

Autonóm rendszerek tesztfuttatásainak kiértékelése

Horányi Gergő, Dr. Majzik István
Méréstechnika és Információs Rendszerek Tanszék
Budapesti Műszaki és Gazdaságtudományi Egyetem
Budapest, Magyarország
horanyi.gergo@inf.mit.bme.hu, majzik@mit.bme.hu

Kivonat — Autonóm rendszerek robusztusságának és biztonságosságának tesztelése a mai napig sok kihívást jelent. Ezek a rendszerek a környezetükre reagálva önálló döntéseket hozhatnak. Ez a környezetfüggő autonóm működés megnehezíti mind a követelmények definiálását, mind ezek ellenőrzését. Munkánk során a környezetfüggő követelmények definiálásához egy új formalizmust hoztunk létre, amely alapján a tesztadatok automatikusan előállíthatók. Cikkünkben a tesztfuttatások eredményeinek kiértékelésére – azaz a követelményeknek való megfelelésének vizsgálatára – adunk hatékony megoldást.

Kulcsszavak — robusztussági tesztelés, tesztkiértékelés, tesztorákulum, környezetfüggő viselkedés

I. BEVEZETŐ

Az autonóm rendszerek képesek döntések meghozatalára és végrehajtására emberi beavatkozás nélkül [1]. Ezek a rendszerek – különös tekintettel a háztartási-, vagy a felügyeleti robotokra – sokszor valós, kontrollálatlan környezetben működnek, így nélkülözhetetlen, hogy elfogadhatóan reagáljanak a környezet váratlan eseményeire és változásaira is. Egy ilyen rendszernek robusztusnak kell lennie, hogy az előre nem látható szituációkat is kezelni tudja, valamint magas biztonsági elvárásoknak kell megfelelnie, hogy elkerülje emberi sérülések okozását. Ennek megfelelően a rendszerek verifikációjának és validációjának fontos részét képezik a robusztussági és biztonsági vizsgálatok.

Az autonóm robotok tesztelésével az R3-COP projekt [2] keretei közt foglalkoztunk, ahol egy olyan modellalapú tesztelési folyamatot dolgoztunk ki [3], ahol a követelmények alapján, keresés alapú tesztgenerálással állítunk elő olyan tesztkonfigurációkat, amelyek komplex, szélsőséges illetve extrém környezeti szituációkat jelentenek a tesztelt robot számára. A teszteket valós, vagy szimulált környezetben hajthatjuk végre: a tesztelt robotot a generált tesztet által meghatározott kezdeti környezetbe (kontextusba) helyezzük, majd autonóm működése közben rögzítjük a tesztelés közben történő bemenő eseményeket, a kimenő akciókat és a környezeti változásokat. A tesztkiértékelés feladata, hogy a naplózott tesztfuttatásokat megvizsgálja és megállapítsa, hogy alapvető biztonsági követelményeknek megfelelnek-e.

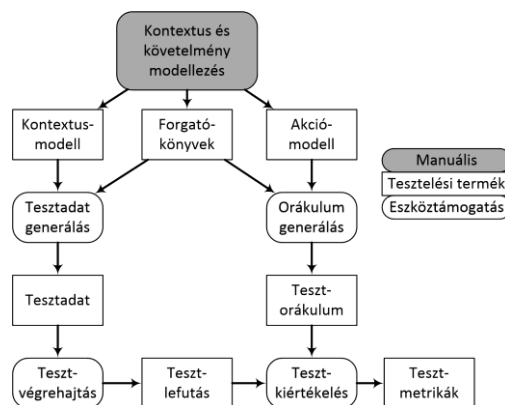
A hasonló kutatások elsősorban valószerű szimulátorokkal [8], kimerítő terepi teszteléssel [9] és a tesztelés fizikai aspektusával [10] kapcsolatosak. Csak néhány olyan keretrendszer létezik, amely automatikus tesztgenerálásra is képes. A megoldásunkhoz hasonló megközelítést mutat be [11]: metaheurisztikákat használó kereséssel online módon állítja elő a tesztteszteket. A mi megoldásunk offline tesztgenerálást és

tesztkiértékelést alkalmaz, amely egy flexibilis kontextus metamodellen alapul, így az alkalmazási területtől független módszer lehet.

A következő fejezetekben bemutatjuk a hatékony tesztkiértékelési megoldásunkat, amely a kontextusok vizsgálatára gráf alapú algoritmusokat, míg az események és akciók sorozatának elemzésére elfogadó automatákat használ.

