Interprocedural Program Analysis for Java based on Weighted Pushdown Model Checking

Li Xin [1,2]

School of Information Science
Japan Advanced Institute of Science and Technology
Nomi, Japan

Ogawa Mizuhito [3]

School of Information Science
Japan Advanced Institute of Science and Technology
Nomi, Japan

Abstract

Based on the observation that "*program analysis is abstraction plus model checking*", this paper investigates pushdown model checking based approach on interprocedural program analyses for mono-thread Java. The running example is an interprocedual dead code detection under PER (partial equivalence relation) based abstraction. The prototype implementation combines SOOT as preprocessing to convert Java to Jimple and the Weighted PDS (pushdown system) library as the back-end model checking engine. With these existing tools, we developed an interprocedural dead code analyzer for mono-thread Java with around 1500 lines of Java codes. This analysis framework enables us a rapid prototyping for an interprocedual analysis design.

Key words: Weighted Pushdown System, Model Checking, Java, SOOT, Interprocedural Program Analysis

[1] This research is partially supported by "Fostering Talent in Emergent Research Fields" in Special Coordination Funds for promoting Science and Technology by Ministry of Education, Culture, Sports, Science and Technology, and by the "21st Century COE Program" for promoting Science and Technology by Ministry of Education, Culture, Sports, Science and Technology.

[2] Email: `li-xin@jaist.ac.jp`

[3] Email: `mizuhito@jaist.ac.jp`

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

1 Introduction

Program analysis is abstraction plus model checking[1,2]. With the progress on model checking algorithms and model checker implementations, this slogan enables us to separate design (abstraction) and implementation (backend model checking) of a program analysis.

Popular model checkers, such as Spin, NuSMV/SMV, are model checkers on finite state space; thus, a data flow analysis based on them is essentially intraprocedural[3]. Recently, several pushdown model checkers, such as Moped [10], Weighted PDS library [11], WPDS++ library [12], become available. They enable us to design an interprocedural analysis based on them.

The difficulty of interprocedural analysis is to handle:

- Interprocedural control flows, i.e., every calls and returns need to correctly match one another.
- Variable scope management, i.e., local variables in different procedure calls need to be correctly distinguished.
- Aliasing and parameter passing mechanisms, i.e., parameter alias and passing (e.g., call-by-value, call-by-reference) need to be correctly traced.

The first problem is naturally solved with pushdown systems by remembering the return point of a procedure call with the stack. The second problem can be solved by renaming variables for finite model checking; but for pushdown model checking, such solution results infinitely many variable names. For the last problem, proper abstractions are needed to handle interactions among procedures and the aliasing among variables.

This paper investigates interprocedural program analyses for mono-thread Java based on weighted pushdown model checking. Our prototype implementation using existing tools enables us a rapid prototyping of an interprocedural analysis design. The running example is an interprocedural dead code detection under PER based abstraction [8].

The prototype is implemented with SOOT as Java preprocessing and the Weighted PDS library as the back-end model checking engine. First, a Java program is converted into Jimple, which is a typed 3-address intermediate representation, for the ease of abstraction to a weighted pushdown system. During this conversion, the *call graph generation* is borrowed from SOOT [13] to handle virtual method calls. *pointer-to analysis* will be needed to handle the aliasing among variables for more precise analysis.

Second, perform abstraction to be accepted by Weighted PDS library. This phase is implemented as 1500 lines of Java code. Then, an interprocedual analysis is reduced to a generalized pushdown reachablity problem; this is solved by weighted pushdown model checking under a user-designed bounded idempotent semiring.

The rest of the paper is organized as follows: Section 2 briefly introduces weighted pushdown model checking problems. Section 3 describes abstraction

from Java programs to pushdown systems. The design of an interprocedural dead code detection based on weighted pushdown model checking is presented in Section 4. Section 5 shows the prototype framework and implementation. Section 6 and 7 discusses related work and future work.

