A Formal Design of a Tool for Static Analysis of Upper Bounds on Object Calls in Java[⋆]

Konrad Siek and Paweł T. Wojciechowski

Poznań University of Technology
{Konrad.Siek, Pawel.T.Wojciechowski}@cs.put.poznan.pl

Abstract. This paper presents a formal design of a tool for statically establishing the upper bound on the number of executions of objects' methods in a fragment of object-oriented code. The algorithm that our tool employs is a multi-pass interprocedural analysis consisting of data flow and region-based analyses. We describe the formalization of each of stage of the algorithm. This rigorous specification greatly aids the implementation of the tool by removing ambiguities of textual descriptions. There are many applications for information obtained through this method including reasoning about concurrent code, scheduling, code optimization, compositing services, etc.We concentrate on using upper bounds to instrument transactional code that uses a synchronization mechanism based on versioning, and therefore benefits from *a priori* knowledge about the usage of shared objects within each transaction. To this end we implement a precompiler for Java that analyzes transactions, and injects generated source code to initialize each transaction.

Keywords: static analysis, data flow analysis, transactional memory

1 Introduction

In this paper we present a tool for estimating the maximum of how many times methods of objects will be called within a fragment of object-oriented code. We report the tool's formalization and discuss its implementation. We see the formalization as one of the main contributions of this paper. The described algorithm is relatively simple: it is based on data flow analysis to establish information about values and paths and region analysis to tally method calls. We expand regions with additional properties so that the final, vital part of the analysis becomes straightforward. We also describe a use of a natural positive set extended by an absorbing value to count uncertain executions. In effect we manage to infer upper bounds (either concrete or infinite) that provide safety.

There are several possible applications for information about the upper bound on the number of objects' method calls obtained through our analysis. Among

[⋆] This work has been partially supported by the Polish Ministry of Science and Higher Education within the European Regional Development Fund, Grant No. POIG.01.03.01-00-008/08.

others, such upper bounds may be used to analyze concurrent code to find relationships between threads: a thread accessing a shared object once, several times, or not at all may impact safety guarantees like isolation or performance in different ways. With this information prior to execution it can be treated differently, i.e., applied proper synchronization, delayed, or executed as-is without breaking guarantees. The upper bounds can also be applied in compile-time resource optimization. For instance, the amount of memory used by a given program or its influence on network traffic may be estimated from calls to particular objects if the interface is known and used to configure the environment appropriately or to optimize the analyzed program. Other uses may be found in code rewriting, automatic refactoring, etc. Apart from the work of [15] these applications seem largely unexplored.

Our particular interest lays with algorithms that require this type of information up front for efficient operation, e.g., those found in scheduling and synchronization via transactions. One such application is Atomic RMI [26, 27]. It is a distributed transactional memory extension to Java RMI, an API for distributed programming using remote procedure calls that is well-established in business and industry. Atomic RMI uses versioning algorithms that need *a priori* information about shared object accesses to figure out whether an object may be released before a transaction commits (or aborts). Releasing objects early gives Atomic RMI a performance edge, so a precompiler that provides upper bounds automatically and precisely has significant practical value. We see the paper as contributing the application and the implementation of the precompiler as well as the analysis and its formalization. The technical documentation of the tool is available on the project web page [19].

The paper has the following structure. In Section 2 we present work similar to ours. We formalize the static analysis in Section 3— each subsection describs an individual constituent part of the analysis. Then, in Section 4 we discuss the precompiler implementation. Finally, we conclude with Section 5.

2 Related Work

There is a large body of research related to analysis of programs that aims at deriving information about execution patterns statically (we sketch out some of these below). However, we do not know examples of using this information for optimizing the execution of distributed transactions in the way we do. The largest body of work to which our static analysis bears resemblance has been done with regard to the Worst Case Execution Time (WCET) problem [24]—establishing upper bounds on the time code takes to run. However, most of this work is aimed at real-time systems, not transactional concurrency control which is the main concern of our work. A number of frameworks are available for WCET analysis, like aiT [4], Bound-T [10], SWEET [7], and SymTA/P [20]. A comprehensive survey of these tools and methods was done in [25]. Whereas our approach is based on region analysis, some work in WCET use symbolic analysis [13], path analysis [8], and abstract interpretation [10, 5]. We also use the latter type of analysis for our value analysis algorithm (Section 3.2). In WCET emphasis is placed on the problem of evaluating loops in general and bounding loop iterations in particular. This is done, among others, by the use of Presburger arithmetic [17], path analysis (using integer linear programming) [21], or a combination of methods involving abstract interpretation [3]. Our work touches on those concerns in Section 3.2 where we use loop unfolding to establish their bounds roughly similar to that of SWEET [7] but simpler. WCET tools additionally often use the Implicit Path Enumeration technique [12] or single feasible paths [28] to establish worst-case paths and perform final timing analyses. While our application presents no need for the latter, we use region-based analysis (described in Section 3.4) to conservatively deduce worst-case paths. WCET tools also allow for manual declaration or correction of difficult-to-deduce information (e.g., loop bounds).

Our work has significant similarities to work on lock inference. Lock inference aims to determine which memory locations or shared objects in a program must be protected by locks and where these locks should be located. Thus, our work and lock inference share the same ultimate goal of providing concurrency control via static analysis albeit by different mechanisms. The authors of [2] employ backward data flow analysis to transform a program's control flow graph into a path graph which is then used to derive locks. In [9] the authors present a method for identifying shared memory locations using type-based analysis, points-to analysis, and label flow analysis [16]. In Autolocker [14] pessimistic atomic sections are converted into lock-guarded critical sections by analyzing dependencies among annotated locks based on a computation history derived from a transformation of the code using a type system.

In [15] the authors propose a tool for the automatic inference of upper bounds on the usage of user-specified resources. Rather than memory or execution time, these may be the number of open files, accesses to database, sent text messages, etc.This work and ours share the set of tools they use (Soot and Jimple [22]) and they both try to solve a similar problem. The tool presented by the authors performs a data flow analysis to derive data dependencies, then creates a set of equations from input-output parameter size relationships. Finally the equations are solved using a recurrence solver. Our approach differs most in that we perform region analysis to determine maximum paths and resource use where they construct and solve equations.

