 17-24: בקרה על המשימות שבמחסנית.
- שורות 25-30: עדכון ה-Behaviors שמועמדים לסיום.
- שורות 31-41: הרחבת המחסנית למשימה הבאה.

נתייחס לכל חלק בנפרד.

חלק א – הרכבת המחסנית

בקוד המתודה נקראת ExpandStackHierarchically.

בחלק זה כל רובוט בוחן את ה-Preconditions של כל אחד מה-Behaviors הבאים בתור לפי ה-Knowledgebase שלו, ומעדכן את שאר הרובוטים איזה תנאים הוא חושב שמתקיימים. תנאים אלה כתובים במאגר המידע כפרדיטקים שיכולים לקבל את הערכים True או False. לפני הקריאה ל-Allocate יש לדאוג שכל הרובוטים יהיו מסונכרנים (יקראו ביחד למתודת ההקצאה). לכן הוספנו בקוד בין שורות 9 ל-10 המתנה לכל חברי הקבוצה. זהו שינוי שהוספנו לאלגוריתם המקורי. פסאודו קוד כולל כל השינויים שהוספנו מופיע בסוף פרק זה.

נקודות חשובות למתכנת לשים לב אליהן בחלק זה:

- מתודת ההמתנה שמימשנו (WaitForTeam) מחכה לסיגנל מכל הרובוטים שנמצאים בתת הקבוצה הנוכחית שמבצעים את אותה משימה בתוכנית. אם רוצים לתמוך בסוגים שונים של פרוטוקולי סנכרון בין הרובוטים יש לשנות פונקציה זו.
- את הקריאה ל-Allocate מימשנו ע"י קריאה ל-Service שמוגדר לכל Node ב-Plan. כלומר, המתודה שמבצעת אלוקציה של המשימות לרובוטים איננה חלק מהאלגוריתם המרכזי אלא Process נפרד שמספק Service של אלוקציה. הדבר מאפשר לממש איזה מתודת אלוקציה שרוצים, וכדי שהיא תפעל עם

האלגוריתם של BITE יש לדאוג שהיא תספק את ה-Service המתאים, וכן להגדיר בקונפיגורציה של ה-Plan את שם המתודה. בנוסף, ניתן להגדיר כמה מתודות אלוקציה שונות, ולכל Node להשתמש במתודה אחרת, כפי שנראה בדוגמא בהמשך.

- ה-Service של האלוקציה כפי שהגדרנו אותו מקבל שתי רשימות – את ה-Nodes האפשריים לבחירה ואת קבוצת הרובוטים שיש לחלק ביניהם את המשימות. ניתן להרחיב אותו ע"י הוספת שדות ב-Service (למשל רשימה נוספת של פרמטרים אפשריים לבחירה ביניהם)

חלק ב – סיום המשימות הישנות והתחלת המשימות החדשות

בחלק זה הוספנו פרמטר נוסף לזוג $\langle b, R \rangle$ ושמרנו ביחד איתם גם את הפרמטרים שנשלחו לאותו Behavior בזמן ההפעלה. דבר זה מאפשר להבדיל בין שני Behaviors שנבחרו עם פרמטרים שונים כך שיש לסיים את ה-Behavior הקודם ולהתחיל את ה-Behavior החדש.

את סיום המשימות מימשנו ע"י שידור איתות על Topic שהגדרנו מראש שכל Behavior יאזין לו. כשה-Behavior מקבל את האיתות הוא יודע שעליו לסיים. את סיום ה-Behavior מימשנו ע"י קריאה לפונקציה `signal_shutdown` המובנית ב-ROS. הקריאה הזאת גורמת לזריקת Exception בכל סיום של Node (כנראה בעיה ב-ROS), אך זה לא מפריע לריצת התוכנית ולכן השארנו את זה ככה.

את התחלת המשימות מימשנו ע"י מחלקה בשם `ROSLaunch` שמאפשרת הרצת Nodes חדשים של ROS מתוך הקוד. כך כל Behavior רץ ב-`Process` נפרד משל עצמו. עקרונית ניתן בצורה זו להריץ כל Node של ROS (בין אם נכתב בפיתון ובין אם נכתב ב-C++), אך כדי להקל על כתיבת Behaviors יצרנו מחלקה אחת בשם `BehaviorLauncher.py` שאנחנו תמיד מפעילים אותה והיא יוצרת Instance של ה-Behavior הרצוי. לכן כרגע ניתן לכתוב Behaviors רק בפיתון, אך בקלות ניתן להרחיב את המתודה שנקראת `StartBehavior` שתריץ את ה-Node שרוצים.

