# Parametric Runtime Verification of C Programs

Zhe Chen[1,2(✉)], Zhemin Wang[1], Yunlong Zhu[1], Hongwei Xi[3], and Zhibin Yang[1]

[1] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, 29 Jiangjun Avenue, Nanjing 211106, Jiangsu, China
{zhechen,wangzm,zhuyl,zhibinyang}@nuaa.edu.cn
[2] Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing, China
[3] Computer Science Department, Boston University, 111 Cummington Street, Boston, MA 02215, USA
hwxi@cs.bu.edu

**Abstract.** Many runtime verification tools are built based on Aspect-Oriented Programming (AOP) tools, most often AspectJ, a mature implementation of AOP for Java. Although already popular in the Java domain, there is few work on runtime verification of C programs via AOP, due to the lack of a solid language and tool support. In this paper, we propose a new general purpose and expressive language for defining monitors as an extension to the C language, and present our tool implementation of the weaver, the MOVEC compiler, which brings fully-fledged parametric runtime verification support into the C domain.

## 1 Introduction

Along with the popularity of runtime verification [16,19], many tools have been developed. These runtime verification tools automatically synthesize the code fragments of event extraction mechanisms and monitors from formal specifications, and then instrument the code into a target program, so that the monitors can extract information from the program executions at runtime, to detect and possibly react to observed behaviors satisfying or violating the specified properties. As automated program instrumentation plays a key role in monitor synthesis and weaving, many current tools are built based on Aspect-Oriented Programming (AOP), which is a programming paradigm that supports the modular implementation of crosscutting concerns [15].

By using AOP compilers, these tools are hence built in the form of specification transformers, that take an expressive high-level specification as input and produce output code written in some AOP language, most often AspectJ, a mature implementation of AOP for the Java programming language [14]. For example, among the large number of runtime verification tools, the most efficient parametric runtime verification tool JavaMOP [4,13,17] is based on AspectJ. JavaMOP transforms monitor definitions including desired properties into aspects, and then these aspects are transformed into Java code fragments and weaved into target programs using an AspectJ compiler. The desired properties can be automatically verified at runtime by running the executable file generated by AspectJ. Other tools like Tracematches [1,2] are designed in a similar way.


M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 299–315, 2016.
DOI: 10.1007/978-3-662-49674-9_17

Therefore, we believe that the popularity of runtime verification of Java programs is supported by the fact that a robust, reliable and efficient AOP compiler such as `ajc` is available.

Although already popular in the Java domain, there is few work on runtime verification of C programs via AOP, due to the lack of a solid language and tool support. For example, AspectC++ is an implementation of AOP for C++, but the generated code cannot be compiled by C compilers [20–23]. Coady et al. used "AspectC" (a hypothetical and simple subset of AspectJ) to modularize the implementation of prefetching within page fault handling in the FreeBSD OS kernel, and showed significant benefits [9,10]. But they used only a paper design for AspectC, supporting only join points of function calls and control flow, and no implementation of AspectC exists. ACC (AspeCt-oriented C) is the most advanced implementation of AOP for the C programming language at present [11], but it is currently not maintained by its developers, and the latest version is incorrect in many cases. For example, join points and pointcuts are sometimes not correctly matched, and instrumented code is possibly not semantically equivalent to its corresponding aspect. Worse, the ACC implementation is not well modularized, so fixing ACC is hard.

However, the fact is that a large number of applications is still being developed in C, especially embedded software applications such as avionics systems, which always require high dependability [7]. Thus, it is meaningful to provide a runtime verification tool or an AOP tool for the C language, so programmers can modularize the crosscutting concerns to improve maintainability, and based on AOP tools, they can also develop or use runtime verification tools to monitor and verify their programs at runtime.

In this paper, we propose a new general purpose and expressive language for defining monitors as an extension to the C language, and present our tool implementation of the weaver, the MOVEC compiler, which brings fully-fledged parametric runtime verification and AOP support into the C domain. The major contributions of our work include:

- We propose a new language for defining monitors for C programs by systematically redesigning the languages of AspectJ and JavaMOP. The main reason is that, the C language uses the procedure-oriented programming paradigm, which is very different from the object-oriented paradigm of Java, thus we have to redesign what we learned from AspectJ and JavaMOP according to the specific peculiarities of the C language. Another reason is that, the traditional AOP languages are somewhat conceptually confusing (the various types of pointcuts and advices are not systematic), not enough elegant and natural (some pointcuts and advices are written in a redundant and uncomfortable way).
- We develop a new instrumentation algorithm for the new language. In the AOP part, this is necessary because the philosophy of the C language is very different from Java, so we cannot implement aspects as classes like in AspectJ. Besides, the instrumentation algorithm of ACC, the most relevant AOP implementation, is incorrect in many cases. In the runtime verification

part, our algorithm has to implement more infrastructures than JavaMOP, because we cannot use the powerful Java class library, such as hashmaps.
– We implement an integrated tool supporting both AOP and parametric runtime verification. Getting AOP and runtime verification into the C language is a hard and tedious task, and our implementation supports all features of the new language and provides convenient user instructions. Experimental performance evaluation shows that our tool is robust, reliable and efficient.