II. A TESZTELÉSI KERETRENDSZER ÁTTEKINTÉSE

A tesztelési keretrendszer manuális és automatikus eszköztámogatással segített részfeladatokat foglal magába (1. ábra). Mivel az autonóm rendszerek (a továbbiakban robotok) viselkedése nem csak a kapott parancsoktól függ, hanem a robot környezete is sokban meghatározza azt, a kialakított keretrendszer komplex környezetek generálására törekszik a robusztuság és biztonságosság teszteléséhez.



1. ábra. Tesztelési keretrendszer áttekintése

A. Követelmények leírása

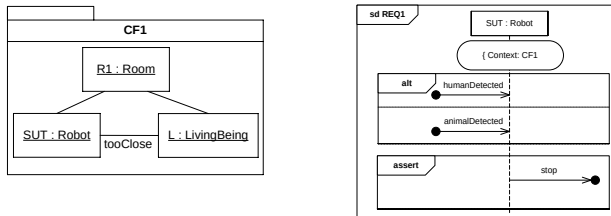
A tesztelési céljainkat követve egy-egy követelmény a robot kontextusfüggő viselkedésének alapvető biztonsági tulajdonságaira vonatkozik. Fontos, hogy a biztonsági követelmények tesztelése nem részletes viselkedést, csak biztonságos illetve veszélyes szituációk kialakulását (pl. ütközés), illetve a robot akcióit (pl. megállás) teszteli. Ennek megfelelően a követelmények leírása mind környezeti objektumokra, mind a robot akcióira hivatkozik.

Első lépésként a robot környezetét (kontextusát), az érzékelők által nyújtott bemeneti eseményeket és a beavatkozók által kiadott kimeneti akciókat modellezzük. Ezekből az információkból egy metamodellt építünk. A környezet metamodellje

azokat az objektumtípusokat tartalmazza, amelyeket a robot érzékelhet, illetve amelyekkel a robot a működése során valamilyen kapcsolatba kerülhet. Meghatározza, hogy ezek között milyen absztrakt relációk relevánsak a követelmények szempontjából (pl. a robot „közel van” egy emberhez). A környezeti modellből az egyes objektumok tulajdonságainak típusai is kiderülnek.

A teszt követelmények egy speciális grafikus leírónyelv segítségével adhatóak meg, amelynek részei a *környezeti (kontextus) leírások* és a *viselkedési leírások*. A követelmény meghatároz egy kezdeti kontextusmintát, ehhez kapcsolódó, előfeltételként tekintett eseményeket és kontextus változásokat, valamint meghatározza azokat a kialakuló kontextusokat illetve kiadott akciókat, amelyek biztonságosnak tekinthetők. A kontextusminták leírása a korábban definiált kontextus metamodellek megfelelő UML osztálydiagramokkal történik, míg a viselkedés (azaz események és akciók sorozatának) leírása UML szekvenciadiagramok segítségével valósul meg. A két diagramtípus közötti kapcsolatot a szekvenciadiagramok megfelelő helyein, a kontextust megadó osztálydiagramokra való hivatkozás jelenti, ahogy az a 2. ábrán is látható.

A szekvenciadiagram két részből áll. Az első rész a követelmény előfeltétele, míg a második rész azt a szituáció- vagy akciósorozatot írja le, amelyet az előfeltétel bekövetkezése esetén elvárunk a rendszertől. Ez utóbbi részt a diagram úgynevezett assert komponensével jelenítjük meg. A 2. ábrán szereplő példa követelmény azt a kezdeti kontextusmintát adja meg, mely szerint egy szobában a robot túl közel van egy élőlényhez. Ezután ha ember vagy állat detektálása történik, akkor a robotnak meg kell állnia.



2. ábra. Példa követelmények leírására

A leírónyelv lehetőséget ad dinamikus események megjelenítésére is, amennyiben a metamodelben ezek definiálva vannak. Ezeket a dinamikus eseményeket a kezdeti kontextusban időbélyeggel ellátva adhatjuk meg, például „2 másodperc múlva megjelenik egy ember”.

B. Tesztgenerálás

A követelményekben megjelenő kezdeti kontextus mintákat és a környezeti metamodelt felhasználva a tesztgenerálás során szisztematikusan előállíthatók a kezdeti kontextus minták kibővítései, ezek kombinációi, valamint az egyes környezeti objektumok tulajdonságainak és relációinak extrém értékei, így teszt esetként felvéve komplex környezeti konfigurációkat.