כדי להקל על כתיבת Behaviors חדשים יצרנו מחלקה אבסטרקטית בשם `BehaviorBase.py` שממנה יורשים כל ה-Behaviors. במחלקה זו מימשנו מתודות בסיסיות כמו `GetKnowledge` או `Update` שמאפשרות גישה ל-`Knowledgebase` (קבלת מידע ועדכון מידע) בצורה פשוטה (ללא צורך בהרשמה ל-Topics והכרת מבנה ההודעות). מצד אחד הדבר מגביל לשימוש בפיתון בלבד, אך מצד שני מאוד מקל על כתיבת Behaviors. בדומה להסבר למעלה, ניתן להוסיף תמיכה גם בקבצי C++, אך זה כורך הכרה של Topics ו-Services של ROS, והרשמה ל-Topics המתאימים שהמערכת שלנו עובדת איתם.

בכל הפעלה או סיום של Behavior כל רובוט משדר לכל שאר חברי הקבוצה שהוא מפעיל / מסיים את ה-Behavior. כך כאשר רובוט מסוים מקבל פריט מידע שרלוונטי לכל מי שמריץ Behavior כלשהו, הוא יוכל לדעת מי באותו רגע מריץ Behavior זה ויכול לדווח לו על פריט המידע הזה. כמובן שחלק זה ממומש ב-`Process` שאחראי על ניהול מאגר המידע של הרובוט.

חלק ג – ניטור הפעולות שבמחסנית

כיון שעדכון מאגר המידע מתבצע ב-Process נפרד ובמקביל לריצת האלגוריתם עצמו, בחלק זה האלגוריתם רץ בלולאה אינסופית (עם המתנה קצרה בין לבין) ופשוט בודק את ה-Termination Conditions של כל ה-Behaviors שבמחסנית. במידה ואחד מהם מתקיים הוא עובר לשלב הבא.

את חלק זה מימשנו בצורה שתהיה זהה לפסאודו קוד (לולאה אינסופית שכל הזמן דוגמת את ה-Termination Conditions), אך ניתן לייעל ע"י שמירה במאגר המידע איזה Termination Conditions רלוונטיים כרגע, ובמידה ואחד מהם הוא True לאותת לאלגוריתם הראשי שמשו צריך להסתיים. ככה לא בודקים כל הזמן את כל ה-Termination Conditions.

נקודה נוספת שיש לשים לב אליה – אנחנו מימשנו מתודת Fuse מאוד טריוויאלית (כל מה שרובוט כלשהו אומר, כולם מאמינים לו). אם רוצים לאפשר החלטה יותר מתוחכמת אם לקבל דעה של רובוט מסוים, או לשקלל בצורה כלשהי בין מידע של רובוטים שונים, יש להרחיב את זה בקובץ KnowledgeBae.py (הקובץ שאחראי על מאגר המידע).

חלק ד – עדכון הפעולות שמועמדות לסיום

בחלק זה מוציאים מהמחסנית את ה-Behavior שצריך להסתיים וגם את כל הבנים שלו. כאמור לעיל, אנחנו שומרים ב-T את השלשות $\langle b, R, V \rangle$ הכולל גם את הפרמטרים שכל Behavior הופעל איתם. במקרה ש-Behavior אמור להסתיים הוא מדווח לכל הרובוטים שצריך לסיים אותו וממשיך לחלק הבא.

חלק ה – הרחבת המחסנית למשימה הבאה

גם בחלק זה שינינו קצת מהאלגוריתם המקורי. הבעיה באלגוריתם המקורי הייתה שהתנאי, אם להרחיב את המחסנית למשימה הבאה או להמשיך לנטר את ה-Behaviors שכרגע במחסנית, היה נבדק בכל רובוט בפני עצמו ולפי המידע שיש לו כרגע. כך יכול להווצר מצב שרובוט אחד יודע שאפשר להמשיך למשימה הבאה (ה-Preconditions מתקיימים) אך רובוטים אחרים לא יודעים זאת. לכן הם מדלגים חזרה לחלק של ניטור ה-Behaviors בעוד שהרובוט הזה נכנס לתנאי ורוצה לעשות Voting הלאה. כדי לפתור את הבעיה הזאת הוצאנו את הדיווח לשאר חברי הקבוצה מחוץ לתנאי כך שכל רובוט קודם כל מודיע לכל חברי הקבוצה איזה Preconditions לדעתו מתקיימים. לאחר מכן, לפני בדיקת התנאי, הרובוטים ממתינים שכולם ידווחו (אותה המתנה שמימשנו בחלק א לפני הקריאה לאלוקציה). רק לאחר שכל הרובוטים דיווחו ונמצאים ביחד, כולם ביחד בודקים את התנאי אם ניתן להמשיך למשימה הבאה, ואם כן קוראים ביחד לפרוצדורה של Vote.

בדומה לחלק א, את הקריאה ל-Vote מימשנו ע"י קריאה ל-Service נפרד שמוגדר ע"י המשתמש בקובץ הקונפיגורציה. לכן, גם כאן המתודה מופעלת בצורה נפרדת מהאלגוריתם הראשי ומאפשרת שימוש באיזה שיטת Voting שרוצים, ואף בשיטות שונות ל-Behaviors שונים.

