Aspect-Oriented Monitoring of C Programs

Klaus Havelund and Eric Van Wyk

***Abstract*— The paper presents current work on extending ASPECTC with state machines, resulting in a framework for aspect-oriented monitoring of C programs. Such a framework can be used for testing purposes, or it can be part of a fault protection strategy. The long term goal is to explore the synergy between the fields of runtime verification, focused on program monitoring, and aspect-oriented programming, focused on more general program development issues. The work is inspired by the observation that most work in this direction has been done for JAVA, partly due to the lack of easily accessible extensible compiler frameworks for C. The work is performed using the SILVER extensible attribute grammar compiler framework, in which C has been defined as a host language. Our work consists of extending C with ASPECTC, and subsequently to extend ASPECTC with state machines.**

I. INTRODUCTION

The work presented in this paper explores the synergy between two research topics: Runtime Verification (RV) and Aspect-Oriented Programming (AOP). The two fields turn out to meet in a "sweet spot" with the common objective of monitoring program executions against temporal properties.

Research in runtime verification attempts to develop technologies for *monitoring* the execution of software systems in order to detect violations of specifications stating the expected behavior [9], [5]. This includes our own work [3], [7]. Specifications are usually written as state machines, regular expressions, some form of temporal logic, or even as context-free grammars. From such a specification a monitor is generated, which driven by events changes state accordingly. The monitor will contain error states, which when entered cause some action to be executed. The reaction is typically the generation of an error message, or in some cases the execution of user provided code. Typically a monitored program is instrumented to drive the monitors. Such instrumentation is performed with various instrumentation packages, such as low level bytecode engineering tools for Java, or source code instrumentation tools like CIL [10] for C. Aspect-oriented programming frameworks have also been used, but typically only as the target of a translation from a runtime verification domain-specific language.

Research in aspect-oriented programming, on the other hand, attempts to develop technologies for *programming* software systems. This includes systems such as ASPECTJ [8] and ASPECTC [2]. The fundamental idea is that of an advice consisting of a pointcut and a reaction (a statement). The pointcut is in the simple case a predicate on program statements, and the reaction is meant to be executed when the application program to which the advice is applied reaches a statement during execution that satisfies the pointcut. This is achieved by weaving in the advice reaction at the right place in the code using an aspect compiler, which essentially performs a program instrumentation of the original program. A branch of research in the aspect-oriented programming community is concerned with enriching pointcut languages. This research includes investigating what in some literature is called *tracecuts* or *stateful aspects* [6], [15], [4], [14], [1], an extension of pointcuts which denote predicates on the execution trace. In this case an advice consists of a tracecut and a reaction, and the reaction gets executed when the tracecut becomes true on the execution trace seen so far.

Our most recent work includes the state machine based monitoring system RMOR [7] for C, which combines ideas from runtime verification and aspect-oriented programming. RMOR is in particular inspired by the graphical requirements capture language RCAT [11], [12]. However, RMOR is not an aspect-oriented programming framework. In this paper we propose to go all the way, and to extend ASPECTC with state machines, inspired by RMOR, resulting in the XSPEC language. Such a framework can be used for testing, fault protection, as well as for aspect-oriented software development. Our work is driven by practical situations within NASA's Jet Propulsion Laboratory (JPL), where state machines form a critical part of most embedded software applications, such as rover and space craft controllers.

II. REQUIREMENTS FOR REAL-WORLD IMPACT

The main goal of the presented work is to develop a framework for monitoring the execution of C programs against specifications. Our experience in building runtime verification frameworks and extensible languages leads us to believe that a successful strategy for ensuring that such a system will be widely used and can be constructed in a cost effective way is to satisfy the requirements described below.

a) Lexical Specification Language: The specification language must be lexical (text-based). The observation is that engineers are comfortable with lexical programming languages. A lexical language can have a graphical counterpart, and conversion to text, but there should also be a reverse mapping, such that programming can be performed in the lexical language. The specification language should be tightly integrated with the programming language.