A tesztgenerálás metaheurisztikák segítségével [4], szisztematikus kereséssel történik. Az automatikusan előállított tesztesetek (mint környezeti konfigurációk) ezt követően valós vagy szimulált környezetben beállíthatóak, és a teszt futások

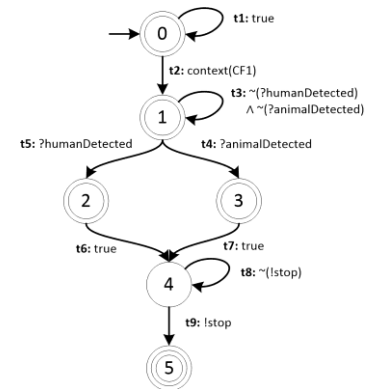
rögzíthetőek. A tesztgenerálás részletes bemutatása nem célja e cikknek, részletesebb leírás [4]-ben illetve [6]-ban található.

C. Tesztkiértékelés

A tesztkiértékelés során a rögzített teszt futásokat vizsgáljuk a rendszer követelményei alapján. A követelményleíró nyelvünket a TERMOS nyelv [5] inspirálta, így a tesztkiértékelési megoldásunk hasonlít annak kiértékeléséhez [12]. Jelentős különbség azonban, hogy esetünkben egyszerre több követelményt kellett hatékonyan vizsgálni, amelyhez egy új, dekompozíción alapuló megoldást adtunk (lásd IV. fejezet).

A követelmények kiértékeléséhez elsőként a követelményleíró nyelv operatív szemantikáját kellett definiálnunk. A szemantikát *elfogadó automaták* segítségével adjuk meg [6]: egy elfogadó automata azokat és csakis azokat a kontextusokból, eseményekből és akciókból álló sorozatokat reprezentálja, amik megfelelnek egy követelménynek. A 2. ábrán látható követelmény elfogadó automatája a 3. ábrán látható.

Az automata egy állapotátmeneten található őrfeltétel a követelmény által az adott előzmények után megengedett kontextust, eseményt vagy akciót ír le. Az automatában található hurokélek teszik lehetővé, hogy az ellenőrzés során a követelmény szempontjából irreleváns eseményeket átugorjunk. A *true* címkével ellátott élek atomi állapotokat jelölnek ki, azaz az őket megelőző állapotátmenetekkel együtt tüzelnek.



3. ábra. A 2. ábrán látható követelmény elfogadó automatája

A teszt futások kategorizálásához három állapot típust különböztetünk meg. Ha az automata *triviálisan elfogadó állapotban* (két körrel jelölve) áll meg, akkor a követelmény előfeltétel része nem valósult meg, így a követelmény se nem sérült, se nem teljesült. *Elutasító állapotok* (egyszeri körrel jelölve) azokat az állapotokat jelölik ki, amelyek a követelmény *assert* részéhez tartoznak, így ha az automata egy ilyen állapotban nem tud továbblépni, akkor a követelmény sérült. *Szigorúan elfogadó állapotok* jelentik meg, hogy az automata az előfeltételeket és a következményeket is sikeresen detektálta, így a követelmény teljesült.

III. A TESZTKIÉRTÉKELÉS NEHÉZSÉGEI ÉS MEGOLDÁSAI

A tesztelési stratégiáknak megfelelően a teszt futások minden egyes szegmensét minden egyes követelményhez viszonyítanunk kell, hiszen minden követelménynek minden (a

feltételrészében rögzített) esetben teljesülnie kell. Az előbbieken bemutatott elfogadó automaták *tesztorákulumból* szolgálnak (a tesztorákulum egy olyan komponens, amelynek feladata az elvárt és a tényleges teszteredmények összehasonlítása, kiértékelése [15]). A kiértékelés az elfogadó automaták futtatásából, azaz a tényleges tesztfutás során rögzített kontextusok, események és akciók, valamint az automata által reprezentált kontextusok, események és akciók illeszkedésének vizsgálatából áll. A következőekben a főbb kihívásokat, nehézségeket és az arra adott megoldásainkat mutatjuk be.