3 Upper Bound Prediction Analysis

In this section we describe an algorithm for deriving upper bounds (or suprema) on the number of method calls to objects via static analysis. The upper bounds for some specific objects—remote objects used in a transaction—are used for concurrency control by Atomic RMI. To derive the suprema the algorithm performs multiple passes over the input code in the form of an intermediate language (see Section 3.1). Three passes correspond to the three phases that form our algorithm: value analysis, region analysis, and call count analysis. In addition another pass is performed before value analysis to identify loops. Value analysis

Identifiers	$j \in \mathit{Ident}$	
Constants	$c \in \mathit{Const}$	
Labels	$l \in \mathit{Lab}$	
Types	$t \in \mathit{Type}$	
Fields	$f \in \mathit{Field}$	$::= j : t$
Immediates	$i \in \mathit{Imed}$	$::= j \mid c$
Right-hand values	$r \in \mathit{Rval}$	$::= i \mid i[i] \mid i.[f] \mid [f]$
Methods	$m \in \mathit{Meth}$	$::= \texttt{invoke}\ i.[j(j_1, ..., j_n)](i_1, ..., i_n)\{b_1, ..., b_n, \}$
Conditions	$p \in \mathit{Cond}$	$::= i == i \mid i \geq i \mid i > i \mid i \leq i \mid i < i \mid i \neq i$
Expressions	$e \in \mathit{Expr}$	$::= i + i \mid i \ / \ i \mid i \ * \ i \mid i\%i \mid -i \mid i - i \mid i \mid i \mid i \ \& \ i$
		$\mid i \ \texttt{xor} \ i \mid i \gg i \mid i \ll i \mid (t)i \mid i \ \texttt{instanceof} \ t$
		$\mid \texttt{new} \ t \mid \texttt{new} \ t[i_1]...[i_n] \mid \texttt{length} \ i \mid p$
Statements	$s \in \mathit{Stmt}$	$::= \texttt{switch}(i)\{\texttt{case} \ c_1 : l_1; ...; \texttt{case} \ c_n : l_n; \texttt{default:} \ l_0\}$
		$\mid \texttt{if} \ p \ \texttt{goto} \ l_1 \ \texttt{else} \ l_2 \mid l \mid j = m \mid j = r \mid m$
		$\mid \texttt{goto} \ l \mid \texttt{return} \ i$
Blocks	$b \in \mathit{Bloc}$	$::= l : b_1; ...; b_n; \mid b_1; ...; b_n; \mid s$

Fig. 1: Jimple syntax (altered).

predicts possible values of variables in the code. It also identifies unfeasible or dead code, and unfolds loops. Region analysis uses the results of value analysis to convert the input code into regions. Finally, call count analysis examines these regions to produce the upper bounds on method call counts. We describe our use of Jimple and the phases of the algorithm in detail in the following subsections.

3.1 Translation to Jimple

In order to analyze a program in Java with Atomic RMI transactions we translate it into an intermediate representation called Jimple [23] using the Soot framework [22]. We use Jimple as an intermediate language because it is much better suited for analysis than either Java source code or bytecode. The reason for this is that Jimple is a 3-address code representation with a very limited instruction set consisting of 17 statements. In our earlier attempts to perform similar analyses using Java source code [18] we learned that such analyses become convoluted and the implementation costly in effort due to the number of constructs needing handling and the complexity of their semantics.

The part of Jimple syntax that is pertinent to our further discussion is presented in Fig. 1. The semantics are mostly straightforward, the reader is referred to [23] for details and the complete language. The constructs most important to us are the conditional statements, switch statements, method invocations, assignments, and labeled blocks. We introduce superficial alterations to the syntax to suit further description of the algorithm. We treat labels as statements and

```
1    Transaction k = new Transaction(reg);
2    a = k.accesses(a, 2); // generated, upper bound = 2
3    b = k.accesses(b, 1); // generated, upper bound = 1
4    k.start();
5    int balance = a.getBalance();
6    if (balance >= sum) {
7        a.withdraw(sum); b.deposit(sum);
8        k.commit();
9    } else
10       k.rollback();
```

Fig. 2: Example Java code for a distributed transaction using Atomic RMI.

```
  k = new soa.atomicrmi.Transaction;
  invoke k.[<init>(@parameter0)](reg){$b0};
  invoke k.[start()](){$b1};
  balance = invoke a.[getBalance()](){$b4};
  if balance < sum goto label1 else label0;
label0:
  invoke a.[withdraw(@parameter0)](sum){$b5};
  invoke b.[deposit(@parameter0)](sum){$b6};
  invoke k.[commit()](){$b2}; return null;
label1:
  invoke k.[rollback()](){$b3};
```

Fig. 3: Java code translated to altered Jimple.

place them at the beginning of labeled blocks. We modify the conditional statement to define target labels for both outcomes instead of having a succeeding block of code called if the condition is false. We do not distinguish among different sorts of method invocations—interface, special, virtual, and static—and we remove type information from invocations while adding a direct definition of the methods' arguments and a set of possible bodies. We also fix method invocations nested in other statements by defining a separate assignment statement instead where the results of the invocation are assigned to an identifier. We show an example Java program using our Atomic RMI distributed transactions Fig. 2 translated to the altered form of Jimple in Fig. 3 (lines 2, 3 are omitted because they are generated from Jimple later—see Section 4.2 for details).

For the purposes of analysis the input program is represented as *Control Flow Graphs* (CFGs) and each method's body is a separate graph. Most statements in Jimple will have one incoming and outgoing edge. The conditional statement will have 2 outgoing edges, and the `switch` statement will have one more outgoing edge than it has conditions. Loop headers and labeled blocks will have more incoming edges. Invoke statements point to CFGs of other method bodies.