This paper is organized as follows. Section 2 introduces the MOVEC compiler, including its software architecture, compilation process and theoretical foundations. Section 3 presents an example to show the tool's functionality, i.e., how to write monitor definitions and run MOVEC. Section 4 focuses on the design of our new language for defining monitors of C programs by introducing the semantics of each language element. Section 5 explains the tool implementation of the new language, including core data structures. Section 6 evaluates and compares the performance of MOVEC and related tools by presenting the experimental results on the same benchmark. We conclude and discuss future work in Sect. 7.

## 2 The MOVEC Compiler

MOVEC is an automated tool for runtime MOnitoring, VErification and Control of C programs as an extension to the C programming language. MOVEC is influenced by AOP and parametric runtime verification, and is an integrated implementation of these ideas for the C programming language. MOVEC aims at providing an infrastructure of AOP, runtime verification and related technologies in the context of software written in C, especially targeting embedded software such as avionics systems, leading to further explorations and investigations not possible today, as no reliable, efficient and stable implementation of these technologies for C programs exists.

MOVEC provides a source-to-source transformation that automatically weaves monitor specifications written in MOVEC into MOVEC-unaware C programs, and generates instrumented C programs which can be compiled by any compliant C compiler such as GCC and other platform-specific compilers. Note that MOVEC does not directly compile the instrumented C programs into a binary executable file, because many embedded platforms use their own C compilers which may be not compatible with each other. Thus, by using source code transformation, MOVEC can be used for all target platforms supported by C compilers.

**Software Architecture and Compilation Process.** The inputs of MOVEC are C programs and files containing monitor definitions, and the outputs are instrumented C programs. There are five major modules in MOVEC, corresponding to the five phases in the MOVEC compilation process: command line analysis (i.e., parsing the options given in a command line), parsing C programs, parsing monitor definitions, monitor generation (i.e., generating C code fragments for monitor definitions) and weaving (i.e., generating instrumented C programs).

**Theoretical Foundations.** Rosu and Chen et al. proposed the theoretical foundation of parametric runtime monitoring and verification [3,5,12,18], and implemented JavaMOP supporting parametric runtime verification of Java programs [4,13,17]. For the parametric runtime verification part, our tool implements their monitoring algorithm in the context of C programs. Our tool also implements a formal semantics of runtime monitoring, verification, enforcement and control [8], which is an instance of a more general computational model, namely control systems [6].

## 3 A Demonstration of the Tool

Generally speaking, MOVEC extends the C language with *monitor* definitions that implement crosscutting concerns in a modular way. A *monitor* definition is composed of declarations of *types*, *variables*, *pointcuts*, *actions*, *properties* and their *handlers*.

In this section, we will present a simple example to show how to write monitor definitions and run MOVEC. Suppose `malloc.c` is a C source program, which requests 10 blocks of memory from the heap by calling `malloc`, and then frees 7 of these blocks. Note that some blocks are not freed, resulting in memory leakage. We will show how to detect the memory leakage by defining monitors.

Let `monitor1.mon` be a monitor file containing the monitor named `mon` in Listing 1.1. This *parametric monitor* definition takes two parameters: `size` and `address`, and includes two parametric named pointcuts, three actions, a property and a handler. MOVEC creates a complete monitor instance for each observed value pair of `size` and `address`, both of which are specified in the creation action in this example (but not necessarily in other examples).

The first *parametric named pointcut* `cm(s)` refers to the function calls to `malloc`, and the parameter `s` binds the value of its actual argument. The second parametric named pointcut `cf(p)` refers to the function calls to `free`, and the parameter `p` binds the value of its actual argument. The symbol `%` is a wildcard character matching continuous strings of any length, e.g., any type name and any parameter identifier. The symbol `:` is a renaming operator that renames a parameter identifier to another one. The predefined pointcut `call` matches the join points of the function calls to the matched functions.

