MOVEC: Parametric Runtime Verification of C Programs

Zhe Chen / 陈哲

Nanjing University of Aeronautics and Astronautics
南京航空航天大学

Published in TACAS 2016

Outline

1 Introduction

2 A Simple Example

3 The Language for Monitors

4 Implementation of Monitoring

5 Experimental Evaluation

6 Conclusion

Introduction

- Runtime verification tools
  - Monitor program executions at runtime, to detect and possibly react to property violations.
  - Use automated program instrumentation for monitor synthesis and weaving.
  - Based on Aspect-Oriented Programming (AOP), modular implementation of crosscutting concerns.
  - Implemented as specification transformers, translating high-level monitor specifications into aspects.
- Java examples:
  - JavaMOP [Rosu and Chen et al.]
  - Tracematches [Avgustinov et al.]
  - Based on AspectJ

Introduction

- Apply runtime verification to C programs?
  - A large number of C applications, e.g., embedded software, avionics systems.
  - They usually require high dependability.
- However, how?
  - Need AOP tool support for C programs.
  - Based on AspectC++? AspectC? ACC?
  - Sadly, we lack AOP tool support.

Our Contribution

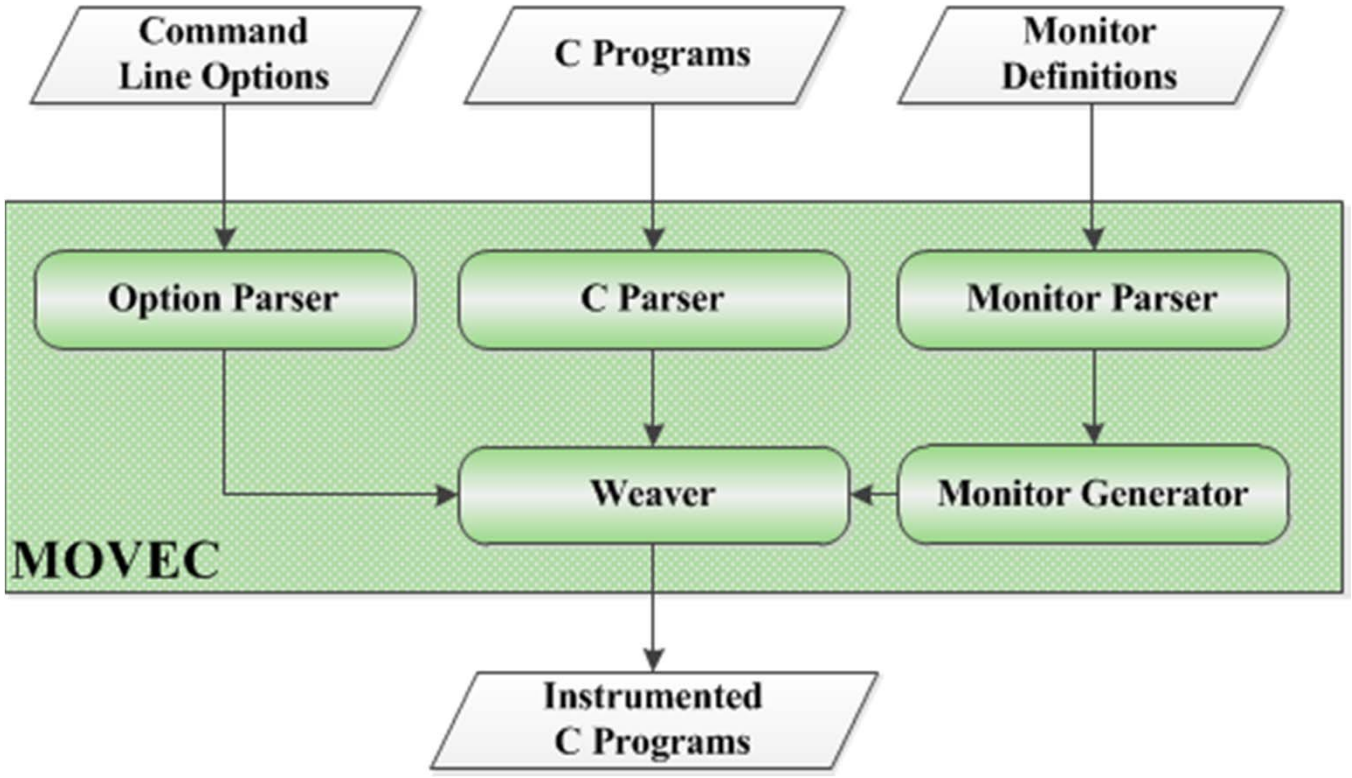
- Design a new general purpose and expressive language for defining monitors, as an extension to the C language.
- Propose a new instrumentation algorithm for the new language.
- MOVEC: an integrated tool implementation of the weaver.
  - AOP support
  - Parametric runtime verification
  - More robust, reliable and efficient