3.2 Value Analysis

As a preliminary to the value analysis we find loops in code. A loop consists of a head and a body. A loop head is a statement s that dominates any other statement s' (all paths from the start to s' lead through s [1], denoted s **dom** s')

$$\begin{array}{l}
\mathbb{G}(s) \triangleq \mathbb{S} \triangleq (\mathbb{S}_V, \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I) \\
\mathbb{G}(s) = \mathsf{eval}(\mathsf{join}(\{\mathbb{G}(p) \mid s \ \mathbf{succ}\ p\}), s) \\
\mathsf{eval}(\mathbb{S}, j = r) \triangleq (\mathbb{S}_V[j \mapsto \{\mathsf{val}(r, \mathbb{S}_V)\}], \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I) \\
\mathsf{eval}(\mathbb{S}, j = m) \triangleq \mathbb{S}' = \mathsf{eval}(\mathbb{S}, m), (\mathbb{S}_V \oplus \mathbb{S}'_V[j \mapsto \{\mathsf{val}(m, \mathbb{S}_V)\}], \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I) \\
\mathsf{eval}(\mathbb{S}, \texttt{invoke}\ \ i.[j(j_1, ..., j_n)](i_1, ..., i_n)\{b_1, ..., b_m\}) \triangleq \\
\quad \text{case } \mathsf{depth}(i.j) \rightarrow \mathbb{S}_V[k \mapsto \omega k \in \mathsf{defs}(b_1) \cup ... \cup \mathsf{defs}(b_m)], \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I) \\
\quad \text{otherwise} \rightarrow \mathbb{S}' = (\mathbb{S}_V[j_1 \mapsto \mathsf{val}(i_1, \mathbb{S}_V), ..., j_n \mapsto \mathsf{val}(i_n, \mathbb{S}_V)], \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I), \\
\qquad \mathsf{join}(\mathsf{eval}(\mathbb{S}', b_1), ..., \mathsf{eval}(\mathbb{S}', b_m)) \\
\mathsf{eval}(\mathbb{S}, l) \triangleq (\mathbb{S}_V \oplus \mathbb{S}_P(l), \mathbb{S}_P, \mathbb{S}_D, \mathbb{S}_I) \\
\mathsf{eval}(\mathbb{S}, s : \texttt{return}\ i) \triangleq (\mathbb{S}_V, \mathbb{S}_P, \mathbb{S}_D \cup \{(s, s') \mid s\ \mathbf{pdom}\ s', s' \in \mathit{Stmts})\}, \mathbb{S}_I) \\
\mathsf{eval}(\mathbb{S}, s : \texttt{if}\ p\ \ \texttt{goto}\ \ l_1\ \ \texttt{else}\ \ l_2) \triangleq \\
\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \texttt{true} \rightarrow (\mathbb{S}_V, \mathbb{S}_P\big[l_1 \mapsto \mathbb{S}_P[p \mapsto \texttt{true}]\big], \mathbb{S}_D \cup \{(s, l_2)\}, \mathbb{S}_I) \\
\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \texttt{false} \rightarrow (\mathbb{S}_V, \mathbb{S}_P\big[l_2 \mapsto \mathbb{S}_P[p \mapsto \texttt{false}]\big], \mathbb{S}_D \cup \{(s, l_1)\}, \mathbb{S}_I) \\
\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \omega \rightarrow (\mathbb{S}_V, \mathbb{S}_P\big[l_1 \mapsto \mathbb{S}_P[p \mapsto \texttt{true}], l_2 \mapsto \mathbb{S}_P[p \mapsto \texttt{false}]\big], \mathbb{S}_D, \mathbb{S}_I) \\
\mathsf{eval}(\mathbb{S}, s : \texttt{switch}(i)\{\texttt{case}\ c_1 : l_1; ...; \texttt{case}\ c_n : l_n; \texttt{default:}\ \ l_0\}) \triangleq (\mathbb{S}_V, \\
\quad \mathbb{S}_P\big[l_1 \mapsto \mathbb{S}_P(l_1)[j = \mathsf{val}(c_1)], ..., l_n \mapsto \mathbb{S}_P(l_n)[j = \mathsf{val}(c_n)]\big], \mathbb{S}_D \\
\quad \cup\{(s, l_k) \mid \mathsf{pred}(c_k = j, \mathbb{S}) = \texttt{false} \vee \mathsf{pred}(c_r = j, \mathbb{S}) = \texttt{true}, k = 1, ..., n, r = 1, ..., k, \} \\
\quad \cup\{l_0 \mid \mathsf{pred}(\exists k, c_k = j, \mathbb{S}) = \texttt{true}, k = 1, ..., n\}, \mathbb{S}_I) \\
\mathsf{eval}(\mathbb{S}, s \in \mathbb{H}) \triangleq \mathsf{evalloop}(s, \mathbb{G}, \mathbb{L}(\mathsf{id}(s)), 1, L) \\
\mathsf{eval}(\mathbb{S}, s \in \mathbb{B} \wedge s \notin \mathbb{H}) \triangleq \mathbb{S} \\
\mathsf{join}(\mathbb{S}^1, ..., \mathbb{S}^n) \triangleq \big(\{k \mapsto \mathbb{S}^1_V(k) \cup ... \cup \mathbb{S}^n_V(k) \mid k \in (\mathrm{dom}\ \mathbb{S}^1_V \cup \mathrm{dom}\ \mathbb{S}^n_V)\}, \{l \mapsto \\
\quad \{k \mapsto \mathbb{S}^1_P(l)(k) \cup ... \cup \mathbb{S}^n_P(l)(k)\} \mid l \in \mathrm{dom}\ \mathbb{S}^1_P \cup ... \cup \mathbb{S}^n_P, k \in \mathrm{dom}\ \mathbb{S}^1_P(l) \cup ... \cup \mathbb{S}^n_P(l)\}, \\
\quad \mathbb{S}^1_D \cup ... \cup \mathbb{S}^n_D, \{k \mapsto \max\limits_{i=1,...,n}(\mathbb{S}^i_I(k)) \mid k \in (\mathrm{dom}\ \mathbb{S}^1_I \cup \mathrm{dom}\ \mathbb{S}^n_I)\}\big) \\
\mathbb{S}'_V \oplus \mathbb{S}''_V \triangleq \{k \mapsto \mathbb{S}'_V(k) \cup \mathbb{S}''_V(k) \mid k \in (\mathrm{dom}\ \mathbb{S}'_V(k) \cup \mathrm{dom}\ \mathbb{S}''_V(k))\}
\end{array}$$

Fig. 4: Value analysis.

while simultaneously being the successor of s' (there is a path from s' to s). A loop body is a sequence of statements all of which are dominated by a loop head and have that loop head as their successor. We gather the heads in set $\mathbb{H}$ and create map $\mathbb{L}$ which contains a unique identifier of each statement h from $\mathbb{H}$ as a key mapped to a set of statements whose elements are all dominated by h.