The first *parametric action* named `malloc` prints the address range of the allocated memory block, and is executed after any function call to `malloc`. The predefined pointcut `returning` assigns an identifier to the return value of the function call. The parameters `address` and `size` bind the address and size of the allocated block respectively, and the variable `tjp->loc` is a predefined variable which stores the line number of the function call. This action is also a `creation` action, which creates a new monitor instance. The second parametric action named `free` prints the address of the freed memory block, and is executed after any function call to `free`. The last action named `end` is executed after the execution of `main`. The symbol `...` is a wildcard character matching item lists of any length, e.g., any parameter list.

**Listing 1.1.** A parametric monitor

```
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) &&
7                                          returning (address) {
8      printf("Allocated address %p–%p (size %lu) at line %d\n",
9             address, address+size, size, tjp->loc);
10   }
11
12   action free(address) after cf(address) {
13     printf("Freed address %p at line %d\n", address, tjp->loc);
14   }
15
16   action end after execution(% main(...));
17
18   ere: (malloc free)* malloc end;
19   @match {
20     printf("error: address %p (size %lu) was not"
21            "correctly freed!\n", monitor->address, monitor->size);
22   }
23 };
```

The *property* over actions `malloc`, `free` and `end` is specified in extended regular expression (ERE). It matches undesired action sequences that start with zero or more `malloc` `free`, followed by a `malloc`, and end with `end`. The *handler* `@match` contains a code fragment that prints a message, which will be automatically executed when an execution of the program matches the property, i.e., a memory block was allocated, but was not correctly freed. The variable `monitor` is a predefined structure variable that refers to the current monitor instance, and its member variables `monitor->address` and `monitor->size` refer to the parameters `address` and `size` of the current monitor instance, respectively.

MOVEC takes monitor files and C header/source files as inputs, and outputs instrumented header/source files, which can be compiled into monitored programs by any compliant C compiler such as GCC. For example, the following command line takes the monitor file `monitor1.mon` and the C source file `malloc.c` as inputs, automatically weaves them together, and outputs the instrumented source file `malloc.c` to the destination directory `/home/user`.

```
$ movec -m monitor1.mon -c malloc.c -d /home/user
```

Besides the instrumented source file `malloc.c`, MOVEC also outputs two additional header files `monitor.h` and `hashmap.h` to this directory. The instrumented source file `malloc.c` can be compiled into an executable file `a.out` by GCC. Running `a.out` prints a list of messages, and the last three error messages indicates that 3 allocated memory blocks were not freed, along with their addresses and sizes (the addresses may be different on your computer).

```
... (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!
```

In this example, MOVEC created 10 complete monitor instances, i.e., one for each value pair of `size` and `address`. Then the handler was invoked for each one of the 3 monitors that reached matching states, whereas the other 7 monitors did not invoke the handler because they did not reach matching states. Thus, the result shows that there are 3 unmatched `malloc` actions at the end.

Note that the above example only demonstrated a small portion of the language and features of MOVEC. In the following sections, we will introduce in-depth the semantics of each language element.

## 4 The Language for Defining Monitors

### 4.1 Join Points, Pointcuts and Actions

We only briefly introduce these language elements, because these concepts stem from AOP languages, although with some improvements such as more systematic design of pointcuts and advices and more concise and comfortable syntax. The reader unfamiliar with these concepts may refer to the literature on AOP languages [11,14,20].

A *join point* is a point in the execution of a program, such as function calls via function names or pointers, function executions.

A *pointcut* is an expression that matches a set of *join points* scattered in the execution of a program. Currently, MOVEC supports *match expressions* for matching program objects such as identifiers, variable declarations and function signatures, and *primitive pointcuts*, *composite pointcuts* and *named pointcuts* for matching join points.

A *literal match expression* matches a program object only if they are exactly the same, whereas a *regular expression* can match a program object by using the symbols `%` and `...` as wildcard characters. For example, the expression `% func%(..., int x, ...)` matches any functions whose name starts with `func` and parameter list contains a parameter `int x`, but the return type and other parameters are left unspecified, e.g., `int* func1 (float foo, int x)`.

The predefined *primitive pointcuts* fall into four classes: *core* pointcut functions, *naming* pointcut functions, *dynamic scope* and *static scope* pointcut functions. The *core pointcut* functions include the following functions.

- `call`(*function-signature*) matches the join points of the function calls to the functions matched by *function-signature*. For example, the expression `call(% func%(..., in%, ...))` matches the function calls to any function whose name starts with `func`, parameter list contains a parameter whose type starts with `in`, but return type is left unspecified, e.g., `int* func1(float foo, int x)`.
- `callp`(*function-signature*) matches the join points of the function calls to the functions matched by *function-signature* via function pointers.
- `execution`(*function-signature*) matches the join points of executing the functions matched by *function-signature*.

