TRANT — A Scalable Trace Analysis Toolset for Java

Kevin Streit · Clemens Hammacher · Andreas Zeller · Sebastian Hack
Saarland University, Saarbrücken, Germany
{streit, hammacher, zeller, hack}@st.cs.uni-saarland.de

ABSTRACT

Understanding and analyzing what is going on in program runs is a common issue for researchers and tool builders. We present TRANT—a toolset for trace analysis of Java programs. TRANT allows for dynamic tracing of data dependencies; unlike other approaches, it allows for running Java programs in an unchanged form and has been designed from the ground up for scalability. Typical applications of TRANT include debugging support and finding candidates for parallelization, which are both illustrated in this paper.

1. INTRODUCTION

Designing and implementing dynamic analyses of program runs is an issue that many scientists have to deal with. Manipulation of the program has to be done in most cases in order to collect the data needed for the analysis. The problem is that correctly manipulating a program is error-prone and requires deep knowledge of the used language—knowledge that typically has nothing to do with the algorithms to implement. In particular, one has to guarantee that the manipulations of the program to analyze do not change the observable behavior. For this purpose, the manipulations should be well formed and produce a valid output program.

In this work we introduce TRANT, a toolset providing an API upon which one can build dynamic program analyses of Java applications. By using TRANT, researchers and tool builders can concentrate on their algorithms and let the tedious work of correctly instrumenting Java bytecode be done by our tools. A typical usage of TRANT is to trace *dynamic dependencies* (Figure 1), addressing the common debugging question "Where does this value come from?".

2. THE TRANT TOOLSET

The main tool we use to do all analyses described in this paper is called TRANT, a toolset initiated by C. Hammacher for his implementation of a dynamic slicer [4]. We mainly make use of two components of the toolset:

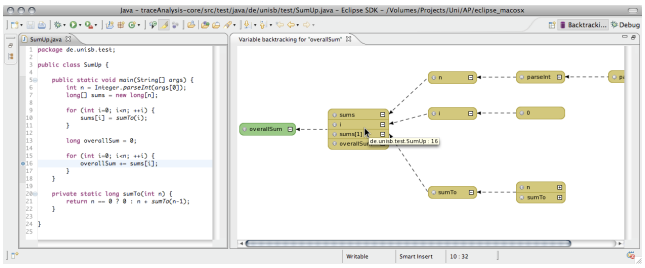


Figure 1: Tracing dependences between variables. This is a proof-of-concept implementation Eclipse plugin on top of TRANT, allowing the programmer to interactively trace back value origins.

Tracing executions. The first part of TRANT is its *tracing engine*, a *Java Agent* which can be appended to the Java program under concern. On load time, the engine instruments the Java bytecode to gather dynamic information during runtime of the program. The information is written to a *trace file* from which the full execution trace of the program can be reconstructed. TRANT can optionally compress the file using the very efficient Sequitur algorithm [6].

Tracking dependencies. The second part of TRANT consists of the implementation of several analyses. These are all based on the *data dependencies* in the program. The complete *dynamic dependence graph (DDG)* is way too large to be held in memory, since it consists of all data dependencies between individual instances of Java bytecode instructions. A lot of effort has been taken to minimize the memory consumption of the analyses by only keeping absolutely necessary information in memory while traversing the DDG. Other analysis techniques or visualizations can hook into this traversal process by following the well known visitor pattern [3]. We will describe this in slightly more detail in the next section.

TRANT and its documentation are publicly available [1].

3. USING THE TRANT APIS

Using the public TRANT APIs to load the trace file, one can read some general information about the program run. Some information provided for example is a representation of all instrumented classes (e.g. all classes which have been loaded during the program run), or a list of all threads which

have been started. One can then request the raw execution trace for a specific thread. An *iterator* over the instruction instances executed by this thread will be returned.

Beside this raw trace representation, there are several analysis interfaces. One example is the *dependence extractor,* which works on top of the trace iterator to compute all data and control dependencies between the bytecode instruction instances contained in the trace. The developer is informed about the identified dependencies through several *visitors*, and about additional events like object creations or method entries and returns.