The MOVEC Compiler

- MOVEC: Runtime Monitoring, Verification and Control of C programs
  - An *aspect-oriented programming language*, inspired by related compilers like AspectJ, AspectC++ and ACC (AspeCt-oriented C)
  - A *monitoring-oriented programming language*, inspired by JavaMOP, TraceMatches etc.
  - An automated generator of runtime verifiers and controllers
- The long-term research objective is an industrial quality tool for monitoring C programs, especially targeting embedded software such as avionics systems.

The MOVEC Compiler

A Source-to-Source Transformer



Outline

1 Introduction

2 A Simple Example

3 The Language for Monitors

4 Implementation of Monitoring

5 Experimental Evaluation

6 Conclusion

Tool Demo

- C source program
  - malloc
  - free

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5    void *pointers[10];
6    unsigned size = 10;
7    int i;
8
9    for(i=0;i<10;i++)
10   {
11     pointers[i] = malloc(size);
12     size *= 2;
13   }
14
15   for(i=0;i<4;i++)
16   {
17     free(pointers[i]);
18   }
19
20   for(i=6;i<9;i++)
21   {
22     free(pointers[i]);
23   }
24
25   return 0;
26 }
```

Are all allocated memory blocks freed ?

Tool Demo

- Monitor definition for memory leakage

```
1  monitor mon(size_t size, void *address)
2  {
3    pointcut cm(s) = call(% malloc(% %:s));
4    pointcut cf(p) = call(% free(% %:p));
5
6    creation action malloc(address,size) after cm(size) && returning (address) {
7      printf("Allocated address %p–%p (size %lu) at line %d\n",
8             address, address+size, size, tjp->loc);
9    }
10
11   action free(address) after cf(address) {
12     printf("Freed address %p at line %d\n", address, tjp->loc);
13   }
14
15   action end after execution(% main(...));
16
17   ere: (malloc free)* malloc end;
18   @match {
19     printf("error: address %p (size %lu) was not correctly freed!\n",
20            __MONITOR->address, __MONITOR->size);
21   }
22 };
```

pointcuts

actions

properties

handlers

Tool Demo

- Run MOVEC