A. Minden követelmény folyamatos illesztése

Ahhoz, hogy bármelyik követelmény megsértését megtaláljuk, minden követelményt folyamatosan vizsgálunk. A sok kontextusvizsgálat rendkívül erőforrásigényes, így erre egy hatékony algoritmust adtunk. Az algoritmust arra optimalizáltuk, hogy egyszerre több követelmény kontextust legyen képes illeszteni a tesztfutásban található kontextusra. Ehhez a kontextusokat típusos gráfokra képeztük le, amelyekben a csúcsok a környezeti objektumokat illetve ezek osztályait, az élek pedig a köztük lévő relációkat jelentik. Az így képzett gráfokból egy *dekompozíciós struktúrát* építünk (Messmer és társainak ötlete nyomán [7]). A struktúra az összes kontextust összevonva, kompakt módon reprezentálja, mivel az azonos részgráfokat csak egyszer tárolja. Ez nemcsak a tárolás hatékonyságát növeli meg, de a keresések hatékonysága is többszörösére nő abban az esetben, ha a követelmények hasonló kontextusokat tartalmaznak. Ezt azonban feltehetjük, mivel a követelményekben egy adott környezet mintáit írjuk le.

B. Minden lehetséges követelménysértés kezelése

Az elfogadó automaták kontextust leíró állapotátmenetei bizonyos esetekben többféle módon illeszkedhetnek a tesztfutásban rögzített kontextusra – hiszen két kontextus között több megfeleltetés is található. Ilyen esetekben az összes lehetőséget vizsgálni kell, azaz ezekben az esetekben több automatapéldányt kell párhuzamosan futtatnunk – mindegyik megfeleltetéshez egyet-egyet.

C. Absztrakt relációk kezelése

A követelményekben az objektumok között absztrakt relációkat jeleníthetünk meg, míg a tesztfutás már konkrét értékeket tartalmaz. Az absztrakt relációk visszaállítását egy előfeldolgozó lépés keretei közt tesszük meg, amely az érvényes és releváns absztrakt relációkat azonosítja a tesztfutásban.

D. Típushierarchia illesztése

A környezet modellezése során típushierarchiát építhetünk fel. Az algoritmusnak ezt figyelembe kell vennie, azaz a kontextusok illesztése során a tesztfutásban található altípusból származó objektumot meg kell feleltetnie a követelményben szereplő őstípusnak. Ennek megoldására a pontos egyezés helyett a kontextusillesztés során egy kompatibilitási relációt használunk, amely figyelembe veszi a típushierarchiát.

E. Nemdeterminisztikus elfogadó automaták

A követelmények sok esetben alternatívákat is tartalmazhatnak a rendszer viselkedésére (ilyen a 2. ábrán található példa

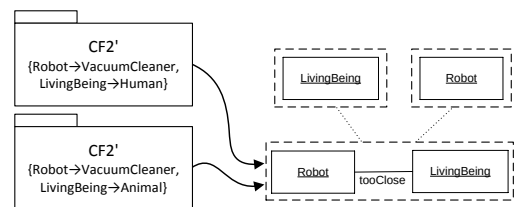
is). Elfogadó automata szintjén ez úgy jelenik meg, hogy egyszerre több állapotátmenet is engedélyezetté válik. A kiértékelés során minden lehetséges futást párhuzamosan vizsgálni kell.

F. Dinamikus események kezelése

A követelmények kezdeti kontextusában dinamikus események definiálhatók, amelyek konkrét időbélyegekhöz kötnék eseményeket. A követelmény többi része azonban nem tartalmaz előre definiált időbélyegeket, csak az események sorrendezését. Ezek az események akkor kapnak időbélyegeket, amikor a tesztfutás megfelelő eseményéhez hozzárendelődnek. Így derül ki az, hogy mikor válik szükségessé a dinamikus események beillesztése, azaz kiértékelés közben kell az illesztendő további kontextusokhoz hozzátenni az adott időponthoz megjelenő dinamikus eseményt.

IV. KONTEXTUSOK ILLESZTÉSE DEKOMPOZÍCIÓ SEGÍTSÉGÉVEL

Az algoritmusunk a kontextusokból gráfokat képez a könnyebb kezelhetőség érdekében. Ezt követően a [7]-ben található megoldáshoz hasonlóan egy dekompozíciós struktúrát épít, amelynek alapelve, hogy az azonos részgráfokat csak egyszer tárolja el. A [7]-ben található algoritmust úgy egészítettük ki, hogy a dekompozíció során a csúcsok típusát vesszük figyelembe és nem a behelyettesített tényleges csúcsot. Így lehetőség van arra, hogy különböző megfeleltetéseket együttesen kezeljünk, mivel a részgráfegyezőség vizsgálata még a behelyettesítés előtt történik meg, és a behelyettesítéseket egy külön struktúrában tároljuk (lásd 4. ábra).