For complex analyses, it is often necessary to store some user-defined data with the visited elements. For this purpose, TRANT provides a *factory* which will be used to create all objects needed during the traversal of the DDG. Arbitrarily complex information can then be attached to each of them. This design also facilitates garbage collection since objects that are never visited are automatically discarded.

4. APPLICATIONS

Some analyses implemented on top of this framework are shown below. They all use the dependency information as described in the previous section to compute more complex data, or to visualize parts of the information.

4.1 Debugging Support

Dynamic slicing is widely used for filtering the input to certain analyses or to further focus the output generated by these analyses [2]. Unfortunately a developer can not intuitively use the slicing technique by hand. The idea of knowing where the specific value of a variable at a specific point in time comes from can be crucial for debugging. While experimenting with using the dynamic dependencies reported by TRANT, we came up with a variable backtracker integrated into the Eclipse IDE, supporting the Java debugging process.

During debugging, developers often monitor the values of variables. Starting from the variable view provided by the Java debugger plugin the developer can open our backtracker for any variable. A bisected perspective opens with a dynamic data flow graph on the one side and a Java editor on the other side. Each node in the graph represents an instruction in the source code and contains all involved variables that influenced the actual value of the target variable. By clicking on a node, the corresponding source file is opened in the editor part and the represented instruction is highlighted.

Using our backtracker one can intuitively follow the path of data flow that leads to the value in doubt.

4.2 Finding Parallelization Candidates

One of the biggest challenges imposed by multi-core architectures is to exploit their potential for *legacy systems* not built with multiple cores in mind. By analyzing *dynamic data dependencies* of a program run, one can identify independent computation paths that could have been handled by individual cores.

In this analysis, the *critical path* through the data dependencies is used as a measure for the shortest possible execution time of a given part of the trace. By dividing the number of dynamic instruction instances, that is the number of individual instruction executions, by the length of the critical path, one can compute the so called *parallelization potential* of that part.

As one application of TRANT, we investigate the potential parallelism of loops in Java applications. To this end, we first compute a loop tree [7] for every method in the program, based on its control-flow graph (the *CFG* is given by our toolset). Second, we perform a backward traversal of the trace using the TRANT APIs. Each time the execution reaches a loop boundary, a new computation of the critical path through the data dependencies is started for this instance of the loop. Consecutive iterations of the loop body of one dynamic instance of the loop are handled as one execution for which the critical path is determined.

From the parallelization potential and the overall influence of the loop instance, we finally calculate the *gain* of speed which could at the best be realized when fully parallelizing this loop. This is used to rank the loops according to their ability to be parallelized. The ranking is then presented to the developer in a result list. Additionally the top ten loops in the source code are marked for the developer to be able to easily identify the parallelization hot spots in the code.

More details on how the candidate proposal works can be found in our earlier work [5].

5. CONCLUSION

TRANT is a high level scalable toolset for keeping the troubles of low level program manipulation for observation purposes away from scientists. This allows them to concentrate on their analyses to implement.

Often TRANT even renders unnecessary to actually implement a program transformation. Instead it can just be simulated on an arbitrary number of program traces to check whether it meets the expectations. This may save several days or weeks which would have been spent on really implementing a technique just to discover that it does not work out as desired in practice.

Based on TRANT, we already implemented several analysis and simulation techniques, and will be implementing more. We are optimistic it will be just as useful for other researchers and tool builders—and are happy to make it publicly available [1].

6. REFERENCES

[1] http://www.st.cs.uni-saarland.de/javaslicer.

[2] M. Ducassé. A pragmatic survey of automated debugging. In *Proc. 1st Workshop on Automated and Algorithmic Debugging*, volume 749 of *LNCS*, 1993.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Abstraction and Reuse in Object-Oriented Designs. In O. M. Nierstrasz, editor, *ECOOP'93: Object-Oriented Programming - Proc. of the 7th European Conference*, pages 406–431, Berlin, Heidelberg, 1993. Springer.