```
$ movec -m monitor1.mon -c malloc.c -d /home/user
$ cd /home/user/
$ gcc malloc.c -o a.out
$ ./a.out
... (omitted) ...
error: address 0x790ad0 (size 160) was not correctly freed!
error: address 0x790cf0 (size 320) was not correctly freed!
error: address 0x792590 (size 5120) was not correctly freed!
```

Generate files: malloc.c monitor.h hashmap.h

  - Created 10 monitor instances.
  - Only 3 monitors reached matching states, and invoked the handler.

Outline

1 Introduction

2 A Simple Example

3 The Language for Monitors

4 Implementation of Monitoring

5 Experimental Evaluation

6 Conclusion

The Language Model

- A monitor contains:
  - Join points
  - Pointcuts
  - Actions

  } AOP

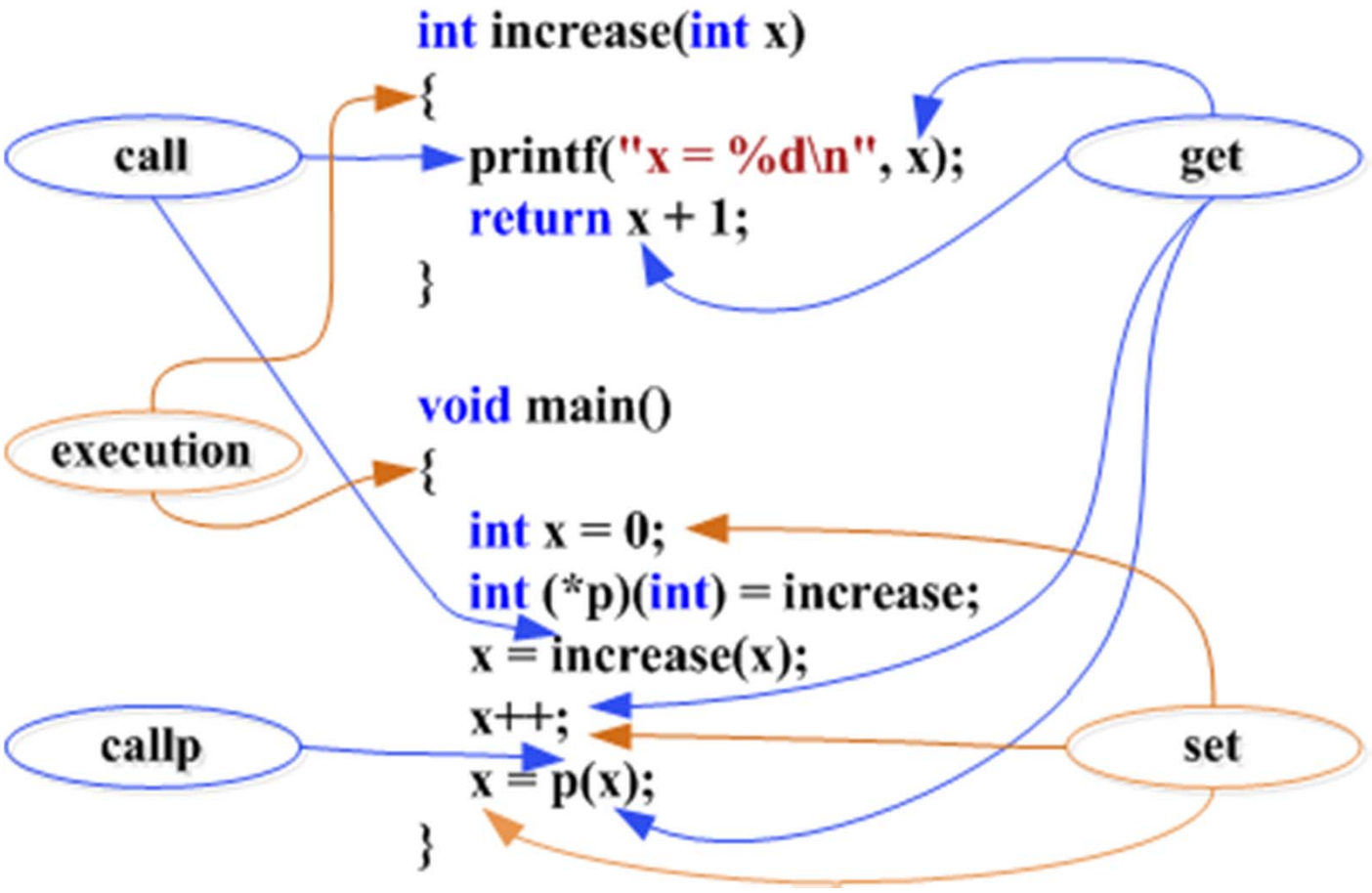
  - Properties
  - Handlers

  } Parametric runtime verification

Join Points

- A *join point* is a point in the execution of a program, such as
  - Call: function calls via function names
  - Callp: function calls via pointers
  - Execution: function executions
  - Set: setting the value of a variable
  - Get: getting the value of a variable
- Match join points using match expressions and pointcuts.

Join Points


int increase(int x)
{
printf("x = %d\n", x);
return x + 1;
}
void main()
{
int x = 0;
int (*p)(int) = increase;
x = increase(x);
x++;
x = p(x);
}
call
execution
callp
get
set

Match Expressions

- *Match expressions* describe a set of statically known program objects related to join points, such as identifiers, variable declarations, parameter lists and function signatures in programs.
  - Literal, e.g., ‘foo’, ‘int foo’, ‘char func(int foo)’
  - Regular, with ‘%’ and ‘...’ as wildcard characters, e.g., ‘% func%(..., int x, ...)’

Pointcuts

- A *pointcut* is an expression that matches a set of join points scattered in the execution of a program.
  - primitive pointcuts
  - composite pointcuts
  - named pointcuts

Primitive Pointcuts

- Primitive pointcuts contain four classes.
- 1. Core pointcut functions
  - call(function-signature)
  - callp(function-signature)
  - execution(function-signature)
  - set(variable-declaration)
  - get(variable-declaration)

Primitive Pointcuts

- 2. Naming pointcut functions
  - returning(identifier)
- 3. Dynamic scope pointcut functions
  - inexec(function-signature)
  - condition(boolean-expression)
- 4. Static scope pointcut functions
  - infunc(function-signature)
  - intype(identifier)
  - infile(identifier)

Composite Pointcuts

- A *composite pointcut* is a primitive pointcut, or a logical composition of composite pointcuts with the following operators: && (and), || (or), ! (not), and ().
- Examples:

```
call(int foo(char, int x)) && returning(ret);
execution(void bar(% p)) && inexec(% foo(...));
call(void func(char c, ...)) && infunc(int main(...));
```

Named Pointcuts

- A *named pointcut* assigns a name to a pointcut to reuse pointcut declarations.
- Examples:

```
pointcut npc1 = call(int foo(char, int x)) && returning(ret);
pointcut npc2 = execution(void bar(% p)) && inexec(%
    foo(...));
pointcut npc3 = call(void func(char c, ...)) && infunc(int
    main(...));
pointcut npca = npc1 || (npc2 && infile(a.c));
pointcut npcb = (npc2 || npc3) && infile(main.c);
```

Actions

- We can associate an *action* to a pointcut, i.e., a code fragment.
  - Dynamic actions
  - Static actions
- *The action code will be automatically executed* when a matched join point is reached in an execution.

Actions

- *Dynamic actions* are executed at runtime, before, around (instead of), or after the matched join points.

```
action before execution(void login(...)) {
    printf("logging in.\n");
}
```

- *Static actions* are executed at compile-time. to extend the source code.

```
action extend infunc(% foo%(...)) {
    printf("leaving function foo.\n");
}
```

Properties and Handlers

- A *property* specifies the desired or undesired set of sequences of matched join points in the execution of a program.
  - Finite State Machines (FSM)
  - Extended Regular Expressions (ERE)
  - Linear Temporal Logic (LTL)
- A *handler* can be automatically executed when the property is matched or violated by an execution of the program.
  - match, fail

Outline

1 Introduction

2 A Simple Example

3 The Language for Monitors

4 Implementation of Monitoring

5 Experimental Evaluation

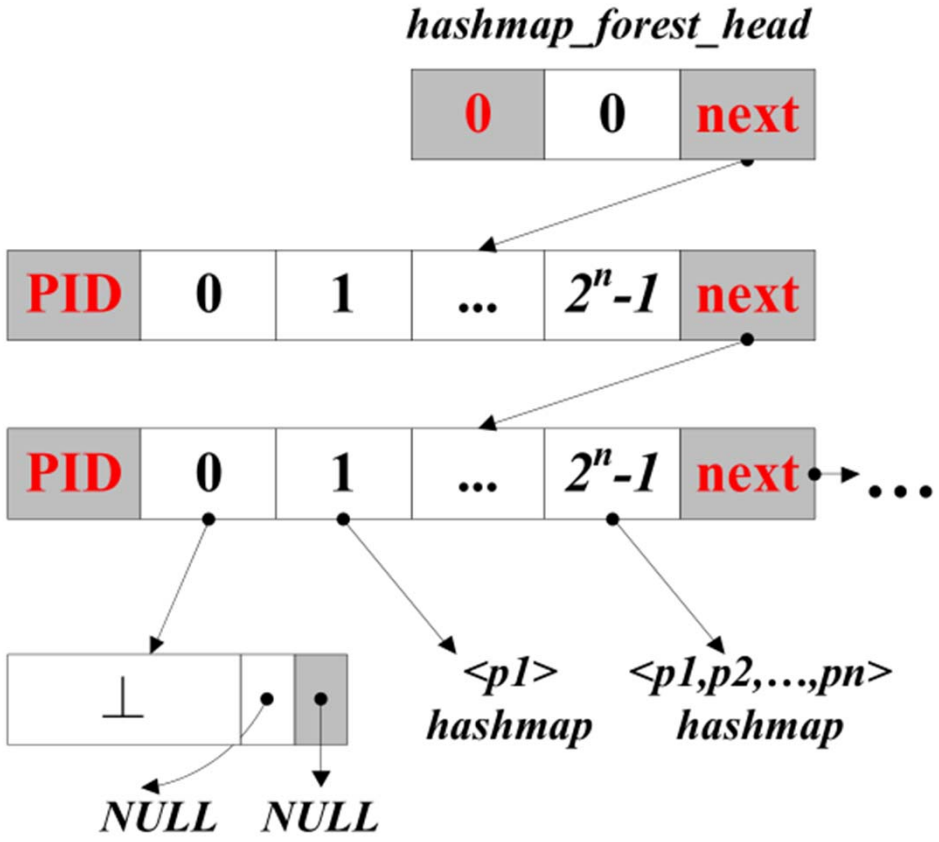
6 Conclusion

Data Structures