להלן הפסאודו קוד המעודכן לאחר השינויים שערכנו:

```

Require: Plan  $P = \langle B, H, N, b_0 \rangle$ 
Require: Knowledgebase  $W$ 
Require: Robots  $R$ 
Require: Allocation Procedure ALLOCATE
Require: Voting Procedure VOTE
Require: Condition Testing Procedure TEST
Require: Belief Update Procedure RECEIVE
Require: Belief Revision Procedure FUSE
Require: Start Execution Procedure S:START
Require: Stop Execution Procedure S:STOP
Require: Inform Procedure INFORM

0:  $v = null$ 
1:  $S \leftarrow \emptyset$ 
2:  $b \leftarrow b_0$ 
3:  $T \leftarrow \emptyset$ 
4:  $PUSH(S, \langle b, R, v \rangle)$ 
5: while  $\exists n$ , where  $(b, n) \in H$  do
6:    $A \leftarrow \{n | (b, n) \in H\}$ 
7:    $C \leftarrow \{a | a \in A, TEST(preconds(a), W)\}$ 
8:   for all  $c \in C$  do
9:      $INFORM(preconds(c), R)$ 
10:   waitForTeam(R)
11:    $C \leftarrow \{a | a \in A, TEST(preconds(a), W)\}$ 
12:    $\langle b, R, v \rangle \leftarrow ALLOCATE(C, P, S, W, R)$ 
13:    $PUSH(S, \langle b, R, v \rangle)$ 

14: for all  $\langle t, R, v \rangle \in T - S$  do
15:   if  $t$  is running S:STOP( $t, R, v$ )

16: for all  $\langle s, R, v \rangle \in S$  do
17:   if  $s$  not running S:START( $s, R, v$ )

18:  $T \leftarrow \emptyset$ 
19:  $E \leftarrow \emptyset$ 
20: while  $E = \emptyset$  do
21:    $\langle K, R \rangle \leftarrow RECEIVE(W)$ 
22:    $W \leftarrow FUSE(W, \langle K, R \rangle)$ 
23:   for all  $\langle s, R, v \rangle \in S$  do
24:     if  $\exists k$  such that  $k$  used in  $s$  then
25:        $INFORM(k, R)$ 
26:    $E \leftarrow \{a | a \in S, TEST(termconds(a), W)\}$ 

27: while  $E \neq \emptyset$  do
28:    $\langle e, R, v \rangle \leftarrow Pop(S)$ 
29:    $T \leftarrow T \cup \{\langle e, R, v \rangle\}$ 
30:   if  $e \in E$  then
31:      $INFORM(termconds(e), R)$ 
32:      $E \leftarrow E - \{e\}$ 

33:  $A \leftarrow \{n | (e, n) \in N\}$ 
34:  $C \leftarrow \{a | a \in A, TEST(preconds(a), W)\}$ 
35: for all  $c \in C$  do
36:    $INFORM(preconds(c), R)$ 
37: waitForTeam(R)
38:  $C \leftarrow \{a | a \in A, TEST(preconds(a), W)\}$ 
39: if  $C \neq \emptyset$  then
40:    $\langle b, R, v \rangle \leftarrow VOTE(C, P, S, W, R)$ 
41:   Goto 4
42: if  $S \neq \emptyset$  then
43:   Goto 19
44: Halt.

```

▷ T holds a list of behaviors to stop
 ▷ Entire team (initial R) runs b_0
 ▷ children of b
 ▷ Inform all robots associated with b that preconditions of child are true
 ▷ Allocate C behaviors, get new R (subteam)
 ▷ Mark down new subteam members associated with new b
 ▷ Plans on the stop list, thats not on S
 ▷ Start execution of all behaviors in S
 ▷ Stopped everything that needed. Reset.
 ▷ Get updates from others, not just from UPDATE
 ▷ If new knowledge, mark down also its source, not just REVISE
 ▷ Inform subteam R associated with behavior s of any relevant knowledge
 ▷ Check termination conditions
 ▷ Not just pop, also determine which robots are affected
 ▷ Inform all of them that termination conditions for e are true
 ▷ e is the top-most terminated behavior
 ▷ Inform all robots associated with e that the preconditions for specific sequential behaviors are true
 ▷ There are potential followers to e
 ▷ Voters are those associated with the terminating behavior, R remains constant

נקודות לשיפור

להלן נעיר כמה נקודות שניתן לשפר במימוש האלגוריתם:

- בעת קבלת מידע חדש (בחלק של ה-KnowledgeBase), אם פריט מידע זה רלוונטי לרובוטים אחרים הרובוט מודיע להם אותו. במימוש פשוט מדי זה נכנס ללולאה אינסופית, שהרי ברגע שמתקבל פריט מידע שרלוונטי לכל הרובוטים, הם יודיעו אותו אחד לשני בלי סוף. כדי להתגבר על הבעיה, בדקנו אם ערך המידע שהתקבל שונה ממה שהיה לפני כן במאגר המידע של הרובוט, ורק אם כן הוא מודיע אותו לשאר הרובוטים (ככה בהכרח בפעם הראשונה שהוא יקבל את המידע הוא יודיע אותו, ולאחר מכן כבר לא יודיע).
- עדיין נוצרות כפילויות (שהרי אם רובוט א' מקבל מידע חדש ומודיע אותו לרובוט ב', אז רובוט ב' יודיע שוב לרובוט א' כי מבחינתו המידע היה חדש), ובנוסף יש מקרים בהם זה נכנס ללולאה אינסופית (אם שני רובוטים מודיעים באותו שבריר שניה ערכים שונים של אותו פריט מידע, זה כל הזמן משתנה והם נכנסים ללולאה אינסופית). לכן הוספנו עוד תנאי – שהרובוט לא מודיע את המידע לרובוט שהודיע לו ישירות.
- עדיין יש עוד מקום לשפר את הפרוטוקול שאחראי להחליט מתי יש לשתף את המידע בין הרובוטים.
- הפרוטוקול שאחראי על המתנה לחברי הקבוצה מחכה תמיד לכל חברי הקבוצה. במקרה שאחד מחברי הקבוצה מת, הקבוצה אף פעם לא תמשיך הלאה. יש מקום לשפר עם פרוטוקול יותר מתוחכם, ואולי גם עם timeouts למקרה שחלק מהרובוטים כבר לא בחיים.
- מתודת ה-Fuse של מידע חדש שממשנו היא די פשוטה – תמיד לקבל את המידע החדש. ניתן לשפר ולהוסיף מתודות יותר מתוחכמות שידעו לשקלל את המידע לפי הרובוט שממנו הוא התקבל.

מדריך למשתמש

כדי להשתמש באלגוריתם יש ליצור Package חדש ב-ROS. הוראות ליצירת פקאג' ניתן למצוא בויקי של ROS

בכתובת: <http://wiki.ros.org/ROS/Tutorials/CreatingPackage>

יש ליצור בתיקיה הראשית של ה-Package את קבצי הקונפיגורציה plan.xml ו-dependencies.xml. ניתן לראות

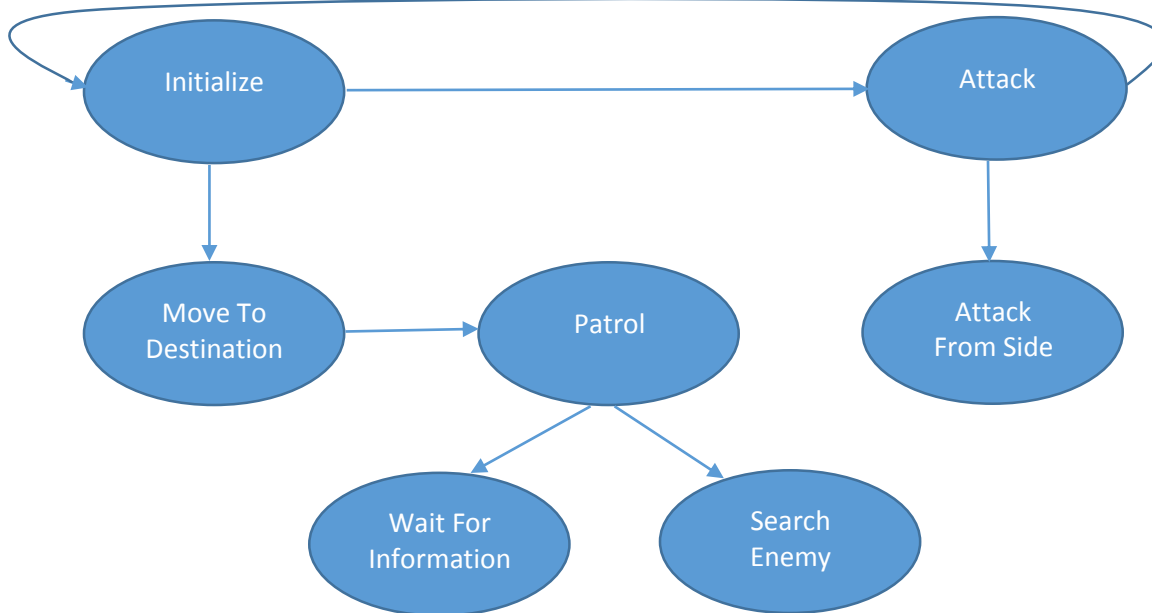
דוגמא לקבצים אלה בתיקיה הראשית של BITE. בנוסף, יש להעתיק את הקבצים BehaviorBase.py ו-

BehaviorLauncher.py מתיקית ה-src של BITE לתיקית ה-src של ה-Package החדש.

כעת יש לממש את ה-Behaviors הרצויים. כדי להמחיש זאת נראה מימוש תוכנית פשוטה יחסית לדוגמא.

ה-Plan שאנחנו רוצים לממש הוא מעין פטרוול, לא בצורה כל כך אמיתית, אבל כדי שימחיש את השימוש ב-BITE.