The *naming pointcut* functions are used to assign names to some objects in the execution of a program, e.g., return values.

- `returning`(*identifier*) assigns an *identifier* to the return value of the function call matched by `call` or `callp` pointcuts, or to the return value of the function execution matched by `execution` pointcuts.

The *dynamic scope pointcut* functions are used to restrict the scope of matched join points at runtime.

- `inexec`(*function-signature*) matches the join points which are invoked during the dynamic execution of the functions matched by *function-signature*.
- `condition`(*boolean-expression*) matches the join points at which the condition specified by *boolean-expression* holds.

The *static scope pointcut* functions are used to restrict the scope of matched join points at compile-time.

- `infunc`(*function-signature*) matches the join points which statically appear in the function definitions matched by *function-signature*.
- `intype`(*identifier*) matches the join points which statically appear in the type definitions matched by *identifier*, such as structures, unions and enumerations.
- `infile`(*identifier*) matches the join points which statically appear in the files whose names are matched by *identifier*.

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

To reuse pointcut declarations, we can assign a name to a pointcut by declaring a *named pointcut*, then the named pointcut can be referred by using its name in any places where a pointcut can be used. For example,

```
pointcut ppc1(x,y) = call(int foo(int x)) && returning(y);
```

An *action* declaration associates a code fragment to a pointcut, and the code fragment will be automatically executed when a join point is reached in an execution of the monitored program, such that the join point is matched by the pointcut defined inside the action declaration. Actions are also called *advices* in AOP and *events* in JavaMOP. The syntax of action declarations is as follows.

```
[creation] ("action" | "advice" | "event")
    [ACTIONID ["(" <paramids-list> ")"] ]
("before" | "after" | "around") <pc-composite>
("{" <act-action> "}" | ";")
```

An action declaration specifies a *passive action* and an *active action*. The passive action contains a composite pointcut expression `pc-composite` to passively match reached join points, and specifies the position where the active action shall be triggered relative to the invocations of matched join points, e.g., `before`, `after` etc. The active action `act-action` is a code fragment enclosed

in curly braces. The active action is automatically executed if the passive action is matched.

For example, the following *parametric action* `pact` prints a message before the execution of function `foo`, which takes only one integer parameter `x`. Note that the parameter `x` is referred as a parameter of the action. As a result, the value of `x` can be accessed and printed in its active action.

```
action pact(x) before execution(int foo(int x)) {
  printf("before executing foo, x=%d\n",x);
}
```

### 4.2 Properties and Handlers

A *property* specifies the desired or undesired set of sequences of matched join points in the execution of a program, and a *handler* can be automatically executed when the property is matched or violated by an execution of the program. Currently, we can express properties using Finite State Machines (FSM) and Extended Regular Expressions (ERE).

An FSM includes a set of states, a set of actions and a set of transitions, in which one of the states is the initial state and a subset of the states is matching states (also called accepting states or final states). FSMs are also called Nondeterministic Finite Automata (NFA) in formal language theory. The syntax of FSM declarations is as follows.

```
"fsm" ":" ( STATEID1 "{"
            (ACTIONID "->" STATEID2 ";")*
            "}" )* ";"
```

An FSM declaration starts with the keyword `fsm` and a colon, possibly followed by a list of state declarations, and finally ends with a semicolon. A state declaration starts with its name `STATEID1`, followed by a list of transition declarations enclosed in curly braces. A transition declaration consists of an action name `ACTIONID`, the symbol `->`, a state name `STATEID2` and a semicolon in sequence, denoting that the action `ACTIONID` will transfer the FSM from state `STATEID1` to state `STATEID2`, where `STATEID1` and `STATEID2` could be either the same state or different states. If a state does not include a certain action, but the action appears in other states, then the action will transfer the FSM from the state to the implicit sink state, from which the FSM will never be matched. Note that the first declared state is the initial state, and the states whose name starts with `acc` are matching states.