- During runtime monitoring, a program may create thousands of monitors.
- Developing an efficient algorithm for indexing these monitors.
- A new data structure: hierarchical hashmap forests
  - Each monitor is added into a hierarchical hashmap wrt. its parameter instance.
  - All monitors can be efficiently retrieved.

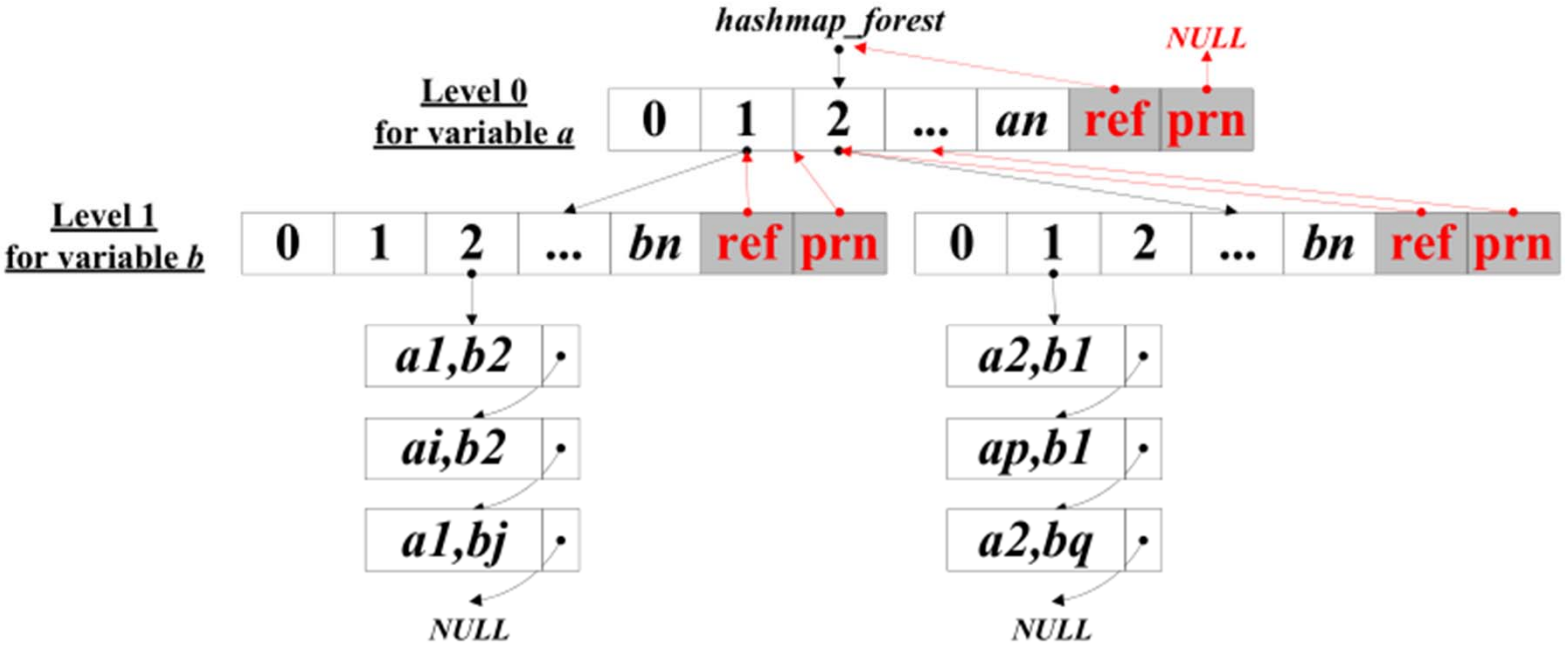
Hierarchical Hashmap

- A list of hierarchical hashmap forests



Hierarchical Hashmap

- A hierarchical hashmap corresponds to a set of parameters, e.g., {a,b}.
- Each level corresponds to a parameter.



Outline

1 Introduction

2 A Simple Example

3 The Language for Monitors

4 Implementation of Monitoring

5 Experimental Evaluation

6 Conclusion

MOVEC vs. JavaMOP

- Benchmark: four projects
  - Two equivalent versions, in C and Java, resp.
  - Two versions are literally similar.

		MOVEC				JavaMOP			
	mon. num	orig. time	hand. num	run time	time diff.	orig. time	hand. num	run time	time diff.
Enum	1000	0.016	1	0.199	0.183	0.114	1	0.218	**0.104**
Enum	20000	0.141	1	21.715	21.574	0.179	1	0.817	**0.638**
File	1000	0.144	1	0.145	**0.001**	0.232	1	0.334	0.102
File	20000	2.585	1	2.793	**0.208**	1.867	~~0~~	2.106	0.239
Grant	1000	0.006	500	0.030	**0.024**	0.102	500	0.205	0.103
Grant	20000	0.010	10000	12.397	12.387	0.110	~~9370~~	0.499	**0.389**
MapIter	1000	0.006	2	0.079	**0.073**	0.104	~~0~~	0.228	0.124
MapIter	20000	0.019	2	35.735	35.716	0.118	~~0~~	22.782	**22.664**

Results

- Correctness
  - MOVEC correctly invoked all handlers.
  - JavaMOP is incorrect in 3 projects, esp. when the number of monitors is large.
- Overhead
  - Generally comparable.
  - JavaMOP outperforms MOVEC when the number of monitors is large.

Outline

1 Introduction

2 A Simple Example

3 The Language for Monitors