K. Havelund is with Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA. klaus.havelund@jpl.nasa.gov

E. Van Wyk is with the Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA. evw@cs.umn.edu

This work is partially funded by NSF CAREER Award #0347860 and NSF CCF Award #0429640.

Part of the research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

b) State Machines: Engineers appear to favor state machines as a specification language. At JPL for example, state machines form an important part of programming tasks. At current times, such state machines are hand-coded in raw C. A domain-specific language for writing state machines seems like a natural extension of C. Such state machines should be usable for programming as well as for monitoring.

c) Temporal Logic: It should be possible for a user to define abstract temporal operators for use in monitoring, as shorthands for parameterized state machines.

d) Programs as Specifications: We believe that in the ultimate case the programmer should be allowed to write specifications that contain pure code. This may be useful in case the provided specification language does not suffice for expressing a given property. It should be possible to integrate this code with proper specifications in such a way that there is not a divisive distinction between the two. It is our belief that the programming language therefore should have specification-like features, such as notions of sets, lists and maps, and logic quantifiers (not discussed further).

e) Aspect-Oriented Programming: Program instrumentation is an important element of any runtime verification environment. The system to be monitored must be instrumented to drive the monitors. Such instrumentation should ideally be automated based on specifications of relevant events. Aspect-oriented programming offers a simple, but powerful language framework for expressing program instrumentation. We therefore believe that a runtime verification environment naturally can be expressed as an extension of an already existing AOP framework, such as ASPECTC.

f) Domain-Specific Monitoring Specification Languages: Different user communities may want different formalisms for specifying monitoring conditions. The difference here is primarily syntactic but such things do matter to real users. Thus, a successful framework must be extensible so that new specification language features can be easily introduced and used. This allows users to write specifications in the notation that they prefer, that is specific to the task at hand or domain of interest.

III. DESIGN OF XSPEC

The design of XSPEC[1] will be illustrated by first giving an example application program and some properties to be monitored. Then follows an example of how this would be done using ASPECTC as is. It is then illustrated what an RMOR specification [7] would look like, using state machines. Finally it is illustrated how these two language frameworks are combined into what we name XSPEC.

A. A file system example

Consider the following over-simplified C program containing functions for opening, processing and closing files identified by their name only (for simplicity):

```
void openfile(char *file);
void processfile(char *file);
void closefile(char *file);
int main {...}
```

Suppose we want to express and monitor the property that a file should be opened, processed and eventually closed, in that order. Furthermore, assume that when an already opened file is re-opened, the attempt should be logged, and that if the program terminates, all not yet closed files should be closed.

B. Programming a monitor in ASPECTC

A monitor for this problem in ASPECTC can be written as follows:

```
SetPtr set;

after(char *file):
  call(void openfile(char*)) &&
  args(file)
{
 if (set_contains(set,file)) log(file);
 else set = set_insert(set,file);
}

after(char *file):
  call(void processfile(char*)) &&
  args(file)
{
 if (!set_contains(set,file)) error();
}

after(char *file):
  call(void closefile(char*)) &&
  args(file)
{
 if (!set_contains(set,file)) error();
 else set = set_remove_element(set,file);
}

before(): call(void end())
{
 set_foreach(set,closefile_late);
}
```

The example illustrates four pieces of advice, one for each of the functions openfile, processfile and closefile, and one final one for a function end, which we assume is called at the end of the monitored application. The first advice catches all calls of the openfile function, using ASPECTC's pointcut language. It inserts the code in between the curly brackets { ... } after each such call. The code checks if the file is in the set of opened files, maintained for this purpose, and if it is, an error message is printed. Otherwise the file name is added to this set. Similarly for the other advice. The advice for the end function iterates through the set of names of open files remaining in the set and closes them. Although

[1]The name XSPEC is a play on (i) *x* for cross-cutting functionality as supported by aspect-oriented programming; (ii) *spec* (short for specification); and (iii) *expect* (expect your program to conform to certain properties).

the example is manageable, it is it is obvious that this form of programming can become complicated in cases when the logic becomes more complex. The next solution illustrates how the use of a state machine can simplify the specification task.