[4] C. Hammacher. Design and implementation of an efficient dynamic slicer for Java. Bachelor's Thesis, November 2008.

[5] C. Hammacher, K. Streit, S. Hack, and A. Zeller. Profiling java programs for parallelism. In *Proc. 2nd International Workshop on Multi-Core Software Engineering (IWMSE)*, May 2009.

[6] C. G. Nevill-Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. In *Data Compression Conference, Snowbird, Utah, IEEE Computer Society*, pages 3–11, 1997.

[7] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.*, 24(5):455–490, 2002.

APPENDIX

A. DEMO SETUP

Assuming a 30-minute slot for our demonstration, we aim for the following structure:

- Introduction to dynamic dependences (slides; 2 minutes)
- A brief look at the TRANT framework (slides; 2 minutes)
- A simple instruction tracer (live programming; 5 minutes)
- Tracking value origins (live application demo; 3 minutes)
- Finding parallelization candidates (live application demo; 5 minutes)
- Scalability (live application demo; 2 minutes)
- Conclusion (slides; 1 minute)

In the above scheme, we have allocated 10 minutes for interaction with the audience and can take questions any time to guarantee for an exciting event. Also, we will conduct the presentation as a pair, thus allowing for lively on-stage interaction in "pair programming" style.

B. DEMO EXECUTION

Because TRANT is a toolset providing public APIs to build analyses on it is not easy to demonstrate TRANT itself. Therefore, after shortly introducing TRANT basics, we will demonstrate its usage on three examples—one developed live on stage, and two proof-of-concept tools we implemented so far. With this, we demonstrate how totally different types of analysis can be realized on top of TRANT.

B.1 A Simple Instruction Tracer

We will shortly introduce the main APIs TRANT provides and its main classes to use in order to get a simple analysis up and running. In our example, we will implement a analysis method counting the number of dynamically executed bytecode instructions in the traced program run.

The usage of the framework follows the well known *visitor* design pattern and it should be easy to follow the demonstration. In our demonstration, we will show that we can implement such an analysis in less than five minutes.

We will use this very lightweight and straight forward analysis implementation to later on demonstrate the memory and runtime scalability of TRANT itself.

B.2 Tracking Variable Origins

Figure 2 shows our variable backtracker in action, as described in Section 4.1.

The backtracker is a proof-of-concept implementation, but it already perfectly works for small to medium scale programs. We plan to mainly base our demonstrations on this tool.

We will:

- Introduce the short *SumUp* example program that can be seen in the screenshot.
- Start a usual debugging session in the *Eclipse IDE* with a breakpoint set at line 16 in the *SumUp* class.
- When the breakpoint is reached we will start variable backtracking via the context menu of the *overallSum* variable in the debugger.
- The *Backtracking perspective* will open and we will explain what the data flow graph that can be seen in the screenshot shows.
- If time permits, we will shortly show that the backtracking equally works with medium to large scale real world programs.

B.3 Finding Parallelization Candidates

Figure 3 shows our parallelization candidate proposal tool described in section 4.2. This tool was developed as an early artifact of our ongoing research in the area of automated parallelization. We presented this work at the International Workshop on Multi-core Software Engineering (IWMSE) hosted at ICSE 2009 in Vancouver, Canada [5].

Although the tool is in its infancy, it is already useful as a first hint on where to start looking when parallelizing legacy code. It also demonstrates a totally different kind of analysis than the profiler or backtracker based on TRANT.

In our demonstration of the tool we will:

- Give a short explanation of our analysis framework in the Eclipse IDE and explain the *Trace Library*, a library for caching analysis results and traces of program runs.
- Explain how the trace of a program run is created and stored in the *Trace Library*.
- Show how to start the analysis process right from the *Trace Library* for the same *SumUp* example program seen before.
- After a very short moment the analysis will finish and the top ten parallelization candidates are presented in a clear view.
- From this we will show how one can use the result view to directly navigate to the corresponding source locations.
- If time permits, we will shortly show what other analyses are already implemented in the toolset and how these analyses scale to larger programs.