4 Implementation of Monitoring

5 Experimental Evaluation

6 Conclusion

Conclusion & Future Work

Download MOVEC and examples/benchmarks:
http://svlab.nuaa.edu.cn/zchen/projects/movec

- We are open for suggestions on how to further optimize its features, language syntax and semantics.

Current and Future Work:

- Fine-tuning parts of the language design, e.g., adding more pointcuts and formalisms.
- Optimizing data structures and building the next generation compiler, to improve the quality.
- IDE extensions and documentation.
- Empirically study the practical value of MOVEC.

Our Related Work

Runtime Verification and Theory

- Zhe Chen, Zhemin Wang, Yunlong Zhu, Hongwei Xi, Zhibin Yang. Parametric Runtime Verification of C Programs. In *TACAS 2016*, *LNCS*, vol. 9636, pp. 299-315. Springer, 2016.
- Zhe Chen, Ou Wei, Zhiqiu Huang, Hongwei Xi. Formal Semantics of Runtime Monitoring, Verification, Enforcement and Control. In *TASE 2015*, pp. 63-70. IEEE Computer Society, 2015.
- Zhe Chen. Control Systems on Automata and Grammars. *The Computer Journal*, vol. 58(1), pp. 75-94. Oxford University Press, 2015.

Our Related Work

Model Checking

- Zhe Chen, Yi Gu, Zhiqiu Huang etc. Model Checking Aircraft Controller Software: A Case Study. *Software-Practice & Experience*, vol. 45(7), pp. 989-1017. Wiley, 2015.
- Zhe Chen, Daqiang Zhang and Yinxue Ma. Modeling and Analyzing the Convergence Property of the BGP Routing Protocol in SPIN. *Telecommunication Systems*, vol. 58(3), pp. 205-217. Springer, 2015.
- Zhe Chen, Daqiang Zhang, Rongbo Zhu etc. A Review of Automated Formal Verification of Ad Hoc Routing Protocols for Wireless Sensor Networks. *Sensor Letters*, vol. 11(5), pp. 752-764. American Scientific Publishers, 2013.

Thank You !

Questions?