C. Programming a monitor in RMOR

In [7] we present the monitoring framework RMOR for monitoring C programs. The RMOR specification language allows us to state an approximation of the two properties as the following state machine:

```
handled monitor OpenClose{
  event open = after call(openfile);
  event process = after call(processfile);
  event close = after call(closefile);

  state FileClosed {
    when open -> FileOpen;
    when process => error;
    when close => error;
  }

  live state FileOpen {
    when open => error;
    when close -> FileClosed;
  }
}
```

The specification defines three events: open, process, and close, each defined as an event emitted after calls of the functions openfile, processfile and closefile respectively. The latter are specified via pointcuts written in a language equivalent to the pointcut language of aspect-oriented programming languages, specifically ASPECTJ [8]. The monitor defines a state machine, with two states FileClosed (the initial state) and FileOpen. In the FileClosed state an open event brings the state machine to the FileOpen state. Similarly, in the FileOpen state the event close brings the state machine back to the FileClosed state. The FileOpen state is *live*, meaning that the program is not allowed to terminate in this state, doing so causes an error. Finally, it is an error for a file to be processed or closed in the FileClosed state, and similarly, it is an error to open a file in the FileOpen state. Errors cause error messages to be printed, but can be handled by a user provided handler function.

Two observations should be made about the RMOR specification: (i) Events do not carry data values (file names) as does the ASPECTC model. This means that the RMOR monitor does not state the desired property. For example, the original ASPECTC model allows two different files to be open at the same time, whereas the RMOR specification does not. (ii) The state machine language is defined as a grammar independently of the C programming language grammar. RMOR is implemented in CIL [10], where the C grammar and parser is a closed system that cannot be extended easily. This means specifically that it is not possible to express the user defined error reactions (logging of file re-openings and closing of files at end) as part of the RMOR specification. Currently this is done by specifying that the monitor is *handled* (an RMOR keyword), and then provide a function *in the application program* with a specific signature, that handles the error cases and which RMOR calls.

D. Programming a monitor in XSPEC

The previous two specification attempts illustrate that a combination of the two notations may be attractive. XSPEC is therefore an extension of ASPECTC with state machines similar to those of RMOR. The specification in XSPEC becomes as follows.

```
xspec OpenClose(char *file) {
  pointcut open :
    call(void openfile(char*)) &&
    args(file);
  pointcut process :
    call(void processfile(char*)) &&
    args(file);
  pointcut close :
    call(void closefile(char*)) &&
    args(file);

  state FileClosed {
    after : open(file) -> FileOpen;
    after : process(file) => error;
    after : close(file) => error;
  }

  live state FileOpen {
    after : open(file) => error {
      log(file);
    }
    after : close(file) -> FileClosed;
    before : end {
      closefile(file);
    }
  }
}
```

We have introduced the notion of an *xspec* to define a state machine. In the tradition of ASPECTC, we can define pointcuts and advices. However, advices can now be grouped into states, the intention being that an advice is only active when the state is active. The whole specification is parameterized with a file, meaning that it is intended to track the behavior of a file. The intended semantics is similar to the semantics of Tracematches [1] in that we consider an xspec to denote an infinite set of monitors, one for each file as indicated by the parameter to the xspec. Each such monitor is tracking the behavior of a specific file. Obviously this semantics is abstract and not intended as an implementation strategy.

It is the intention to allow state machines to be defined independently of monitors. In this way a state machine can be used for programming as well as for monitoring. Another planned extension is the possibility of defining temporal operators on top of the state machine language. It is our

intention to allow a user to name a frequently used state machine pattern, parameterized with the events that drive the specification. This is similar to parameterizing aspects with pointcuts.