For example, the following FSM declaration includes three states `q0`, `q1`, `acc1`, two actions `a`, `b` and six transitions, in which the first declared state `q0` is the initial state, and state `acc1` is a matching state. For each state, there are two transitions, e.g., state `q0` has a transition labeled action `a` from `q0` to `q1`.

```
fsm: q0 { a -> q1; b -> q2; }
     q1 { a -> q1; b -> q0; }
   acc1 { a -> q0; b -> acc1; };
```

An ERE is a sequence of identifiers and operators that defines a pattern to match sequences of identifiers. The operators in EREs include the concatenation of elements, the choice operator `|` which matches either the expression before or the expression after the operator, the asterisk operator `*` which matches the preceding element zero or more times, the plus operator `+` which matches the preceding element one or more times, the question mark `?` which matches the preceding element zero or one time, and the parentheses `()` which are used to define the scope and precedence of the operators. EREs can be translated into equivalent Nondeterministic Finite Automata (NFA) in formal language theory. The syntax of ERE declarations is as follows.

```
"ere" ":" <ere> ";"
```

An ERE declaration starts with the keyword `ere` and a colon, followed by an extended regular expression `<ere>` over action names, and finally ends with a semicolon. For example, the following ERE declaration over actions `malloc`, `set`, `get` and `free` matches the action sequences that start with `malloc`, followed by zero or more `set` and `get`, and end with `free`.

```
ere: malloc  (set | get)*  free;
```

A *handler* includes a category of property (e.g., match and violation) and an active action (i.e., a code fragment). A property can be associated with several handlers, so that an active action will be automatically executed when an execution of the program transfers the property to the corresponding category. Handlers can be used for many purposes, e.g., output or logging observed information, controlling, recovering, blocking or terminating the execution. The syntax of handler declarations is as follows.

```
"@" <cate> "{" <act-action> "}"
```

A handler starts with the symbol `@`, followed by a predefined category name `<cate>`, and finally ends with an active action `<act-action>` enclosed in curly braces. Note that different formalisms may have different sets of predefined categories, and the active action will be automatically executed when an execution of the program transfers the property to the category. Currently MOVEC provides two predefined categories `match` and `fail` for FSMs and EREs. The category `match` means that the associated property is matched by the execution, `fail` means that the property will never be matched by any extension of the execution.

### 4.3 Monitors

A *monitor* declaration collects multiple pointcuts, actions, properties and their handlers together, to implement crosscutting concerns in a modular way. A monitor declaration can also include additional type declarations and variable declarations. The syntax of monitor declarations is as follows.

```
<modifier>* ("monitor" | "aspect")
              MONITORID ["(" <param-list> ")"] "{"
  ( <C-type-decl>   | <C-var-decl>
  | <pointcut-decl> | <action-decl>
  | <property> <handler>* )*  "}" ";"
```

A monitor declaration starts with a list of modifiers, then specifies the signature of the monitor. The signature declaration starts with one of the keywords `monitor` or `aspect`, which can be used interchangeably. The keyword is followed by a name `MONITORID` and possibly an enclosed parameter list `param-list`. If the parameter list is given, then the parameter declarations should be separated by commas in the parentheses, and the monitor is called a *parametric monitor*.

Then the monitor declaration specifies the body of the monitor enclosed in curly braces, and a semicolon denotes the end of the declaration. In the declaration body, we can declare types, variables, pointcuts, actions, properties and their handlers. Note that,

- All declared types and variables will be instrumented as global declarations.
- At least one action declaration should be preceded by the keyword `creation`, denoting that observing this action should create a new monitor instance with different parameter values. If the monitor is parametric, then some of the action declarations must be parametric, such that the union of all action parameters is exactly the set of monitor parameters in `param-list`. That is, creation actions do not necessarily contain all monitor parameters.
- Each property should be specified in one of the supported formalisms, and can refer to the declared action names. Each property may be associated with zero or more handlers.
- The handlers can access the declared types and variables, and can access the predefined variable `monitor` which refers to the current monitor instance, through which we can access the monitor parameters in `param-list` of the current monitor instance.

## 5 Implementation of Parametric Monitoring

Recall that a monitor definition may contain a set of parameters, and MOVEC may create a *monitor* (instance) for each parameter instance containing the observed values of a subset of the parameters, to store the current state of each parameter instance. That is, a monitor or parameter instance may be complete or partial (i.e., containing a strict subset of the parameters). As the literature shows, a program may create thousands of monitors during runtime monitoring, thus storing these monitors using naive structures like linked lists or arrays will significant increase runtime overhead. Therefore, developing an efficient algorithm for indexing monitors is one of the most valuable and challenging parts in implementing parametric runtime monitoring.