4. ábra. Egyszerű kontextus dekompozíciója különböző megfeleltetésekkel

Az dekompozíciós algoritmus pszeudokódja az 5. ábrán olvasható. A struktúra építése során az egyes követelményekben szereplő kontextusokat egymás után adjuk hozzá a struktúrához. Az építés alapelve, hogy a legnagyobb már feldolgozott részgráfot megkeresi (4-6. sor), és arra már csak egy hivatkozást tárol. Az algoritmus a részgráfon kívüli maradékot rekurzív módon dolgozza fel (9-12. sor) és a szétbontás egészen addig történik, ameddig már csak egy, még tárolatlan csúcs nem lesz a gráfban (2-3. sor).

A struktúrán történő keresés elsőként a dekompozíciós struktúra egy-egy csúcsot tartalmazó csomópontjaitól indul. Elsőként előállítjuk az összes lehetséges illesztést a bemeneti gráfhoz, majd továbblépünk az összetettebb gráfokat tartalmazó csomópontok felé. Egy csomópont akkor dolgozható fel, ha mindkét részgráfját leíró csomópont már fel van dolgozva. Az összetett csomópontok feldolgozása esetén az egyes részgráf csúcsok lehetséges megfeleltetéseit megvizsgáljuk, majd azokból kiválasztjuk az összes ellentmondásmentes kombinációt.

Decomposition
input : G graph, V set of valuations, D decomposition output : the updated D decomposition
1 $S_{max} \leftarrow \emptyset$ 2 if $ V(G) = 1$ then 3 exit 4 foreach G_i <i>graph</i> $\in D$ do 5 if G_i <i>is subgraph of</i> G and $ V(S_{max}) < V(G_i) $ then 6 $S_{max} \leftarrow G_i$ 7 if S_{max} <i>is isomorphic to</i> G then 8 exit 9 if S_{max} <i>is empty and</i> $V(G) > 1$ then 10 $S_{max} \leftarrow$ random subgraph of G 11 Decomposition(S_{max}) 12 Decomposition($G - S_{max}$) 13 $E \leftarrow$ set of edges between S_{max} and $G - S_{max}$ in G 14 add $(G, S_{max}, G - S_{max}, E)$ to D and associate valuations

5. ábra: A dekompozíciós struktúrát építő algoritmus

A dekompozíció alapú megközelítés jelentős teljesítménynövekedést eredményez [7]: a keresés hatékonyabb, mint a klasszikus Ullman algoritmus esetében. Legjobb esetben a keresés költsége $O(IM)$, míg a legrosszabb esetben $O(N^M M^2)$, ahol N a gráfok számát, I az illesztendő (bemeneti) gráf csúcsainak számát illetve M a követelményekben található kontextusokat leíró gráf csúcsainak átlagos számát jelöli. A dekompozíciós struktúra építése legrosszabb esetben $O(N^2 M^{M+3})$ lépésszámmal.

V. AZ ELFOGADÓ AUTOMATÁK FUTTATÁSA

A tesztkiértékelés során elfogadó automatákat futtatunk, hogy a tesztlefutásban található kontextus, esemény és akció szekvenciákat összevegyük a követelményekben megengedetttekkel. Az ellenőrzéshez egy *futtatókörnyezetet* definiáltunk, amely az automaták példányait párhuzamosan futtatja. A futtatókörnyezet felel azért, hogy létrehozza a megfelelő példányokat és a futásukat felügyelje. A futtatókörnyezet automatákat futtató alapeljárásuk egy két lépést tartalmazó ciklusból áll.

Az úgynevezett *csendes lépésben* a futtatókörnyezet olyan független utakat keres az elfogadó automata aktuális állapotából, amelyek anélkül hajthatók végre (azaz illeszthetők), hogy a tesztlefutásból egy eseményt vagy akciót fel kellene dolgozni. Ilyen utak akkor jöhetnek létre, ha az út minden állapotátmeneten olyan kontextusok vannak, amelyek kompatibilisek a tesztlefutás pillanatnyi kontextusával. A futtatókörnyezet az összes lehetséges út mentén új automatapéldányokat hoz létre, amelyeket a továbbiakban párhuzamosan tovább ellenőriz. A futtatókörnyezet minden esetben a dekompozíciós gráfillesztő megoldást használja a kontextusok illesztéséhez.

Minden csendes lépést egy *eseményt vagy akciót illesztő lépés* követ, amely során az ellenőrizendő tesztlefutás következő bejegyzését vetjük össze az engedélyezett állapotátmenetekkel.

