Trace Server: A Tool for Storing, Querying and Analyzing Execution Traces

Igor Andjelkovic
University of Belgrade
Belgrade, Serbia
igor.andjelkovic@etf.rs

Cyrille Artho
RCIS/AIST,
Tsukuba, Japan
c.artho@aist.go.jp

***Abstract*—Various techniques for software verification are in use today, including testing and software model checking. Each of them has certain limitations, imposed by limited memory and computation time. This limits the types of properties that can be analyzed during one execution on a given computer.**

By carrying out additional analysis of program traces outside the execution of the program, one can extend the scope of the analysis. This paper presents the *Trace Server,* a solution for collecting, storing, querying and processing data describing program execution traces. The work is implemented as an extension of the Java PathFinder model checking tool. The collected data can be saved in a database for further processing, or be processed during the operation of the system. Data can also be sent to a remote server.

The tool defines the interface for creating data analyzers and includes examples of its use, providing a deadlock analyzer and an analyzer of executed methods. A developer using our tool can create new reports or supplement existing data.

***Index Terms*—Software verification; software execution traces; trace analysis.**

I. Introduction

Model checking can be seen as a kind of dynamic analysis, which is executed against the model of the program. The model of the system consists of an initial state, a set of program states, and transitions between states. Model checking traverses the state space in order to find reachable states that violate the properties of the system [11]. The big challenge with this approach is the state space explosion problem. Various techniques are used to alleviate this problem, such as system abstraction or partial-order reduction.

Java PathFinder (JPF) [20] is a model checking tool for Java programs using an explicit representation of the program state. It uses the program as a model, in the form of Java bytecode instructions. JPF implements a Java virtual machine that is able to execute all possible paths of the program (in accordance with the described general limitations of all tools).

A. Generating, storing, and analyzing execution traces

An execution *trace* is defined as a sequence of events that represent the important moments in the execution of the program (instructions executed, a thread being blocked, etc.). When detecting property violations, trace information can generate the path that led to this state, aiding in discovering the cause of a disturbance. If an exhaustive search is not feasible, incomplete trace information may give clues to possible system behaviors. Trace generation can be done in several ways [4]: (1) by code instrumentation, (2) by defining wrappers or (3) by customized execution environment. This paper describes a way to store data by extending JPF, which can be classified as (3).

Saving execution traces in memory is a significant problem. The verification of a large software can result in sequences of several billion instructions. If these execution traces are to be kept in memory for further analysis, they compete with the memory needs for storing the search space during model checking. A possible solution to this problem is to preserve the results of all choices (choice generators in JPF), while discarding detailed execution trace data. After execution, if more detailed information are needed, it is necessary to restart the analysis and execute all choices again. This solution is not ideal, as it may be difficult to faithfully recreate the results of system calls or interactions with external processes.

Another problem is caused by storing trace data and analyzing it adequately. It is difficult to store arbitrary data about an executed program. Currently it is possible to generate a report in readable form with information on traces, but the implementation of the report is specific to a particular platform and cannot be easily extended. Analysis of traces collected that way, after verification, requires parsing the report.

Many analyzers currently implemented in JPF (deadlock analyzer, methods analyzer and others) use their own data structures to store trace data. In addition to that, they must implement mechanisms to maintain execution paths, due to state space exploration. Much of this code is common to all analyzers and can be aggregated. From the above it follows that there is a large overhead for the development of an analysis algorithm. Furthermore, such an analysis can be carried out only while the JPF is executed, not post-mortem.

B. Architecture of our solution

Our tool, the *Trace Server,* is implemented as an extension to Java PathFinder (JPF). JPF does not support persistent trace data, but provides a flexible mechanism for notifications on the internal state of the program being checked. Our aim was to use that mechanism and develop a tool that manages trace data, and provide an infrastructure for storage and processing. Our tool can store trace events in memory or in a database, or it can forward them to a remote computer.

There is also a need for examining the collected data (by placing a query) through a well-defined interface. Trace processing can be conducted during or after verification. The tool enables the creation of analyzers, units that analyze properties of the executed program. Our work includes the implementation of several analyzers. The provided infrastructure enables the creation of analyzers with much less code. A powerful reporting system is also implemented, which allows an easy creation of custom reports.

C. *Paper outline*

This paper is organized as follows: Section II gives more background on model checking and JPF, and lists related work. Section III describes the architecture of our tool, and Section IV gives an evaluation based on several algorithms we have implemented. Section V concludes the paper and outlines future work.