ה-Plan נראה ככה:



כל הרובוטים מתחילים מה-Behavior הראשון שהוא Initialize. הם הולכים ביחד לנקודת יעד כלשהי, ומשם מתחילים לפטרוול. הפטרוול עצמו מחולק לשתי קבוצות – קבוצה אחת מחפשת אויבים והשנייה נמצאת בחדר בקרה ומחכה לדיווחים. ברגע שהתקבלו כמה דיווחים על אויבים, ה-Initialize יכול להסתיים וממשיכים הלאה ל-Attack. כאן מופעלת אלוקציה של Attack From Side לכל רובוט עם פרמטר מאיזה צד לתקוף.

מימוש ה-Behaviors הינו פיקטיבי בדוגמא שלנו, כל Behavior רק מדפיס למסך ומעדכן את מאגר המידע. ניקח לדוגמא את ה-Behavior של חיפוש האויב (Search Enemy):

```

SearchEnemy.py (~/.workspace/catkin/src/bite_sample/src) - gedit
Open Save Undo Redo
SearchEnemy.py
1 import rospy
2 from BehaviorBase import BehaviorBase
3 from random import randrange
4
5 class SearchEnemy(BehaviorBase):
6     def __init__(self, robotName, behaviorName, params):
7         BehaviorBase.__init__(self, robotName, behaviorName, params)
8         rospy.sleep(0.5)
9
10    def run(self):
11        while True:
12            x = randrange(3, 14)
13            print str.format('In {0} seconds going to find enemy', x)
14            rospy.sleep(x)
15            if not self.getKnowledge('Attack') == 'True':
16                self.update('EnemyFound', self.robotName)
17            else:
18                break

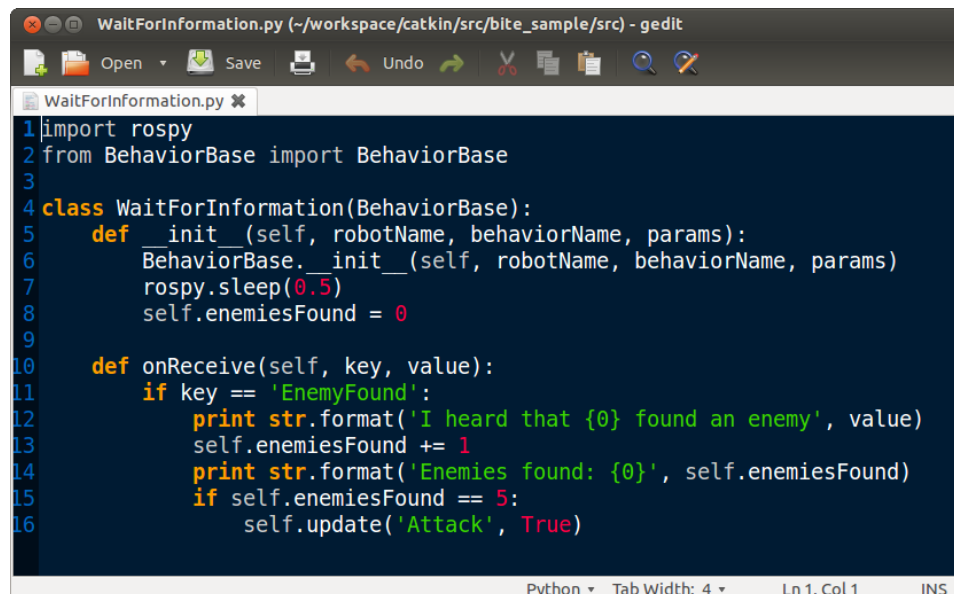
```

Python Tab Width: 4 Ln 12, Col 33 INS

במימוש של ה-Behavior פשוט רצנו בלולאה אינסופית וכל זמן רנדומלי עדכנו שמצאנו אויב. נשים לב לשימוש במתודה (`getKnowledge('Attack')`). כך אנחנו ניגשים למאגר המידע ומבקשים את הערך של פריט המידע שנקרא `Attack`. אם הערך שלו הוא `True` (כלומר, הוחלט לצאת להתקפה) אנחנו יוצאים מהלולאה האינסופית שלנו. אחרת, אנחנו מעדכנים את מאגר המידע שלנו שמצאנו אויב נוסף ע"י שימוש במתודה `update`.

הדוגמא הנ"ל ממחישה בצורה טובה איך יוצרים Behavior חדש. על ה-Behavior לרשת מ-BehaviorBase, ולממש את המתודה `run`. ברגע הפעלת ה-Behavior, המתודה הזאת תופעל ומה שבתוכה יתבצע.