The first phase of the analysis is a forward data flow analysis performed on the CFG. Its main purpose is threefold: to establish the possible values of variables at each node of the CFG representing the program, to count the maximum number of loop iterations through loop unfolding, and to establish which nodes of the CFG are dead or unfeasible (will not be executed). There are two principal functions in value analysis, eval and join. These functions are used to compute members of global state $\mathbb{G}$, a data structure that results from the analysis. We present all of those elements in Fig. 4 and describe them below.

Global state $\mathbb{G}$ maps Jimple statements to states which apply to them. Global state is constructed during value analysis by constructing a state for each statement using a transfer function eval and an aggregation of states for the predecessors of a given statement using join. We designate individual states $\mathbb{S}$, such that $\mathbb{S}$ is a quadruple consisting of a value map $\mathbb{S}_V$, an inferred value map $\mathbb{S}_P$, a dead edge set $\mathbb{S}_D$, and a loop iteration map $\mathbb{S}_I$. $\mathbb{S}_V$ is a map of locals (identifiers and constants) to sets of values—it indicates what values a given variable or constant may take at this point in the program. $\mathbb{S}_P$ maps labels (names of blocks) to value maps and indicates assumptions about values of variables and constants inferred from conditions that will apply at a particular succeeding statement. $\mathbb{S}_D$ contains pairs of statements indicating edges that will definitely not be used in the execution of the program. $\mathbb{S}_I$ is a map of loop heads to numbers indicating the maximum estimated iteration count of the loop, or an unknown value. All components of the state are initially empty.

Transfer function eval is the key function of the analysis. It analyzes each Jimple statement and establishes the state of the program that holds after the statement is evaluated. The resulting state depends on the type of statement and the state before that statement.

When encountering an assignment of a right-hand side expression r to an identifier j, a new mapping is added to $\mathbb{S}_V$ that maps j to the set of possible values of expression r. When eval encounters an assignment of the results of method invocation m to identifier j, first m is evaluated separately and state after its evaluation $\mathbb{S}'$ is extended by the mapping of j to the result of m. A method invocation itself is analyzed by first extending the value map by parameter identifiers mapped to the values of arguments. Then all possible bodies are evaluated and the results are joined (the particular bodies are identified from the type hierarchy and arguments but we leave the details to Soot). But if recursion exceeds a depth L all the values defined within possible method bodies are set to unknown (this degrades precision but maintains safety). L must be tuned to a given application. A label l extends the value map with predictions from the inferred map. A `return` statement adds all other statements it dominates to $\mathbb{S}_D$.

When analyzing an `if` statement the expression that is the condition is checked. If the condition yields true then the edge in the CFG from the current statement to label l_2 is added to dead edges, and predictions about variables are made under the assumption that the condition will be true at label l_1. Conversely, if the condition yields false the edge from the statement to l_1 will be dead and predictions will be made for l_2 under the assumption that the condition is false. If the condition yields an unknown, no edges will be added to the dead edge set, but predictions for both l_1 and l_2 will be made. A `switch` statement is analyzed by creating a prediction for each constant $c_1, ..., c_n$ that the local i is equal to it at an appropriate label $l_1, ..., l_n$. Furthermore, if any of the constants c_k is definitely equal to i, edges from this statement to labels subsequent to that constant $l_{k+1}, ..., l_n$ and the default label l_0 are added to the dead edge set $\mathbb{S}_D$.

Function join (Fig. 4) is responsible for joining states and is used when a statement has two or more incoming edges. Each component of the state is

```
$\mathsf{evalloop}(s, \mathbb{G}', \mathbb{U}, i, L) \triangleq$
    $\mathbb{G}'' = \mathbb{G}', \quad \mathbb{G}''(u) = \mathsf{eval}(\mathsf{join}(\{\mathbb{G}''(u) \mid u \textbf{ succ } p \land u \in \mathbb{U}\})),$
    $\mathbb{E} = \{e \mid s \textbf{ succ } e \land s \notin \mathbb{U} \land e \in \mathbb{U}\},$
    $\mathbb{E}' = \mathbb{E} \setminus \{d \mid \mathbb{G}''(d) = \mathbb{S}', \mathsf{unpredecessed}(\mathbb{S}'_D, d) \land d \in \mathbb{E}\},$
    $\mathbb{S}^e = \mathsf{join}(\{\mathbb{G}''(e) \mid e \in \mathbb{E}'\}),$
    $\mathbb{Z} = \{(b, h) \mid h \textbf{ dom } b \land h \textbf{ succ } b \land \nexists s \in \mathbb{U}, h \textbf{ succ } s \textbf{ succ } b\}$
    $\mathbb{S}^z = \mathsf{join}(\{\mathbb{G}''(b) \mid (b, h) \in \mathbb{Z}\}),$
    case $\mathbb{Z} \subseteq \mathbb{S}^z_D \lor (\forall (b, h) \in \mathbb{Z}, \ \mathsf{unpredecessed}(\mathbb{S}^z_D, b)) \to (\mathbb{S}^e_V, \mathbb{S}^e_P, \mathbb{S}^e_D, \mathbb{S}^e_I[h \mapsto i])$
    case $i > L \to (\mathbb{S}^e_V[k \mapsto \omega, k \in \mathsf{defs}(\mathbb{U})], \mathbb{S}^e_P, \mathbb{S}^e_D, \mathbb{S}^e_I[h \mapsto \omega])$
    case $i \leq L \to \mathsf{evalloop}(h, \mathbb{G}'', \mathbb{U}, i + 1, L)$
$\mathsf{unpredecessed}(\mathbb{S}_D, s) \triangleq \forall s \ \mathsf{succ} \ p, \ (p, s) \in \mathbb{S}_D \lor \mathsf{unpredecessed}(p)$
$\mathsf{defs}(\mathbb{U}) \triangleq \{j \mid s \in \{j = m, j = r\} \land s \in \mathbb{U}\}$
```

Fig. 5: Loop unfolding within value analysis.

joined with its counterpart in the second state. Sets $\mathbb{S}_D$ are added together. The keys and values are copied to a new map, and if a key is present in both maps, the values are added ($\mathbb{S}_V$, $\mathbb{S}_P$) or the higher one is selected ($\mathbb{S}_I$).