II. Background

A. *Model checking*

Model checking analyzes a representation of a system (model), to determine the validity of properties of interest [11], [18]. Traditionally, model checking has been applied to descriptions of programs or algorithms. Recently, *software model checkers* that directly verify the program have been developed [5], [6], [9], [12], [14], [20]. This dispenses with the need of creating a special representation of the program in the form of the model.

If a property violation is found during verification, the tool should generate a *counterexample* trace, which represents a sequence of execution that leads to a property violation. Software model checking tools provide insight into the current state of the program being checked, enabling one to gather information about the trace.

B. *Java PathFinder (JPF)*

Java PathFinder (JPF) [19], [20] provides an executive environment for test and verification of Java programs. It is implemented in the Java programming language, with emphasis on scalability and configurability. Many modules that can be connected to the core of JPF expand the range of possible uses of this tool.

The JPF core represents a virtual machine (VM) that interprets Java bytecode. Speed is not the main feature of JPF because represents a virtual machine running on top of the original Java virtual machine (JVM).

JPF identifies non-deterministic choices in a program, and systematically covers all outcomes of these choices. Choice points can be scheduling-related, or assignments of a set of values to a variable. JPF is especially suitable for the verification of concurrent programs because it allows traversing all execution paths caused by thread scheduling.

JPF is able to detect various defects in a system. By default, it checks for deadlocks and unhandled exceptions, including assertion violations. In addition to that, JPF provides a mechanism for defining arbitrary properties to check. If a property violation is found, JPF generates a counterexample.

C. *Related work*

TraceContract [7] implements an API for trace analysis in the Scala programming language. Its expressive specification notation specifies trace properties to be checked, in form of a hybrid between state machines and temporal logic. TraceContract can be used for analyzing log files or for monitoring systems executing online. However, trace data cannot be collected and stored persistently. Furthermore, TraceContract does not separate report generation from property analysis.

Java PathExplorer (JPaX) [15] is a runtime verification tool for monitoring the execution of Java programs. Only one execution path is observed and trace data are collected by code instrumentation. A fixed set of events is written to a file or to a socket, in plain text format, unlike our Trace Server which stores a configurable set of events in a database. Like in our tool, trace analysis algorithms are customizable. JPaX contains specialized trace analysis algorithms for deadlock and data race analysis. These algorithms take a single (defect-free) execution trace and try to conclude the presence or the absence of deadlocks and data races in other potential traces of the program [8], [15].

Another concurrency analysis tool, jPredictor [10], detects possible property violations based on a technique called sliced causality. The program under test is instrumented to log partial information of its execution. The log is then post-processed to construct a more informative trace using static analysis on the original program.

The SPIN [17], BLAST [16], and MoonWalker [2] model checking tools, generate data on performance only when they detect a violation of characteristics of the system. Unlike them, JPF Trace Server generates data on all paths executed during the test, whether the error occurred or not. SPIN can work as a simulator, execute only one program path and report the executed code sequence.

To our knowledge, compared to our tool, no other model checking tool has implemented the option of such detailed data collection and reporting on the executed program.

III. Architecture

A. *Event and trace analysis inside JPF*

The *listener* mechanism provided by JPF allows external modules to observe and even influence program execution, by interacting with JPF. Listeners can be added at run time; JPF and its listeners are executed on the same virtual machine.

JPF notifies registered listeners by using the observer design pattern [13]. Notifications concern specific events during the search (e. g., search start/end, search backtracked) and execution of JPF's virtual machine (e. g., a new thread is started, method or instruction executed) . With each notification, complete information about the internal state of JPF and program that is checked can be obtained.

The *reporting system* of JPF consists of several modules:

- Reporter: a data collector. Coordinates the work of other parts of the system of reporting.

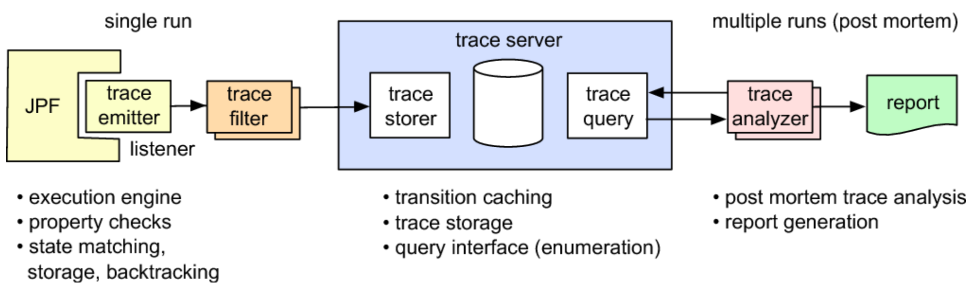

Figure 1. Trace Server architecture.