IV. Extensible XSPEC implementation

To implement XSPEC in a cost-effective way and to satisfy the "extensibility" requirement described in Section II, we will build XSPEC as a language extension using ABLEC—an extensible implementation of ANSI C based on attribute grammars. ABLEC is implemented using SILVER-an attribute grammar system with several features designed to support the specification of modular and composable language extensions. In using this approach, we build XSPEC in three layers. The first layer is ABLEC. Second, ASPECTC [2] is defined as an extension to C. Finally XSPEC is defined as an extension to ASPECTC. The first layer, ABLEC, has been implemented in a beta-version. The second layer, ASPECTC, is under construction. The third layer, XSPEC, is under design.

A. SILVER *attribute grammars and* ABLEC

SILVER has previously been used to build ABLEJ [13], an extensible implementation of Java 1.4. Several modular and composable language extensions have been implemented in this framework. For example, one extends Java with SQL so that database queries can be typed in a natural syntax and can be checked for syntax and type errors at compile time. Another extension adds dimension analysis to the type system so that, for example, a length measurement in feet is not incorrectly added to a volume measurement in meters.

In attribute grammars, the context-free grammar that defines the abstract syntax of the language is enhanced by associating attributes with nonterminals in the grammars. The values of these attributes are specified by declarative rules (attribute assignments) associated with productions in the grammars. These rules implement the desired semantic analysis, *e.g.* type checking, of the language. ABLEC is an extensible specification of ANSI C implemented as an attribute grammar written in SILVER. To understand how attribute grammars are used in the extensible implementation of ABLEC, consider the production below for function definitions.

```
abstract production func_def
fd::FuncDef ::= ds::DclSpecs dc::Dcl
                body::CStmt
{
 fd.errors = ds.errors ++ dc.errors ++
             body.errors;
 body.env = appendEnv(dc.defs,
      appendEnv(dc.param_defs, fd.env));
 ...
}
```

This abbreviated production indicates that a function definition (`FuncDef` nonterminal) named `fd` consists of declaration specifiers (`DclSpecs`) named `ds`, a declarator (`Dcl`) named `dc` and a compound statement (`CStmt`) function body named `body`). It defines the *synthesized* `errors` attribute on `fd` from the `errors` on the children, and defines the *inherited* `env` attribute on `body` to pass a symbol table down the AST for type checking. Many details are omitted, but it suffices to say that ANSI C can be implemented in this manner. ABLEC is such an implementation that performs type checking, but does not generate machine code. It instead outputs ANSI C code to be compiled by a traditional compiler.

Attribute grammar fragments defining language extensions can be added to ABLEC. A mechanism called *forwarding* can be used by productions defining new language constructs to specify their translation into semantically equivalent ANSI C constructs. Note that issues related to parsing extensible languages are also dealt with in SILVER, see [13] for details.

B. The ASPECTC *extension*

The attribute grammar that defines the ASPECTC extension is based on the specification in [2]. As an example, consider the advice declaration below:

```
after(char *file):
   call(void openfile(char*)) &&
   args(file)
{...}
```

Part of the SILVER specification of ASPECTC that defines the abstract syntax of the advice construct can be seen below:

```
abstract production advice
adv::FuncDef ::= dc::Dcl pc::Pointcuts
                 cs::CStmt
{
 dc.env = adv.env;
 pc.env = appendEnv(dc.param_defs,
                    adv.env);
 cs.env = appendEnv(dc.param_defs,
                    adv.env);
}
```

In C grammar terminology, it consists of a declarator "`after(char *file)`", a single pointcut "`call(void openfile(char*)) && args(file)`", and a compound statement "`{...}`". Such an advice definition is regarded as a special case of a C function definition and adds a new alternative to the `FuncDef` nonterminal.