Indeed, thanks to the indexing algorithm of JavaMOP, it becomes the most efficient parametric runtime verification tool at present. Our indexing algorithm

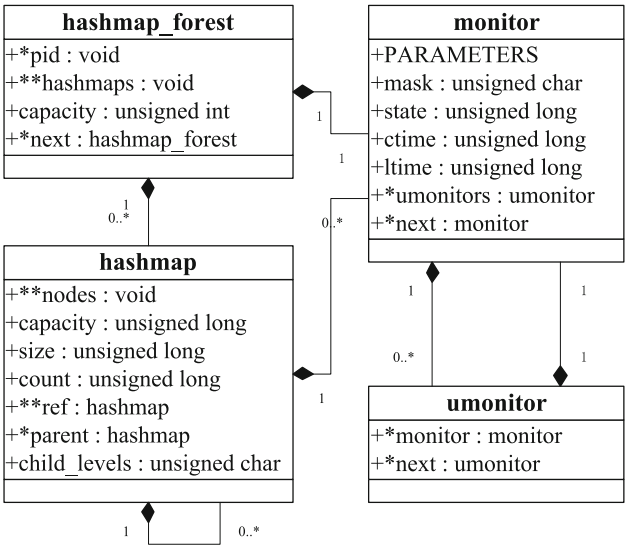


**Fig. 1.** Data structures of hierarchical hashmap forests

is inspired by JavaMOP, but our task is even more hard and tedious. We have to implement the data structures and related algorithms from scratch, because we cannot use the powerful Java class library, which includes efficient data structures such as hashmaps.

In this section, we present a new data structure, namely *hierarchical hashmap forests*, which is implemented in MOVEC for indexing monitors. Generally speaking, we maintain a list of hierarchical hashmap forests during runtime monitoring, and each created monitor is added into a hierarchical hashmap according to its corresponding property and parameter instance, so that all monitors can be efficiently retrieved. Figure 1 shows the data structures used by hierarchical hashmap forests. The `monitor` structure abstracts a monitor, including the values of `parameters`, a `mask` denoting the parameter instance, the current `state` etc. In the followings, we will present these structures from the top level.

As show in Fig. 2, we maintain a list of hierarchical hashmap forests during runtime monitoring. In the list, for each property `pid`, we create a node containing a forest of hierarchical hashmaps. The capacity of the forest depends on the number of parameters associated with the property. If a property includes $n$ parameters from $p_1$ to $p_n$, then there are $2^n$ hashmaps in the node, and each hashmap corresponds to a combination of the parameters. For example, the first location corresponds to the empty set of parameters, and the last one corresponds to the complete set. Note that the first location actually points to a monitor, instead of a hashmap, because there is only one parameter instance for this empty combination of parameters. Next we introduce the hierarchical hashmap for a set of parameters.

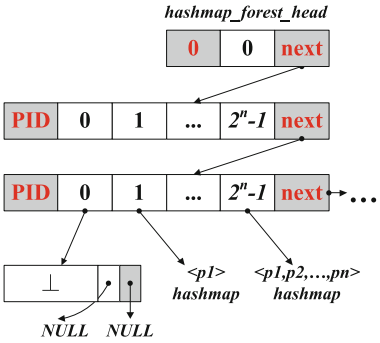


**Fig. 2.** A list of hierarchical hashmap forests

As shown in Fig. 3, a *hierarchical hashmap* is a multi-level hashmap, i.e., a hashmap may have several child hashmaps, like a tree. Note that a hierarchical hashmap corresponds to a set of parameters, thus each level corresponds to a parameter, and the last level points to the stored objects, i.e., monitors. To put these hashmaps in a tree, each hashmap contains not only the addresses of the next level hashmaps, but also a pointer *prn* to its parent and a pointer *ref* to the reference node in its parent hashmap, i.e., the pointer that refers to itself.

Recall that a hashmap maps keys to values, i.e., it uses a hash function to compute an index from which the desired value or object can be found, e.g., using modulo arithmetic. For our hierarchical hashmaps, we use parameter values as the keys for the corresponding level of hashmaps. Furthermore, we use linked lists to solve hash collisions.

For example, the hierarchical hashmap in Fig. 3 corresponds to the parameters $a$ and $b$, thus contains two levels. The first level is used to index the values of variable $a$, while the second to index the values of variable $b$. Each monitor stored in this hashmap corresponds to a parameter instance of $a$ and $b$. The monitor of the parameter instance $a_1b_2$ can be located via index 1 of the first

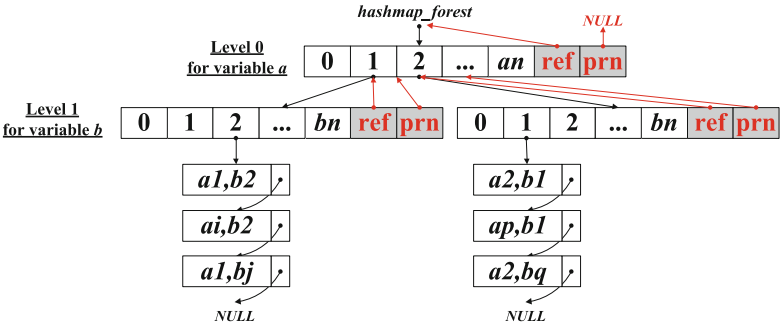


**Fig. 3.** A hierarchical hashmap

level, and then index 2 of the second level. Suppose parameter instances $a_i b_2$ and $a_1 b_j$ have the same index as $a_1 b_2$, then we put them in a linked list to solve the hash collisions. Similarly, the monitors of the parameter instances $a_2 b_1$, $a_p b_1$ and $a_2 b_q$ can be located via another path in the hierarchical hashmap.