- Publisher: responsible for managing, formatting and printing the data collected. Examples of formats are text, HTML, XML.
- PublisherExtension: registered for each separate Publisher, for reporting on certain specific topics that define the Publisher.

The default Publisher is ConsolePublisher, which generates a human-readable report in text form. JPF allows the registration of any number of reporters.

B. Trace Server

The architecture of the Trace Server comprises several modules, as shown in Figure 1.

1) Trace emitter and filter: The trace *emitter* is an implementation of the JPF Listener interface. It listens to events broadcasted by JPF, creates *Event* objects that describe these events, and forwards them to a trace filter. Event objects contain all the information about the event; events are encoded using Java's primitive types and strings. The data (event properties) in the Event object are stored as <key, value> pairs, where the key is of type *PropertyID*. A special key type is used instead of a string to allow efficient transfer of information to a remote server.

In our system there can be only one trace emitter. To listen to any number of events, it is necessary to extend the base emitter by overriding the appropriate methods. For every event there is a special method. Certain methods cannot be overridden as they are necessary to maintain the data structure containing information about the executed program.

The trace *filter* receives events and may reject or change events. The filter forwards its results to another filter, or to the trace storer. This stage in the processing of events is introduced in order to allow the use of existing emitters for the purpose of on-the-fly processing and multiplexing. The filter is capable of processing events in the order in which they are received. Processing may consist of testing certain properties, but a filter can also change events by adding, changing or removing properties, before it forwards it to the next filter. Filters implement a chain of responsibility design pattern [13]. If multiple filters are present, their execution order depends on the order specified in a configuration file. Events are removed from the filter chain by not forwarding them to the next stage. Events are already preprocessed into the internal format that the trace server uses, so events are filtered by their value attribute, as shown by an example in Figure 2. A real filter should keep more events, as a meaningful analysis requires richer information. More code examples and information on the implementation and API can be found online [3].

```
public class OnlyNewInsnFilter extends TraceFilter {
  public void processInstructionExecuted(Event ev) {
    String insnOpcode = (String)
      ev.getProperty(PropertyCollection.INSTRUCTION_OPCODE);
    if (insnOpcode.equals("new") {
      forward(ev, eventType.instructionExecuted);
} } }
```

Figure 2. A filter that only keeps events on executions of NEW instructions.

2) Trace storage: The central part of the system is referred to as the trace server in Figure 1. Its role is to accept events (trace storer), store them in a database and provide an interface to query the stored content (trace query).

The trace *storer* provides a database-independent interface for storage, as defined by configuration parameters. The trace storer stores all the events received from the filter.

Three types of storage are implemented: *neo4j*, *inMemory* and *remote*. As stated earlier, the JPF search state space forms a graph, with nodes representing states and edges representing transitions. A relational database is not ideal to store this kind of data. Therefore, we chose a specific type of database, which uses a graph structure with nodes, links, and associated properties to present and preserve information.

These requirements led us to choose the *neo4j* database [1], which stores graph data persistently. This storage module realizes the initial idea of developing the Trace Server, to store data outside JPF in a persistent way. Unfortunately, neo4j does not perform very well in terms of computation time and storage space. The reason is that neo4j is designed as a transactional database. This creates unnecessary overhead if nodes are never updated or deleted. Furthermore, transactions have to be committed periodically to allow queries to see the updated data set. Empirically, we currently commit a transaction after the creation of every 100,000 nodes.

The *inMemory* storage module stores trace data temporarily, using a non-transactional database. It takes up less memory and runs faster than neo4j.

Finally, the *remote* storage interface forwards events to another host. On the remote side, a server receives events from the client (where JPF is executing). The client sends all event properties to the server, including their type (PropertyID). To store trace data, the server uses one of the two solutions already described, neo4j or inMemory.

3) Query interface, analysis: The trace *query* module allows database-independent access to trace data and provides a query interface that uses the search mechanisms offered by the different underlying databases. A search can be implemented as an iteration over events, or as a predicate. A predicate is defined as a boolean function that takes an event, and decides if that event should be included in the result set. In a query, the predicate is evaluated on each graph node.