We use the following helper functions within eval. Function val substitutes values from a value map for identifiers and constants (where possible) in a given expression and evaluates it to establish a set of values that the expression may yield. The returned set may consist of a single value, any number of elements or contain the unknown value ω. We use the function pred in a similar manner, except that only conditional expressions are evaluated and a single ternary value is returned—`true`, `false`, or ω. We use depth to find out the depth of a method's recursion. Function id produces a unique identifier of a statement. Operators **succ**, **dom** and **pdom** denote the succession, domination and post-domination relation of two statements in the CFG.

When encountering a statement that was identified as a head of a loop, function evalloop is used where the statements that form the body of the loop are taken from $\mathbb{L}$ and evaluated. During evaluation a collection of states $\mathbb{G}'$ is created and used to find those exit statements $\mathbb{E}'$ and back edges $\mathbb{Z}$ that may be executed during this iteration. If no back edge could be used during this iteration we know the loop exits, so we aggregate the states after all exit statements and finish evaluating the loop. It can also be deduced at this point that the loop will be executed at most as many times as we performed iterations. Otherwise, if we have not reached an arbitrary limit of iterations we conduct another iteration using evalloop. If the limit was reached we do not proceed but assume that this loop will continue indefinitely and set all the values that are defined within its body to unknown ω. Upon evaluation exit statements from the loop body are derived from the dead edge set of the resulting state. If there is only one exit from the loop then the loop exits in the current iteration and both the state of

Unit regions	$U \in \mathit{Units}$	$::=$	`unit`
Statement regions	$S \in \mathit{Statements}$	$::=$	`statement` s
Invocation regions	$I \in \mathit{Invocations}$	$::=$	`invoke` $j, R_1, ..., R_m, s$
Block regions	$B \in \mathit{Blocks}$	$::=$	`block` $[R_1, ..., R_n]$
Condition regions	$C \in \mathit{Conditions}$	$::=$	`condition` p, R_1, R_2
Loop regions	$L \in \mathit{Loops}$	$::=$	`loop` h, R
Regions	$R \in \mathit{Regions}$	$::=$	$U \mid S \mid I \mid B \mid C \mid L$

Fig. 6: The region-based intermediate representation.

the variables and the number of iterations are added to $\mathbb{S}$. Otherwise another iteration is required and the evaluation is repeated. In order to manage infinite loops or those where the conditions of exiting are uncertain, an iteration limit L is given which, when reached, will cause the evaluation to cease and set all effects of the loop to unknown value ω. Setting values to ω preserves safety. We use two additional helper functions within $\mathsf{evalloop}$. We define predicate $\mathsf{unpredecessed}$ which checks whether a statement's predecessors are all dead or the edge from them to it are unused. We also define function defs which returns the names of variables defined in a given statement.

3.3 Regions

The second phase of our analysis is concerned with preparing the input structure required by the third phase which is conducted using region-like structures. Thus we introduce a function to convert the CFG into a region graph. *Regions* [1, 11] are areas of code with a single entry point, like code blocks. We extend each region with information about its rôle in the code. We distinguish unit regions, statement regions, invocation regions, block regions, condition regions, and loop regions. We show their definitions in Fig. 6.

Regions are converted from Jimple CFG by the analysis defined in Fig. 7. The analysis is performed on the root of the CFG using regf. The function then handles each node of the CFG by recursion and returns a tree of regions. It uses the loop header set $\mathbb{H}$ and a map of loop headers to their bodies $\mathbb{L}$ from the previous analysis, and a set of dead statements $\mathbb{D}$ whose all incoming edges or predecessors are dead (according to $\mathbb{S}_D$). For convenience, we also define function block which creates a block region from a sequence of statements by applying regf to each of them in succession and aggregating them into a single region.

3.4 Call Count Analysis

Call count analysis is performed on the region tree in order to establish the number of times each object's methods are called. It is depicted in Fig. 8. The analysis begins with the application of function ccount at the root of the region tree and proceeds depth-first through the subregions. In general, method calls