2 Weighted Pushdown Model Checking

Definition 2.1 A **pushdown system** $\mathcal{P} = (Q, \Gamma, \Delta, q_0, w_0)$ is a pushdown automata regardless of input, where Q is a finite set of states, Γ is a finite set of stack alphabet, Δ is a mapping from $Q \times \Gamma$ to finite subsets of $Q \times \Gamma^*$, $q_0 \in Q$ is the initial state, $w_0 \in \Gamma^*$ is the initial stack contents. A *configuration* of $\mathcal{P}$ is a pair $\langle q, w \rangle$, where $q \in Q$ and $w \in \Gamma^*$. And $\langle q, w_i \rangle \hookrightarrow \langle q', w_j \rangle$ if $((q, w_i), (q', w_j)) \in \Delta$.

A pushdown system is a transition system with a finite set of control states and an unbounded stack. A weighted pushdown system associates a weight, in a bounded idempotent semiring to each transition rule of a pushdown system.

Definition 2.2 A **bounded idempotent semiring** is a quintuple $S = (D, \oplus, \otimes, 0, 1)$, where D is a set, 0 and 1 are elements of D, and $\oplus$ and $\otimes$ are binary operators on D such that

(i) $(D, \oplus)$ is commutative monoid with 0 as its neutral element, and where $\oplus$ is idempotent.

(ii) $(D, \otimes)$ is a monoid with 1 as the neutral element.

(iii) $\otimes$ distributes over $\oplus$, that is, $\forall a, b, c \in D$, $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$.

(iv) 0 is an annihilator with respect to $\otimes$, that is, $\forall a \in D, a \otimes 0 = 0 \otimes a = 0$.

(v) The partial order $\sqsubseteq$ defined as: $\forall a, b \in D, a \sqsubseteq b$ iff $a \oplus b = a$

Intuition behind $\otimes$ and $\oplus$ of weights is:

- A weight intends a function to represent how a property is carried at each step of program execution.
- $f \otimes g$ intends the composition $g \circ f$.
- $f \oplus g$ intends the conservative approximation at the meet of two dataflow, such as in the case of a conditional sentence.

Thus, a weighted pushdown system naturally represents how properties are carried at each transition, which is an abstraction of a step of program execution.

Definition 2.3 A **weighted pushdown system** is a triple $W = (\mathcal{P}, S, f)$, where $\mathcal{P} = (Q, \Gamma, \Delta, q_0, w_0)$ is a pushdown system, $S = (D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring, and $f : \Delta \rightarrow D$ is a function that assigns a value from D to each rule of P.

The generalized reachability of a weighted pushdown system is to examine all paths leading to a set of configurations of interest.

Definition 2.4 Given a weighted pushdown system $W = (\mathcal{P}, S, f)$, where $\mathcal{P} = (Q, \Gamma, \Delta, q_0, w_0)$ is the pushdown system, and $S = (D, \oplus, \otimes, 0, 1)$ is the bounded idempotent semiring. Assume $\sigma = [r_0, ..., r_k]$ be a sequence of pushdown rules, where $r_i \in \Delta, 0 \leq i \leq k$, and $v(\sigma) = f(r_0) \otimes ... \otimes f(r_k)$, let path($c$,$c'$) denotes the set of all rule sequences that transform c into c'. Let $C \subseteq P \times \Gamma^*$, the **generalized pushdown reachability problem** is to find for each $c \in P \times \Gamma^*$:

$$\delta(c) = \bigoplus \{v(\sigma) | \sigma \in path(c, c'), c' \in C\}$$

An algorithm for solving the generalized reachablity problem is presented in [5]. It is implemented as libraries, Weighted PDS [11] and WPDS++ [12]. We will apply the former in our prototype implementation.

3 Abstraction from Java programs to Pushdown Systems

Abstraction is a fundamental step for analyses based on model checking. The primary task is how to abstract a problem and encode it into the underlined model (transition system) of the back-end model checking engine. The precision and correctness of an analysis depends on that of abstractions.