The main reason for the implementation of JPF Trace Server is to create a common infrastructure for analyzing trace of programs executed by JPF. The trace *analyzer* should query the database, perform some analysis and report the results. Multiple types of analysis can be combined. The *analyze* method provides a uniform interface to execute analyses. Thanks to the trace server, there is no need to parse information obtained from the database. For optimization, it is useful to use an appropriate trace emitter that does not broadcast all the events possible, to save time and space. This means that analyzers need to be compatible with the data sets stored by the emitter and filters. If multiple analyzers can work with the same set of data, or a set is general enough, then this problem does not exist and is not necessary to restart the model checking tool to collect new trace data.

After data analysis, it is necessary to show the results in human-readable form. We have seen that the analyzers can define specific reports, characteristic for a particular type of analysis. Since it is not possible to use JPF's reporting system, as data are collected in a completely different way, the need for a custom reporting mechanism arises. The *report* subsystem is implemented to generate general reports, which can still be extended by specific information in predefined places. One can print individual events, as well as the complete execution sequence.

Reports can be displayed on the console, stored in a text file, or viewed in a JPF shell panel, called trace report. Results are grouped into topics that can be individually examined in the same panel.

C. Example report

Figure 3 shows an example report generated by an extended report, which compared to the default report adds information about the location where the object is initialized. The specific implementation needs to process data relating to all NEW instructions to find out where every instance is created. This data is gathered by an analyzer, which processes the execution trace prior to printing it.

IV. Evaluation

A. Analysis algorithms

Our current JPF Trace Server implementation includes three analyzers [3]:

1) DeadlockAnalyzer: analyzes deadlocks by observing thread interactions.

```
...
instructionExecuted
  oldclassic.java:130
    SecondTask.run()V
      invokevirtual Event.wait_for_event()V
objectLocked
  291 # LEvent; # init at: oldclassic.java:48
instructionExecuted
  oldclassic.java:79
    Event.wait_for_event()V
      aload_0
objectUnlocked
  291 # LEvent; # init at: oldclassic.java:48
objectWait
  291 # LEvent; # init at: oldclassic.java:48
```

Figure 3. Example report that shows information about the location where the object is initialized.

Table I
Savings in code size for different analyzers.

Analyzer module	Code size		Ratio
	old version	with Trace Server	old/new
DeadlockAnalyzer	306	160	1.91
MethodAnalyzer	212	86	2.47
OverlappingMethodA.	113	112	1.01

2) MethodAnalyzer: shows method calls (type of call, transition, thread).
3) OverlappingMethodAnalyzer: specialized MethodAnalyzer that searches for overlapping method calls on the same object from different threads. Such overlapping calls indicate a data race.

To evaluate our claim that the trace server provides an advantage when writing analyzer algorithms, we compared how many lines of code are needed to implement an analyzer with and without the Trace Server. The results are shown in Table I; in two out of three cases, we managed to reduce the amount of code by about half. It is noteworthy that the Overlapping-MethodAnalyzer extends MethodAnalyzer and thus share code for storing trace data and maintaining execution paths; that shared count amounts to 17 lines in the old version and 29 lines in the analyzer using the trace server.

In addition to the analyzers described, more examples are designed that collect and print various types of events and features of the program during execution (method calls with arguments, location where the object is created, etc.). These augmented reports improve the understanding of the counterexample traces, and also give examples on how reports generated the trace server can be augmented with specific data. The distribution of the trace server contains an example where any string written to the console is also logged; this augments the execution trace with information that appeared on screen during program execution [3].

B. Experiments

For measuring the performance of the Trace Server, we have used the examples from the JPF distribution. To run all experiments, the Oracle Java Virtual Machine, version

Table II
EXPERIMENTAL RESULTS.

Program	Size (LOC)	Instr. executed	Execution time [sec]			
			JPF	inMem.	neo4j	remote
Crossing	121	50,490	1.6	2.5	22.6	3.5
DiningPhil	26	81,752	3.1	5.0	64.0	7.0
oldclassic	47	3,466	1.0	1.1	5.2	1.4
Prod./Cons.	53	120,373	5.5	9.3	563.4	11.2
TreeMap	32	183,728	2.0	4.0	71.0	6.9

1.6.0_21 is used, running on Windows XP Professional on a Pentium D820 processor with 2GB of RAM available. Table II summarizes the results.