כדי להמחיש Feature נוסף נראה איך מימשנו את `WaitForInformation`:



```
WaitForInformation.py (~/.workspace/catkin/src/bite_sample/src) - gedit
1 import rospy
2 from BehaviorBase import BehaviorBase
3
4 class WaitForInformation(BehaviorBase):
5     def __init__(self, robotName, behaviorName, params):
6         BehaviorBase.__init__(self, robotName, behaviorName, params)
7         rospy.sleep(0.5)
8         self.enemiesFound = 0
9
10    def onReceive(self, key, value):
11        if key == 'EnemyFound':
12            print str.format('I heard that {0} found an enemy', value)
13            self.enemiesFound += 1
14            print str.format('Enemies found: {0}', self.enemiesFound)
15            if self.enemiesFound == 5:
16                self.update('Attack', True)
```

נשים לב ש-Behavior זה לא מממש כלל את המתודה `run`. כלומר, הוא לא עושה כלום כשמפעילים אותו. אבל הוא מממש את המתודה `onReceive` שנקראית כאשר מתקבל מידע כלשהו לרובוט. ה-Behavior הזה מנהל מונה של כמה אויבים נמצאו וכאשר הערך הגיע ל-5 (סתם מספר שרירותי שבחרנו) הוא מעדכן את מאגר המידע שצריך כרגע לתקוף.

כמובן כל Behavior יכול לממש גם את `run` וגם את `onReceive` ולהגיב בהתאם. הראנו כאן שימוש בשתי המתודות האלה בצורה פשוטה.

דבר נוסף שניתן להשתמש בו הוא הפרמטרים שנשלחו ל-Behavior. נראה זאת במימוש AttackFromSide:

```
AttackFromSide.py (~/.workspace/catkin/src/bite_sample/src) - gedit
1 import rospy
2 from BehaviorBase import BehaviorBase
3 from random import randrange
4
5 class AttackFromSide(BehaviorBase):
6     def __init__(self, robotName, behaviorName, params):
7         BehaviorBase.__init__(self, robotName, behaviorName, params)
8         rospy.sleep(0.5)
9
10    def run(self):
11        print str.format('Attacking from the {0}', self.params[0])
12        x = randrange(3, 24)
13        print str.format('In {0} seconds going to destroy enemy', x)
14        rospy.sleep(x)
15        if self.getKnowledge('EnemyDestroyed') == 'False':
16            if self.getKnowledge('OddNumberOfEnemiesDestroyed') == 'False':
17                self.update('OddNumberOfEnemiesDestroyed', True)
18            else:
19                self.update('OddNumberOfEnemiesDestroyed', False)
20            self.update('EnemyDestroyed', True)
```

ניתן לשים לב שבתחילת המתודה הדפסנו מאיזה כיוון אנחנו מתכוונים לתקוף. למרות שכל רובוט קיבל את המשימה לתקוף מהצד, כל רובוט יכול לקבל אותה עם פרמטר שמגדיר מאיזה צד לתקוף וכך יכול לדעת לתקוף מהצד הזה.

השתמשנו פה בבדיקה נוספת עם פרדיקט שנקרא OddNumberOfEnemiesDestroyed שכשמו – מגדיר אם מספר אי-זוגי של אויבים הושמדו. עשינו את זה בשביל שה-Plan תרוץ פעמיים עד שיושמדו מספר זוגי של אויבים, כפי שנראה בהמשך הדוגמא.

לסיכום קצר, ראינו איך מממשים Behaviors ואיזה פונקציונליות יש לכל Behavior עם גישה למאגר המידע, עדכון המידע במאגר, שימוש בפרמטרים שהתקבלו לתוכנית, ומימוש מתודה שמגדירה טיפול בקבלת מידע חדש.

מימוש של כל ה-Behaviors ניתן לראות בקבצי הדוגמא המלאים המצורפים.

בנוסף למימוש ה-Behaviors עצמם יש לעדכן את הקובץ BehaviorLauncher.py שיכיל את כל ה-Behaviors

שהכנו. כך זה נראה בדוגמא שלנו:

```
BehaviorLauncher.py (~/.workspace/catkin/src/bite_sample/src) - gedit
1 #!/usr/bin/env python
2
3 import rospy
4 from sys import argv
5 from std_msgs.msg import String
6
7 # Import all behaviors here ->
8 from Initialize import Initialize
9 from MoveToDestination import MoveToDestination
10 from Patrol import Patrol
11 from SearchEnemy import SearchEnemy
12 from WaitForInformation import WaitForInformation
13 from Attack import Attack
14 from AttackFromSide import AttackFromSide
15
16 class BehaviorLauncher:
17     def init (self, robotName, behaviorName, params):
```