$$
\begin{array}{l}
\mathbb{D} \triangleq \{s \mid \mathbb{S} = \mathbb{G}(s), \mathsf{unpredecessed}(\mathbb{S}_D, s)\} \\
\mathsf{block}(\mathbb{H}, [s_1, ..., s_n]) \triangleq \texttt{block}\ [R_i \mid 1 < i < n, R_i = \mathsf{regf}(\mathbb{H}, s_i) \wedge (i = 1 \vee \neg s_i \in R_{i-1})] \\
\mathsf{regf}(\mathbb{H}, l : b_1; ...; b_n;) \triangleq \mathsf{block}(\mathbb{H}, [l, b_1, ..., b_n]) \\
\mathsf{regf}(\mathbb{H}, b_1; ...; b_n;) \triangleq \mathsf{block}(\mathbb{H}, [b_1, ..., b_n]) \\
\mathsf{regf}(\mathbb{H}, s \in \mathbb{H}) \triangleq \texttt{loop}\ s, \mathsf{block}(\mathbb{H} \setminus \{s\}, [s' | s' \in \mathbb{L}(s)]) \\
\mathsf{regf}(\mathbb{H}, s \in \bigcup_{\forall h \in \mathbb{H}} \mathbb{L}(h) \vee s \in \mathbb{D}) \triangleq \texttt{unit} \\
\mathsf{regf}(\mathbb{H}, s : \texttt{if}\ p\ \texttt{goto}\ l_1\ \texttt{else}\ l_2) \triangleq \\
\quad \text{case } \nexists e \in \mathit{Stmt}, e\ \mathbf{pdom}\ s \rightarrow \\
\quad\quad \texttt{condition}\ p, \mathsf{block}(\mathbb{H}, [s' \mid l_1\ \mathbf{dom}\ s']), \mathsf{block}(\mathbb{H}, [s' \mid l_2\ \mathbf{dom}\ s']) \\
\quad \text{case } \exists e \in \mathit{Stmt}, \nexists e' \in \mathit{Stmt}, e\ \mathbf{pdom}\ s \wedge e'\ \mathbf{pdom}\ s \wedge e\ \mathbf{psdom}\ e' \rightarrow \\
\quad\quad \texttt{condition}\ p, \mathsf{block}(\mathbb{H}, [s' \mid l_1\ \mathbf{dom}\ s' \wedge e\ \mathbf{pdom}\ s']), \\
\quad\quad\quad\quad \mathsf{block}(\mathbb{H}, [s' \mid l_2\ \mathbf{dom}\ s' \wedge e\ \mathbf{pdom}\ s']) \\
\mathsf{regf}(\mathbb{H}, s : \texttt{switch}(i)\{\texttt{case}\ c_1 : l_1; ...; \texttt{case}\ c_n : l_n; \texttt{default:}\ l_0\}) \triangleq \\
\quad \text{case } \nexists e \in \mathit{Stmt}, e\ \mathbf{pdom}\ s \rightarrow \\
\quad\quad \texttt{condition}\ (i = c_1), \mathsf{block}(\mathbb{H}, [s' \mid l_1\ \mathbf{dom}\ s']), \big(, ..., \\
\quad\quad\quad \big(\texttt{condition}\ (i = c_n), \mathsf{block}(\mathbb{H}, [s' \mid l_n\ \mathbf{dom}\ s']), \mathsf{block}(\mathbb{H}, [s' \mid l_0\ \mathbf{dom}\ s'])\big)\big) \\
\quad \text{case } \exists e \in \mathit{Stmt}, \nexists e' \in \mathit{Stmt}, e\ \mathbf{pdom}\ s \wedge e'\ \mathbf{pdom}\ s \wedge e\ \mathbf{psdom}\ e' \rightarrow \\
\quad\quad \texttt{condition}\ (i = c_1), \mathsf{block}(\mathbb{H}, [s' \mid l_1\ \mathbf{dom}\ s' \wedge e\ \mathbf{pdom}\ s']), \big(, ..., \\
\quad\quad\quad \big(\texttt{condition}\ (i = c_n), \mathsf{block}(\mathbb{H}, [s' \mid l_n\ \mathbf{dom}\ s' \wedge e\ \mathbf{pdom}\ s']), \\
\quad\quad\quad\quad \mathsf{block}(\mathbb{H}, [s' \mid l_0\ \mathbf{dom}\ s' \wedge e\ \mathbf{pdom}\ s'])\big)\big) \\
\mathsf{regf}(\mathbb{H}, s : \texttt{invoke}\ i.[j(j_1, ..., j_n) : t](i_1, ..., i_n)\{b_1, ..., b_m\}) \triangleq \\
\quad\quad \texttt{invoke}\ i, \mathsf{regf}(\mathbb{H}, b_1), ..., \mathsf{regf}(\mathbb{H}, b_m), s \\
\mathsf{regf}(\mathbb{H}, s) \triangleq \texttt{statement}\ s
\end{array}
$$

Fig. 7: Region finding analysis.

on objects in the tree's leafs are counted and the counts are aggregated upwards, either by adding the call counts (with addjoin) in cases of sequences or by taking the highest count (using maxjoin) in cases of alternative program paths.

Function ccount takes three arguments—the global state $\mathbb{G}$, the maximum number of executions of the parent region n, and the region of appropriate type. The function returns a map of object identifiers to the number of times that particular object's method were called. Thus, when the function comes across statement or unit regions it returns empty sets. When it reaches an invoke region it notes the object owning the method and creates a mapping of that object to the number of times the parent region is to be executed; this mapping is then aggregated using function addjoin to the results of the evaluation of the joined bodies of the invoked method using ccount. If a block region is encountered its subregions are evaluated first and the results of these evaluations are aggregated using addjoin. When ccount encounters a conditional region the condition is

$$
\begin{aligned}
&\mathsf{ccount}(\mathbb{G}, n, \texttt{unit}\ ) \triangleq \varnothing\\
&\mathsf{ccount}(\mathbb{G}, n, \texttt{statement}\ s) \triangleq \varnothing\\
&\mathsf{ccount}(\mathbb{G}, n, \texttt{invoke}\ j, R_1, ..., R_m, s) \triangleq \mathbb{S} = \mathbb{G}(s),\\
&\quad \mathsf{addjoin}(\{\mathbb{S}_V(j) \mapsto n\}, \mathsf{maxjoin}(\mathsf{ccount}(\mathbb{G}, n, R_1), ..., \mathsf{ccount}(\mathbb{G}, n, R_m)))\\
&\mathsf{ccount}(\mathbb{G}, n, \texttt{block}\ [R_1, ..., R_n]) \triangleq \mathsf{maxjoin}(\mathsf{ccount}(\mathbb{S}_V, n, R_1), ..., \mathsf{ccount}(\mathbb{G}, n, R_n))\\
&\mathsf{ccount}(\mathbb{G}, n, \texttt{condition}\ p, R_1, R_2, s) \triangleq \mathbb{S} = \mathbb{G}(s),\\
&\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \texttt{true} \to \mathsf{ccount}(\mathbb{G}, n, R_1)\\
&\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \texttt{false} \to \mathsf{ccount}(\mathbb{G}, n, R_2)\\
&\quad \text{case } \mathsf{pred}(p, \mathbb{S}) = \omega \to \mathsf{maxjoin}(\mathsf{ccount}(\mathbb{G}, n, R_1), \mathsf{ccount}(\mathbb{G}, n, R_2))\\
&\mathsf{ccount}(\mathbb{G}, n, \texttt{loop}\ h, R) \triangleq \mathbb{S} = \mathbb{G}(h), \mathsf{ccount}(\mathbb{G}, n * \mathbb{S}_I(h), R)\\
&\mathsf{maxjoin}(\mathbb{M}_1, ..., \mathbb{M}_n) \triangleq \{k \mapsto \max(\mathbb{M}_1(k), ..., \mathbb{M}_n(k)) \ \mid\ k \in \operatorname{dom} \mathbb{M}_1 \cup ... \cup \operatorname{dom} \mathbb{M}_n\}\\
&\mathsf{addjoin}(\mathbb{M}_1, ..., \mathbb{M}_n) \triangleq \{k \mapsto \mathbb{M}_1(k) + ... + \mathbb{M}_n(k) \ \mid\ k \in \operatorname{dom} \mathbb{M}_1 \cup ... \cup \operatorname{dom} \mathbb{M}_n\}\\
&\omega + c = \omega,\ \omega * c = \omega,\ \max(\omega, c) = \omega
\end{aligned}
$$