## 6 Experimental Performance Evaluation

MOVEC uses a different weaving algorithm, compared with related tools. As JavaMOP and ACC are the most relevant and advanced tools in the runtime verification and AOP domains respectively, we compared the performance of MOVEC and the latest versions of JavaMOP and ACC on the same benchmark respectively. All experiments are done under the following platform: Intel i5-2410M CPU at 2.30 GHz, 4 GB memory, running Ubuntu 14.04 LTS 64-bit operating system.

MOVEC *vs. JavaMOP.* In our experiment, we designed a benchmark containing four projects. Note that MOVEC and JavaMOP can only process C and Java programs respectively, so each project is implemented as two equivalent versions, written in C and Java respectively, and these two versions are also very similar literally.

The first project `unsafe-Enum` creates a set of vectors, then creates an enumeration for each vector, and uses the enumeration to traverse the elements in the vector. But for one of these vectors, the vector is modified by adding an element while the enumeration is in use. A monitor with a regular expression property is designed to match the unsafe case where a vector with an associated enumeration is modified while the enumeration is in use. If the property is matched, a handler is invoked to print an error message.

The second project `unsafe-File` opens a set of files, then writes some strings into the files, and finally closes all files, except one. A monitor with a regular expression property is designed to match the unsafe case where a file was opened, but has not been closed until the program terminates. If the property is matched, a handler is invoked to increase a counter, and the count is printed when the program terminates.

The third project `unsafe-Grant` creates a set of tasks and a set of resources, then grants these resources to tasks, and finally these tasks release some of the granted resources, but not all. A monitor with a regular expression property is designed to match the unsafe case where a resource was granted to a task, but has not been released by the task until the program terminates. If the property is matched, a handler is invoked to increase a counter, and the count is printed when the program terminates.

The last project `unsafe-MapIterator` creates a map, then creates a set of collections for the map, creates an iterator for each collection, and adds an element to the map. But for two of these iterators, the iterators are used to get the next element in the collection, after the map is modified. A monitor with a regular expression property is designed to match the unsafe case where a map with an associated iterator is modified while the iterator is in use. If the

property is matched, a handler is invoked to print an error message. This offers a larger challenge, because the monitor creation actions do not contain all the parameters (collections are created before iterators).

**Table 1.** Experimental performance evaluation

| | | 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** |

Note: JavaMOP failed to correctly print the numbers of monitors and invoked handlers.

For each of the two versions of each project, we used two settings to generate different numbers of complete monitors. For each setting, we ran each version for three times, and measured in average the original run time (in seconds), the number of invoked handlers, the run time after instrumentation (in seconds) and the time difference. The data is listed in Table 1. Note that the two versions create the same number of complete monitors. Besides, JavaMOP failed to correctly print the numbers of monitors and invoked handlers, so we have to get the numbers by temporarily putting a `println` statement in the handlers.

The results show that MOVEC correctly invoked handlers for all projects, whereas JavaMOP failed to correctly invoke handlers in 3 projects (denoted by numbers with strikethrough lines), especially when the number of monitors is large. We considered two criteria of overhead: absolute time difference (i.e., the difference between the run time before and after instrumentation) and relative time difference (i.e., the ratio of the increased run time after instrumentation). Note that Java VM spends some time to load Java programs before execution, which is included in original run time but not in the difference, thus Java programs will benefit if we use relative time difference. In contrast, absolute time difference can avoid the effect of loading time. Indeed, absolute time difference is largely due to the algorithm for indexing and retrieving monitors, thus can more accurately reflect overhead. Hence, absolute time difference is an appropriate criterion for comparing their performance. According to this criterion, our algorithm is comparable with JavaMOP, because each tool succeeded in half of the runs. We also note that JavaMOP outperforms MOVEC when the number of

monitors is large. The reason probably is that JavaMOP uses the efficient data structures from Java class library, such as hashmaps, whereas our data structures and algorithms are less optimized.