VI. ÖSSZEFOGLALÁS

Munkánk során autonóm rendszerek robusztussági és biztonságossági tesztelésével foglalkoztunk. Jelen cikkben a meg-

alkotott tesztelési keretrendszer tesztkiértékelő megoldásait mutattuk be. A komplex környezeti leírásokat is tartalmazó követelmények vizsgálata speciális gráfillesztő algoritmusok használatát igényli. Az általunk bemutatott algoritmus mellett, hogy egyszerre több gráf illesztését képes elvégezni, lehetőség ad típushierarchia, absztrakt relációk és dinamikus események definiálására is. A tesztkiértékelő megoldásainkat kidolgoztuk és implementáltuk. Pillanatnyilag a keretrendszerünk részletes validációjával foglalkozunk, amelynek során ROS (Robot Operating System) [13] alapú robot implementációkat tesztelünk egy Gazebo alapú szimulációs környezetben [14].

KÖSZÖNETNYILVÁNÍTÁS

Jelen munka az ARTEMIS JU és a Nemzeti Fejlesztési Ügynökség támogatásával jött létre, az R3-COP projekt keretei között. Az első szerzőt munkája során részben a Magyar Fejlesztési Bank támogatta a Habilitas Programon keresztül.

HIVATKOZÁSOK

- [1] Connelly, J., et. al. Hong, W., Mahoney, R., Sparrow, D.: Challenges in Autonomous System Development. In: Proc. of Performance Metrics for Intelligent Systems (2006)
- [2] R3-COP: Resilient Reasoning Robotic Cooperating Systems. ARTEMIS-2009-1 Project No. 100233, <http://www.r3-cop.eu/>
- [3] Micskei, Z., Szatmári, Z., Oláh, J., Majzik, I.: A Concept for Testing Robustness and Safety of the Context-Aware Behaviour of Autonomous Systems. In: Proc. 1st Int. Workshop on Trustworthy Multi-Agent Systems, Springer LNCS 7327, pp. 504–513, (2012)
- [4] Szatmári, Z., Oláh, J., Majzik, I.: Ontology-Based Test Data Generation Using Metaheuristics. In: Proc. 8th International Conference on Informatics in Control, Automation, and Robotics (ICINCO 2011), pp. 217–222, SciTePress (2011)
- [5] Waeselynck, H., Micskei, Z., Riviere, N., Hamvas, Á., Nitú, I.: TERMOS: a Formal Language for Scenarios in Mobile Computing Systems. In: Proc. 7th Int. Conf. on Mobile and Ubiquitous Systems, pp. 285–296 (2010)
- [6] Majzik, I., Micskei, Z., Oláh, J., Szatmári, Z.: Models, Languages and Coverage Criteria for Behaviour Testing of Individual Autonomous Systems. R3-COP deliverable D4.2.1 Part I, <http://www.r3-cop.eu/>.
- [7] Messmer, B. T., Bunke, H.: Efficient Subgraph Isomorphism Detection: A Decomposition Approach. IEEE Trans. on Knowledge and Data Engineering, 12:2, pp. 307–323 (2000)
- [8] Scrapper, C., Balakirsky, S., Messina, E.: MOAST and USAR-Sim: A Combined Framework for the Development and Testing of Autonomous Systems. Proc. SPIE 6230, (2006)
- [9] Kelly, A., et al.: Toward Reliable Off Road Autonomous Vehicles Operating in Challenging Environments. Journal of Robotics Research, 25:5-6, pp. 449–483 (2006)
- [10] Michelson, R. C.: Test and Evaluation for Fully Autonomous Micro Air Vehicles. In: ITEA Journal, 29.4, pp. 367–374 (2008)
- [11] Nguyen, C.D., Perini, A., Tonella, P., Miles, S., Harman, M., Luck, M.: Evolutionary Testing of Autonomous Software Agents. In: Proc. AAMAS (1) pp. 521–528, (2009)
- [12] Nguyen, M. D., Waeselynck, H., Riviere, N.: GraphSeq: A Graph Matching Tool for the Extraction of Mobility Patterns. In 3rd Int. Conf. on Software Testing, Verification and Validation, ICST 2010, pp. 195–204, IEEE Computer Society, (2010)
- [13] Robot Operating Systems (ROS) Wiki. <http://www.ros.org/wiki/>
- [14] Gazebo – Multi-Robot Simulator. <http://gazebo-sim.org/>
- [15] ISTQB Glossary version 2.2. <http://www.istqb.org>