לאחר מכן, יש לעדכן את קובץ ה-plan.xml. קובץ זה מכיל את המידע על ה-Nodes השונים, תנאי הכניסה והיציאה מהם, מתודת האלוקציה וההצבעה הרלוונטית עבורם והקשרים בין ה-Nodes. כך נראה ה-plan שלנו:

```

1 <?xml version="1.0"?>
2 <data>
3   <nodes>
4     <node nodeName="initialize">
5       <behaviorName>Initialize</behaviorName>
6       <allocateMethod>allocate_all</allocateMethod>
7       <voteMethod>vote</voteMethod>
8       <preCond>OddNumberOfEnemiesDestroyed</preCond>
9       <termCond>Attack</termCond>
10    </node>
11    <node nodeName="move_to_destination">
12      <behaviorName>MoveToDestination</behaviorName>
13      <allocateMethod/>
14      <voteMethod>vote</voteMethod>
15      <termCond>AllRobotsReachedDestination</termCond>
16    </node>
17    <node nodeName="patrol">
18      <behaviorName>Patrol</behaviorName>
19      <allocateMethod>allocate_separate</allocateMethod>
20      <voteMethod>vote</voteMethod>
21      <preCond>AllRobotsReachedDestination</preCond>
22    </node>
23    <node nodeName="search_enemy">
24      <behaviorName>SearchEnemy</behaviorName>
25      <allocateMethod/>
26      <voteMethod/>
27    </node>
28  </nodes>
29  <heirarchicalEdges>
30    <heirarchicalEdge from="initialize" to="move_to_destination"/>
31    <heirarchicalEdge from="patrol" to="wait_for_information"/>
32    <heirarchicalEdge from="patrol" to="search_enemy"/>
33    <heirarchicalEdge from="attack" to="attack_from_side"/>
34  </heirarchicalEdges>
35  <sequentialEdges>
36    <sequentialEdge from="move_to_destination" to="patrol"/>
37    <sequentialEdge from="initialize" to="attack"/>
38    <sequentialEdge from="attack" to="initialize"/>
39  </sequentialEdges>
40 </data>

```

כל Node ב-Plan מוגדר בפני עצמו עם שם ה-Behavior כפי שהגדרנו. לכל Node ניתן להגדיר מתודת אלוקציה משלו ומתודת הצבעה משלו. שדות אלו יכולים להיות ריקים או שונים בין Node ל-Node (לדוגמא, ה-Node הראשון משתמש במתודת אלוקציה שהגדרנו בשם allocate_all, וה-Node השלישי משתמש במתודה allocate_separate).

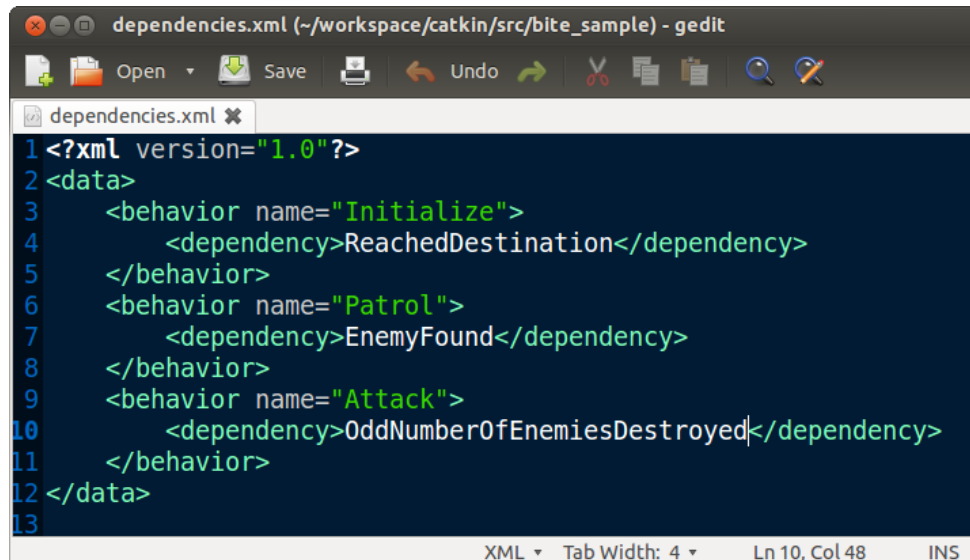
בנוסף, כל Node יכול להכיל כמה Preconditions וכמה Termconditions או לא להכיל בכלל. לבסוף יש להגדיר את הקשתות השונות בין ה-Nodes שב-Plan.

בדוגמא שלנו, ה-Behavior הראשון מסתיים כאשר הפרדיקט Attack = True, ולכן שמנו כ-termCond את הפרדיקט Attack. זה גורם שכאשר הרובוטים נמצאים בתת משימה של Initialize ומחליטים שצריך לתקוף (כפי שראינו קודם שה-Behavior שממתין למידע סופר עד 5 ומעדכן שצריך לתקוף) אז ה-Behavior הנ"ל מסתיים וכן כל תת העץ שנבנה ממנו, והרובוטים ביחד קוראים למתודה vote שלו ומחליטים להמשיך למשימה הבאה של תקיפת האויב.

הוספנו גם preCond ל-Behavior הזה שהוא OddNumberOfEnemiesDestroyed. כך, כפי שראינו קודם, לאחר שהרובוטים יסיימו את Attack, הם יעשו Voting נוסף ויבחרו לבצע שוב Initialize כיון שהפרדיקט הזה יהיה True.