For all programs, events about executed instructions are stored, with default information, and events that are required to maintain the structure of the graph in the database. The inMemory solution that stores data in memory gives the best performance with a slowdown of 1.05 – 1.97 times. In this case, data is not stored persistently; persistency can be achieve by sending data to a remote server. Sending data to a remote computer results in an overhead of 1.38 – 3.38 times. Neo4j requires much time to generate parts of the graph and to execute transactions, making it an order of magnitude slower than other solutions (a slowdown between 5.05 and 104 times). If neo4j is combined with remote storage, its performance is acceptable, at least in our examples, where the remote server was not saturated with incoming data.

V. Conclusions and Future Work

This paper presents the Trace Server, a solution for storing, querying and processing the data describing the execution of a program begin verified. The work is implemented as an extension of Java PathFinder model checking tool and is publicly available as a supplement to the core tool [3]. Data can be stored in memory or in a database, or can be sent to a remote server; data access is independent of the underlying storage. Queries to the trace database can be made during execution or post mortem, by one or multiple analysis algorithms. Support for report generation facilitates the development of new types of property verification. Reports can be extended with arbitrary data. The Trace Server is implemented as a modular system, so it is flexible and extensible.

The modular architecture of the trace server allows us to improve its components one at a time, for example, to address possible scalability issues in the trace storage. Other possibilities for improvement exist in the analysis and reporting system. It is difficult to define a universal way of displaying trace data. What is feasible is to show different levels of detail. This could be achieved by a graphical interface. Furthermore, Java is not necessarily an ideal language to implement queries and reports; we are considering the use of a domain-specific language for this purpose. Finally, given other front ends (listeners collecting execution data on environments other than JPF), the trace server could become a generic platform for runtime verification, allowing verification algorithms to be written for any platform that the trace server supports.

Acknowledgments

The authors would like to thank Peter Mehlitz for all the great comments and ideas, and Darko Marinov for discussion about the tool. This work was started under the Google Summer of Code (GSoC) program, during summer 2010.

References

[1] Neo4j open source nosql graph database. http://neo4j.org/, 2011.
[2] N. Aan de Brugh, V. Nguyen, and T. Ruys. Moonwalker: Verifcation of .net programs. In S. Kowalewski and A. Philippou, editors, *Proc. 15th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, volume 5505 of *LNCS*, pages 170–173, Berlin, 2009. Springer.
[3] I. Andjelkovic and C. Artho. Trace Server. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-trace-server, 2011.
[4] C. Artho. *Combining Static and Dynamic Analysis to Find Multi-threading Faults Beyond Data Races*. PhD thesis, ETH Zürich, 2005.
[5] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. 16th Int. Conf. on Computer Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.
[6] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 268–285, Genova, Italy, 2001. Springer.
[7] H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *Proc. 17th Int. Conf. on Formal Methods (FM 2011)*, pages 57–72, 2011.
[8] S. Bensalem and K. Havelund. Dynamic Deadlock Analysis of Multi-Threaded Programs. In *Proc. Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2005)*, volume 3875 of *LNCS*, pages 208–223, Haifa, Israel, 2005. Springer.
[9] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.
[10] F. Chen, T. Şerbănuţă, and G. Roşu. jPredictor: a predictive runtime analysis tool for Java. In *Proc. 30th Int. Conf. on Software Engineering (ICSE 2008)*, pages 221–230, New York, NY, USA, 2008. ACM.
[11] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
[12] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd Int. Conf. on Software Engineering (ICSE 2000)*, pages 439–448, Limerick, Ireland, 2000. ACM Press.
[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, USA, 1995.
[14] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL 1997)*, pages 174–186, Paris, France, 1997. ACM Press.
[15] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.
[16] T. Henzinger, R. Jhala, and R. Majumdar. The BLAST software verification system. In *Proc. 12th Int. SPIN Workshop (SPIN 2005)*, volume 3639 of *LNCS*, pages 25–26. Springer, 2005.
[17] G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.
[18] M. Mansouri-Samani, P. Mehlitz, C. Pasareanu, J. Penix, G. Brat, L. Markosian, O. O'Malley, T. Pressburger, and W. Visser. Program model checking – a practitioner's guide. Technical report, 2008.
[19] NASA. Java PathFinder. http://babelfish.arc.nasa.gov/trac/jpf, 2011.
[20] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.