B.4 Scaling Up

In order to show how TRANT scales, we will use the sample profiler introduced in the demonstration of the APIs. We use this particular analysis here since it introduces almost no overhead to the runtime or memory consumption of TRANT itself and thus makes it possible to measure the scalability of TRANT alone.

For the demonstrations, we will use several programs of the DaCapo benchmark suite that range from 180,000 executed bytecode instructions to several billion. The benchmark suite only contains real world Java applications like for example parts of the Eclipse IDE itself.

These applications are a plausible candidate to show that the bytecode instrumentations performed by TRANT are sound and do not manipulate the observable program behavior. DaCapo performs several tests to ensure that the programs did exactly what they should do without manipulations.

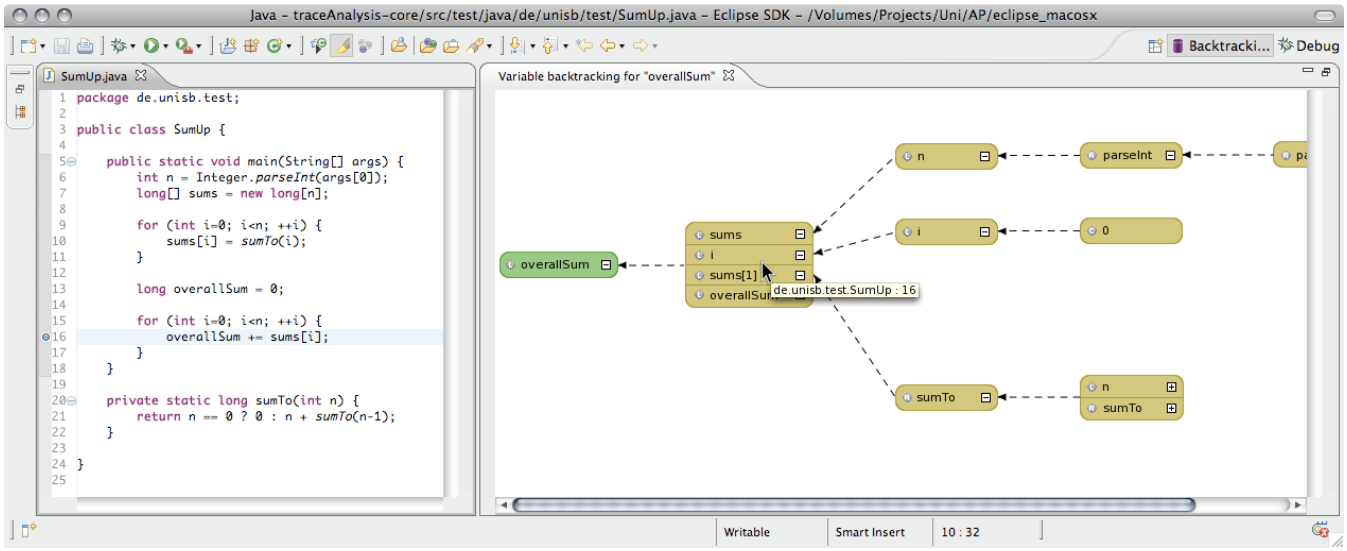
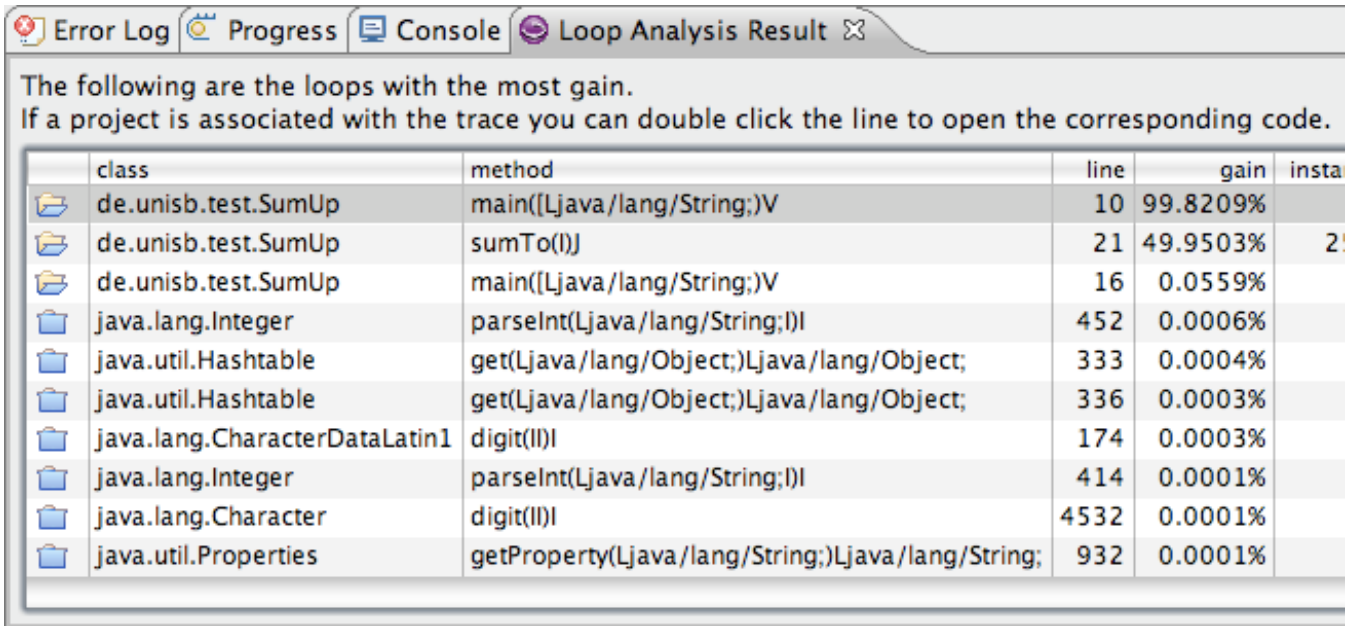