Fig. 8: Call count analysis.

checked and one of the subregions is evaluated, if the condition is true or false or both conditions are evaluated and their results are aggregated using maxjoin if the condition is unknown. Finally, with loop regions the subregion that is the loop's body is processed using ccount, but the number of executions of the parent region is multiplied by the number of loop iterations (obtained from $\mathbb{S}_I$).

Function maxjoin is used for joining the results of evaluations of two or more subregions where it is unknown which ones will execute. It takes n maps of some keys to numerical values as arguments and returns a similar map. Out of all values that share a key across the maps the maximum one is inserted into the resulting map. Function addjoin is used for aggregating the results of evaluations of a sequence of subregions that will execute one after another. It takes n maps of some keys to numerical values as arguments and returns a similar map. All values that share a key across the maps will be added together and the sum will be inserted into the resulting map under that key.

Functions at this stage of the analysis may need numerical values to be added or multiplied with the unknown value ω. If this happens, we treat it as an absorbing element, and the result of such an operation is always unknown. In a similar vein, the maximum of any set of numbers including ω is also unknown.

4 Precompiler Implementation

We implemented our precompiler as a tool for Atomic RMI using the Soot framework. The precompiler implementation consists of three elements: Jimple creation, upper bound analysis, and code generation (as shown in Fig. 9). The Jimple creator converts Java source code into the Jimple intermediate language—this is provided by Soot. The upper bound analysis deduces the information

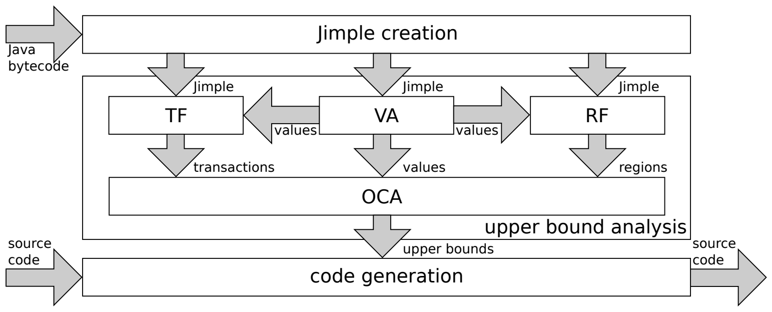

Fig. 9: Components and information flow in the precompiler.

about remote object calls within Jimple. It is divided into four analyses, each responsible for one pass over the code. The code generator instruments the input source code with instructions based on the information obtained by the analysis. The two components are described in more detail below.

4.1 Upper Bound Analysis

The upper bound analysis consists of value analysis (VA), region finding (RF), transaction finding (TF), and object call analysis (OCA). These are forward flow analyses implemented in Soot. Each of them makes one pass over the input code in the form of a CFG from a particular starting point (the main method, for instance). The implementation of each analysis defines a transfer function applied to each node of the CFG, a join operator for joining result sets, and initial result sets (see Fig. 4, Fig. 7, and Fig. 8).

Value analysis is the most complex of the analyses. It is the implementation of the algorithm in Section 3.2 such that the transfer function and join implement `eval` and `join`. The transfer function performs whatever action is needed for a given statement type (these are recognized via the type system). The result sets represent $\mathbb{S}_V$ and $\mathbb{S}_P$, $\mathbb{S}_I$, and $\mathbb{S}_D$ are passed via separate fields (for convenience). The implementation finds loops headers and bodies using Soot's built-in loop finder. Loops are processed by running the analysis repeatedly on a pruned copy of the CFG that contains only the statements from the loop and integrating the results into the original analysis. Recurrent calls are handled by finding all applicable method bodies, starting an analysis on each, and joining the results. A stack of calls keeps track of the depth of recursion and when to bound it.

The implementation of value analysis needs to take care of additional significant mechanisms that are obvious in the formalization and therefore glossed over. These include mechanisms for evaluating expressions. Expressions' arguments' types are recognized and the semantics appropriate to them is applied (i.e. `a + b` is addition if `a` and `b` are integers or concatenation if they are strings). All combinations of basic types (at least primitives and `Object`) and operators need to be implemented. We take the approach that operators are defined by classes and perform argument-dependent operations.

Region finder converts the CFG into a region graph in accordance with the algorithm in Section 3.3. The algorithm performs numerous graph searches like finding domination and post-domination relations within the graph (provided by Soot) and finding if particular paths exist within the CFG (e.g. whether all paths from a conditional expression leads to the end of the body or to a common post-dominator). RF creates a region hierarchy, were each region is characterized by its type and type-specific fields.

Transaction finder is a component that tracks Atomic RMI transactions and their components: it identifies the start and possible ends of transactions, remote objects used within, and transactions' preambles. These information are collected for use by OCA and marked in Jimple using the Soot tag system.

OCA is responsible for tallying remote objects calls as in Section 3.4. The implementation is straightforward: it accepts the data from the preceding analyses and uses them to traverse regions and identify those that make calls to remote objects. The number of executions of these regions is predicted and the counts are summed up with reference to particular remote objects.

The implementation must take into account the unknown values that may appear in the course of this analysis. These are implemented as a new type that allows any positive natural number or a value representing ω. The type also defines the maximum function and arithmetical operations using the unknown value (specifically addition and multiplication from Fig. 8).

4.2 Code Generation

The code generator for Atomic RMI modifies code on the lexical level using the suprema obtained from the execution of the upper bound analysis. Necessarily, in order for the code generator to modify the existing source code that source code must be available for analysis. The source is converted into tokens by the SableCC lexer [6] and then divided into lines.