בפעם השניה שהם ישמידו אויב, ערך הפרדיקט הזה יהיה False, ולכן לא יהיה לאן להתקדם והרובוטים יסיימו את התוכנית.

קובץ קונפיגורציה נוסף שיש לעדכן הוא ה-dependencies.xml. כך הוא נראה בדוגמא שלנו:



```
1 <?xml version="1.0"?>
2 <data>
3   <behavior name="Initialize">
4     <dependency>ReachedDestination</dependency>
5   </behavior>
6   <behavior name="Patrol">
7     <dependency>EnemyFound</dependency>
8   </behavior>
9   <behavior name="Attack">
10    <dependency>OddNumberOfEnemiesDestroyed</dependency>
11  </behavior>
12 </data>
13
```

כפי שאמרנו למעלה, כאשר רובוט מסוים מעדכן את מאגר המידע שלו, אם אין לו סיבה, הוא לא מדווח הלאה לשאר הרובוטים בקבוצה. לכן, כדי להגדיר לרובוט מה רלוונטי לרובוטים אחרים הגדרנו למשל שה-Behavior שאחראי על ההתקפה יקבל דיווחים על מספר האויבים שהושמדו. אנחנו יודעים שכל הרובוטים ביחד מפעילים את ה-Behavior של ההתקפה, ולאחר מכן מתבצעת אלוקציה לכל רובוט בנפרד שיתקיף מהצד שלו. הרובוט הראשון שמשמיד את האויב משנה את הערך של הפרדיקט הזה במאגר המידע שלו, אך סתם ככה שאר הרובוטים לא ידעו שמספר אי זוגי של אויבים הושמד. לכן, לאחר שהגדרנו שכל מי שמריץ את Attack מעוניין לדעת אם מספר זוגי של אויבים הושמד, כאשר רובוט מסוים מקבל עדכון של ערך זה, הוא מדווח לשאר הרובוטים וככה כולם ידעו מה הערך המעודכן של פרדיקט זה.

לבסוף, יש לספק את ה-Services שאנחנו משתמשים בהם לפי קובץ הקונפיגורציה. בדוגמא שלנו אנחנו צריכים לספק את ה-Services: allocate_all, allocate_separate, allocate_half ו-vote. נראה לדוגמא איך מימשנו את אחת ממתודות האלוקציה:

```
Allocate.py (~/.workspace/catkin/src/bite_sample/src) - gedit
dependencies.xml Allocate.py
7 class AllocateService:
8     def __init__(self, robotName):
9         self.robotName = robotName
10
11     # Providing the allocate service
12     rospy.Service(str.format('/bite/{0}/allocate_all', self.robotName), Allocate, self.allocateAll)
13     rospy.Service(str.format('/bite/{0}/allocate_separate', self.robotName), Allocate, self.allocateSeparate)
14     rospy.Service(str.format('/bite/{0}/allocate_half', self.robotName), Allocate, self.allocateHalf)
15
16     print 'Waiting for knowledgebase...'
17     knowledgeServiceName = str.format('/bite/{0}/get_knowledge', self.robotName)
18     rospy.wait_for_service(knowledgeServiceName)
19     self.getKnowledgeClient = rospy.ServiceProxy(knowledgeServiceName, GetKnowledge)
20     print 'Ready'
21
22 # The allocate all method
23 def allocateAll(self, req):
24     print str.format('Allocating nodes: {0} to team: {1}', req.nodes, req.currentTeam)
25     if req.nodes == []:
26         return AllocateResponse('', [], '')
27
28     node = req.nodes[0]
29     params = ''
30
31     return AllocateResponse(node, req.currentTeam, params)
```

אנחנו מראים רק חלק מקובץ זה. מה שחשוב לשים לב הוא שאנחנו מספקים את ה-Service שנקרא `allocate_all`, שכל מה שהוא עושה זה מחזיר ממערך ה-Nodes שהוא מקבל את ה-Node הראשון ואת כל הקבוצה שהתקבלה. כלומר, המתודה פשוט מחלקת לכל הקבוצה ביחד לעשות את המשימה הראשונה מבין המשימות. כעת, כדי להפעיל את האלגוריתם יש להפעיל קודם כל את ה-Nodes שמספקים את מתודות האלוקציה וההצבעה. לאחר מכן, יש להפעיל את שני ה-Nodes של BITE שהם ה-`KnowledgeBase.py` וה-`Node` הראשי – `BITE.py`. הפרמטרים שיש לשלוח ל-`KnowledgeBase` הם מה שם ה-Package שמריצים (במקרה שלנו `bite_sample`) ומה שם הרובוט שמריץ אותו. באופן דומה יש לשלוח פרמטרים אלה ל-BITE ובנוסף, יש לשלוח את שמות כל הרובוטים שמתחילים ביחד את התוכנית מופרדים עם פסיקים ביניהם. דוגמא לקובץ `launch` ניתן למצוא בקובץ הדוגמא המצורף.