Figure 2: The variable backtracking perspective — An active variable backtracking is shown for a simple example program and the value of the *overallSum* variable in line 16. On the right hand site the partially expanded data flow graph is shown. Each node represents a particular source code instruction and contains all variables whose value took part in forming the actual value of a variable or a method return value.



class	method	line	gain	insta
de.unisb.test.SumUp	main([Ljava/lang/String;)V	10	99.8209%	
de.unisb.test.SumUp	sumTo(I)J	21	49.9503%	2
de.unisb.test.SumUp	main([Ljava/lang/String;)V	16	0.0559%	
java.lang.Integer	parseInt(Ljava/lang/String;I)I	452	0.0006%	
java.util.Hashtable	get(Ljava/lang/Object;)Ljava/lang/Object;	333	0.0004%	
java.util.Hashtable	get(Ljava/lang/Object;)Ljava/lang/Object;	336	0.0003%	
java.lang.CharacterDataLatin1	digit(II)I	174	0.0003%	
java.lang.Integer	parseInt(Ljava/lang/String;I)I	414	0.0001%	
java.lang.Character	digit(II)I	4532	0.0001%	
java.util.Properties	getProperty(Ljava/lang/String;)Ljava/lang/String;	932	0.0001%	

Figure 3: The parallelization candidate proposals as reported by our parallelization toolset. Only the top ten reported loops for a simple example program can be seen in this screenshot. Each of the entries in the result list is selectable. On selection the corresponding source file, if available, will open in the Java editor and the parallelization candidate will be highlighted.

C. POSTER

In addition to the live demos, we will also have a poster showing the following:

- Motivation and benefits.
- The main classes and APIs provided by the TRANT toolset.
- A rough sketch of the system architecture.
- The complete work flow starting from a program to analyze, tracing a run of this program, using TRANT to analyze the trace and producing and visualizing the result of the analysis.
- The URL where TRANT can be downloaded [1].