MOVEC *vs. ACC.* For evaluating ACC, we have to use another benchmark, because ACC does not support parametric monitoring. In our experiment, we used ten projects from MiBench, a free and commercially representative embedded benchmark suite. We evaluated the performance of MOVEC and ACC by defining exactly equivalent monitors for each project, and of course in a different syntax. Due to page limit, we do not list the data here. The results show that the instrumentation time of MOVEC is less than ACC for all projects, and MOVEC significantly outperforms ACC in reliability (the results of ACC are incorrect for 7 projects, whereas MOVEC is correct for all projects according to our manual inspection) and efficiency (the overhead introduced by ACC is greater than MOVEC for all remaining 3 correctly executed projects of ACC).

## 7 Conclusion and Future Work

The main elements of the language design and compiler implementation are now fairly stable, but the project is not nearly finished. We are focusing on 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, performance and power of the compiler. We are also working on its IDE extensions and documentation. We want to build up and support a real user community of MOVEC, and plan to work with them to empirically study the practical value of MOVEC. We are open for suggestions how to further optimize the syntax and semantics. MOVEC and a set of working code examples/benchmarks are available for download from http://svlab.nuaa.edu.cn/zchen/projects/movec.

**Acknowledgement.** This work was supported by National Natural Science Foundation of China (61100034 and 61502231), Joint Research Funds of National Natural Science Foundation of China and Civil Aviation Administration of China (U1533130), Scientific Research Foundation for the Returned Overseas Chinese Scholars of State Education Ministry (2013) and Fundamental Research Funds for the Central Universities (NS2016092).

## References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L.J., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: Johnson, R.E., Gabriel, R.P. (eds.) Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), pp. 345–364. ACM (2005)
2. Avgustinov, P., Tibble, J., de Moor, O.: Making trace monitors feasible. In: Gabriel, R.P., Bacon, D.F., Lopes C.V., Steele G.L. (eds.) Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007), pp. 589–608. ACM (2007)

3. Chen, F., Meredith, P.O., Jin, D., Rosu, G.: Efficient formalism-independent monitoring of parametric properties. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), pp. 383–394. IEEE Computer Society (2009)
4. Chen, F., Rosu, G.: MOP: an efficient and generic runtime verification framework. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007), pp. 569–588. ACM (2007)
5. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 246–261. Springer, Heidelberg (2009)
6. Chen, Z.: Control systems on automata and grammars. Comput. J. **58**(1), 75–94 (2015)
7. Chen, Z., Gu, Y., Huang, Z., Zheng, J., Liu, C., Liu, Z.: Model checking aircraft controller software: a case study. Softw. Pract. Experience **45**(7), 989–1017 (2015)
8. Chen, Z., Wei, O., Huang, Z., Xi, H.: Formal semantics of runtime monitoring, verification, enforcement and control. In: Proceedings of the 9th International Symposium on Theoretical Aspects of Software Engineering (TASE 2015), pp. 63–70. IEEE Computer Society (2015)
9. Coady, Y., Kiczales, G., Feeley, M.J., Smolyn, G.: Using AspectC to improve the modularity of path-specific customization in operating system code. In: Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2001), pp. 88–98. ACM (2001)
10. Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N., Ong, J.S.: Structuring operating system aspects: using AOP to improve OS structure modularity. Commun. ACM **44**(10), 79–82 (2001)
11. Gong, W., Jacobsen, H.A.: Aspect-oriented C language specification. Working technical draft, University of Toronto, May 2010
12. Jin, D., Meredith, P.O., Griffith, D., Rosu, G.: Garbage collection for monitoring parametric properties. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), pp. 415–424. ACM (2011)
13. Jin, D., Meredith, P.O., Lee, C., Rosu, G.: JavaMOP: efficient parametric runtime monitoring framework. In: Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), pp. 1427–1430. IEEE (2012)
14. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
15. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
16. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebraic Program. **78**(5), 293–303 (2009)
17. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. Int. J. Softw. Tools Technol. Transf. (STTT) **14**(3), 249–289 (2012)
18. Rosu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Logical Methods Comput. Sci. **8**(1), 1–47 (2012)
19. RV: The Runtime Verification workshop series (2001–2015). http://www.runtime-verification.org/

20. Spinczyk, O.: AspectC++ language reference. Version 1.10, Pure-systems GmbH, October 2012
21. Spinczyk, O.: AspectC++ compiler manual. Version 1.7, Pure-systems GmbH, September 2013
22. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: AspectC++: an aspect-oriented extension to the C++ programming language. In: Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), pp. 53–60. Australian Computer Society (2002)
23. Spinczyk, O., Lohmann, D.: The design and implementation of AspectC++. Knowl. Based Syst. **20**(7), 636–651 (2007)