The declarator environment inherits the environment coming in from above. The environment passed to the pointcut as well as to the compound statement is the environment produced by the parameter list of the declarator, which includes the inherited top level environment. Type checking and other static checks are performed based on these environments. The ASPECTC specification does not use forwarding to translate ASPECTC code to ANSI C; instead, it will type check and then output ASPECTC code to be compiled by an existing ASPECTC compiler.

C. The XSPEC extension

The XSPEC extension will be constructed following the pattern described above, which has been used successfully in other extensible language implementations such as ABLEJ. XSPEC extends our implementation of ASPECTC. It differs from the ASPECTC extension in that it will use forwarding to translate XSPEC code to semantically equivalent ASPECTC code that will perform the weaving that implements the code instrumentation. Note that the XSPEC example from Section III-D will most likely not translate to the ASPECTC example from Section III-B but to a more direct implementation of the state machine. An advantage of using attribute grammars with forwarding is that the productions defining the XSPEC extensions will define attributes to do semantic analysis such as type checking at the XSPEC level. In addition, domain-specific analyses, such as checking for un-reachable states, are now possible. Such analysis and more helpful error messages can be generated in this way. This analysis and error messages over XSPEC specifications are not possible if all semantic analysis is performed at the ASPECTC level.

V. CONCLUSION AND FUTURE WORK

We have presented current work on implementing an extensible definition of ASPECTC in the SILVER extensible compiler framework. We have furthermore discussed the design of an extension of ASPECTC with state machines, in order to facilitate convenient programming of monitors. It is important to emphasize that even though our emphasis is runtime verification, we do believe that this topic can leverage the popularity of aspect-oriented programming. XSPEC can be used as a general purpose aspect-oriented programming language as well as for runtime verification purposes. The framework should in addition allow user-defined temporal operators to be defined as parameterized state machines. An approach currently under investigation is to use an even stronger underlying rule-based framework [3], in which state machines and other logics can be defined.

REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittamplan, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*. ACM Press, 2005.

[2] AspectC. `http://research.msrg.utoronto.ca/ACC`.

[3] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for runtime monitoring: from Eagle to RuleR. In *7th International Workshop on Runtime Verification (RV'07), Vancouver, Canada*, LNCS. Springer, March 2007.

[4] C. Bockisch, M. Mezini, and K. Ostermann. Quantifying over dynamic properties of program execution. In *2nd Dynamic Aspects Workshop (DAW05), Technical Report 05.01*, pages 71–75. Research Institute for Advanced Computer Science, 2005.

[5] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, 2007.

[6] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Sgura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with Arachne. In *Proceedings of the 4th international conference on Aspect-oriented software development*, Chicago, USA, Mar. 2005. ACM Press.

[7] K. Havelund. Runtime verification of C programs. In *1st Int. TESTCOM/FATES Conference on Testing of Communicating Systems (TESTCOM) and Formal Approaches to Testing of Software (FATES)*, *LNCS*. Springer Verlag, June 2008.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *European Conference on Object-oriented Programming*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[9] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a runtime assurance tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier Science, 2001.

[10] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of Conference on Compiler Construction*, 2002.

[11] M. Smith. Requirements for the demonstration version of the Requirements CApture Tool (RCAT). JPL/RSS Technical Report, RSS Document Number: ESS-02-001, 2005.

[12] M. Smith and K. Havelund. Requirements capture with RCAT. In *16th IEEE Int. Requirements Engineering Conference*. September 2008.

[13] E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for Java. In *European Conference on Object Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 575–599. Springer Verlag, July 2007.

[14] W. Vanderperren, D. Suvé, M. A. Cibrán, and B. D. Fraine. Stateful aspects in JAsCo. In *4th International Workshop on Software Composition, ETAPS 2005*, volume 3628 of *LNCS*. Springer, 2005.

[15] R. Walker and K. Viggers. Implementing protocols via declarative event patterns. In R. Taylor and M. Dwyer, editors, *ACM Sigsoft 12th International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159–169. ACM Press, 2004.