The generator performs three passes over the collection of lines of tokens. In the first pass the generator locates transactions in the source code using the information provided by the transaction finding (TF) phase of code analysis. When found, all definitions in a transaction's preamble are marked for removal, with the exception of those which are followed by a comment string specifying them as manual overrides. The second pass inserts a line of code into each transaction's preamble for each identified remote object pertaining to that transaction (lines 2, 3 in Fig. 2). The insert contains a variable representing the remote object and a supremum on the number of method calls to that object, and it is built using on a simple template. All the inserts are marked for prepending to the beginning of the transaction. The final pass applies all the changes marked by the previous two passes to the tokens and they can then be written to the output stream.

5 Conclusion

Our work illustrates a static analysis for extracting the maximum number of times objects will be called in a fragment of code. Such information has a number of applications (we discuss them in Section 1) but we concentrate on using the upper bounds as input data for Atomic RMI. We have so far found that the analysis we implemented solves this problem satisfactorily for our purposes. The tree-like region-based intermediate representation allows to find all of the method calls within the code and the use of the absorbing unknown value produces conservative results when uncertain values are involved. Both of these guarantee that the statically derived upper bounds are correct, i.e. not lower than any actual number of method calls on a particular object. Apart from being conservative, the estimated upper bounds should also be as accurate as possible—as close to the actual number of executions as possible. For typical Atomic RMI transaction code, the analysis is able to handle most scenarios adequately.

The formalization of our algorithm and adherence to it simplified the implementation of the tool. The formalization was a blueprint for the join operators and transfer function of the individual data flow analyses which it defined their *modi operandi* and allowed us to concentrate on the details of the interfaces, data structures, etc. during implementation. Another advantage is that the correctness of the created tool is verifiable, extensible, and amendable by inspection and modification of the underlying algorithm, without an initial need to delve into the actual source code.

Our future work may include extending the current analysis with some additional refined analyses. In particular, there are ways to provide better identification of particular objects in the code, and more accurate ways to bound loops and recursion. The current algorithms may have trouble analyzing certain instances of input code accurately, especially when the depth of recursion or the number of loop iterations exceeds L. This may be resolved by following some of the approaches we list in Section 2. We also plan on exploring some elements of lock inference [2]. In particular, if an object's method were called an unknown number of times due to loops or recursion it would be possible to mark the last use of the remote object and to free it on that basis in run-time. These two methods could provide complementary mechanisms covering most scenarios. We also look forward to using our tool in new applications such as scheduling.

References

1. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison Wesley, 2nd edition, Aug. 2006.
2. D. Cunningham, K. Gudka, and S. Eisenbach. Keep off the grass: Locking the right path for atomicity. In *Proc. of CC '08*, LNCS 4959, Apr. 2008.
3. A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proc. of the 7th Workshop on WCET Analysis*, July 2007.
4. C. Ferdinand and R. Heckmann. AiT: Worst-case execution time prediction by static program analysis. In *Proc. of IFIP WCC '04*, Aug. 2004.
5. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. of EMSOFT '01*, LNCS 2211, Oct. 2001.

6. É. M. Gagnon and L. J. Hendren. SableCC, an object-oriented compiler framework. In *Proc. of TOOLS '98*, Aug. 1998.
7. J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system C programs. In *Proc. of WORDS '05*, Sept. 2005.
8. T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad. A modular worst-case execution time analysis tool for Java processors. In *Proc. of RTAS '08*, Apr. 2008.
9. M. Hicks, J. S. Foster, and P. Prattikakis. Lock inference for atomic sections. In *Proc. of TRANSACT '06*, June 2006.
10. N. Holsti, T. Långbacka, and S. Saarinen. Worst-case execution-time analysis for digital signal processors. In *Proc. of EUSIPCO 2000*, Sept. 2000.
11. Y.-F. Lee, B. G. Ryder, and M. E. Fiuczynski. Region analysis: A parallel elimination method for data flow analysis. *IEEE TSE*, 21:913–926, Nov. 1995.
12. Y.-T. S. Li and S. Malik. *Performance analysis of real-time embedded software*. Springer, Nov. 1998.
13. T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, 1999.
14. B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *Proc. of POPL '06*, Jan. 2006.
15. J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo. User-definable resource usage bounds analysis for Java bytecode. *ENTCS*, 253(5):65–82, Dec. 2009.
16. P. Pratikakis, J. S. Foster, and M. Hicks. Existential label flow inference via CFL reachability. In *Proc. of SAS '06*, LNCS 4134, Aug. 2006.
17. W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:4–13, 1992.
18. K. Siek and P. T. Wojciechowski. Statically computing upper bounds on object calls for pessimistic concurrency control. In *Proc. of the EC^2'10: Workshop on Exploiting Concurrency Efficiently and Correctly*, July 2010. Brief Announcement.
19. K. Siek, P. T. Wojciechowski, and W. Mruczkiewicz. Atomic RMI documentation. http://www.it-soa.pl/atomicrmi/, 2011.
20. J. Staschulat, J. Braam, R. Ernst, T. Rambow, R. Schlor, and R. Busch. Cost-efficient worst-case execution time analysis in industrial practice. In *Proc. of ISoLA '06*, Nov. 2006.
21. H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Syst.*, 18:157–179, May 2000.
22. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java optimization framework. In *Proc. of CASCON '99*, Nov. 1999.
23. R. Vallée-Rai and L. J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical Report 1998-4, McGill University, July 1998.
24. R. Wilhelm. Determining bounds on execution times. In *Handbook on Embedded Systems*, chapter 14. CRC Press, 2006.
25. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem-overview of methods and survey of tools. *ACM TECS*, 7(3), Apr. 2008.
26. P. T. Wojciechowski. *Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems*. Poznań University of Technology Press, 2007. Habilitation thesis.
27. P. T. Wojciechowski and K. Siek. Transaction concurrency control via dynamic scheduling based on static analysis. In *Proc. of WTM '12*, Apr. 2012.
28. F. Wolf, R. Ernst, and W. Ye. Path clustering in software timing analysis. *IEEE Trans. Very Large Scale Integr. Syst.*, 9:773–782, Dec. 2001.