Example 3.1 is used to illustrate our scenario thoroughout the paper.

Example 3.1 Figure 1 (a) shows a Java program with three classes: `Example`, `Call`, and `CallSuper`. The class `Call` inherits the class `CallSuper` and redefines the *call* method for calculating the factorial of some integer `a`, with an integer `b`. At the virtual call site of line number `n`, the method *call* of the class `Call` should be invoked at runtime. Note that `b` is a dead parameter.

3.1 Interprocedural Control Flows

Instead of abstracting programs directly, control flow graphs (CFG) are first prepared for each procedure. However, it is not easy to get a precise interprcedural control flow graph (or supergraph) for Java, due to polymorphism and dynamic binding of virtual method calls.

In Example 3.1, reference variable *cs* may point to instances of either the class *CallSuper* or *Call* at runtime. At the virtual call site n, *cs* will invoke the method *call* from the class *Call* instead of the declared type class *CallSuper*. To identify virtual method calls, type analysis is usually made to compute the possible type sets of reference variables at virtual call sites. At the first stage, we make use of the results of *call graph* generated by SOOT. It is a set of possible call edges among procedures with consideration on virtual method calls.

Figure 1 (b) shows part of the supergraph of the Java program in Example 3.1. Compared with CFGs from intraprocedural cases, three more kinds of edges are added [6]:

- A *call edge* from the call site to the entry point of callee procedure.
- A *return edge* back from the callee procedure to the return point of caller procedure.
- A *call to return edge* from call site to the return point of this call.

Note that, the control flow corresponding to the *call to return edge* does not exist in the actual program execution. However, this edge depict the semantical changes of data flows when a call is invoked. During a procedure call, local variables in the caller procedure keep unchanged and can take a short-cut on the control flows by introducing the *call to return edge*. *call to return edge* will be kept in the abstractions for our running example.

3.2 Abstract Parameter Passing and Return

In an interprocedural case, a problem is how to abstract interactions among procedures through parameter passings and return values. We use an *abstract parameter passing* mechanism to handle these interactions, which is realized by introducing two extra kinds of global variables for procedure parameters and return values respectively. In particular, variables for procedure parameters are characterized by their positions declared instead of only by names.

This approach is shown in Figure 1 (b) with Example 3.1, in which only the static class member f is "global". To depict the interactions among procedures, two extra kinds of global variables are introduced: the integer parameter variables **call_arg0**, **call_arg1** for the method *call* and the integer return variable **r_int** . The introduction of parameter and return variables correctly depict the localness of local variables. Whenever a procedure is invoked, the corresponding procedure variables are assigned if they exist. Whenever a procedure invocation returns with some non-void value, the global return variable with coincident type is assigned. Local variables within one procedure are always unseen to others.

Each edge in the supergraph is labelled with a transfer function. It is usually a set of mappings from program variables to some abstract data domain. In Figure 1 (b), *top* could be understood as non-exist values and used for later generation of the *exploded supergraph*. Please refer to [6] for formal definitions. Some treatments on transfer functions need to be mentioned here:

- When a procedure is invoked (*call edge*), all local variables in the caller procedure are assigned to *top*.
- When an invoked procedure returns(*return edge*), all local variables in the callee procedure are assigned to *top*.
- All global variables are assigned to *top* along *call to return edge*.

Although Java always uses call by value passing mechanism, a method can still change the state of an object reference parameter. At the first stage, only parameters of primitive types, that is numbers or Boolean values, are handled. In our abstractions, procedures are distinguished by declared class and procedure signatures like names and parameters. But the analysis is still context-sensitive in the sense that parameters passed to procedures are distinguished. The analysis is also field-sensitive. An instance field is distinguished by its name, type, declared class and the class instance it belongs to. To further identify the aliasing among object reference variables, pointer-to analysis in SOOT will be needed.

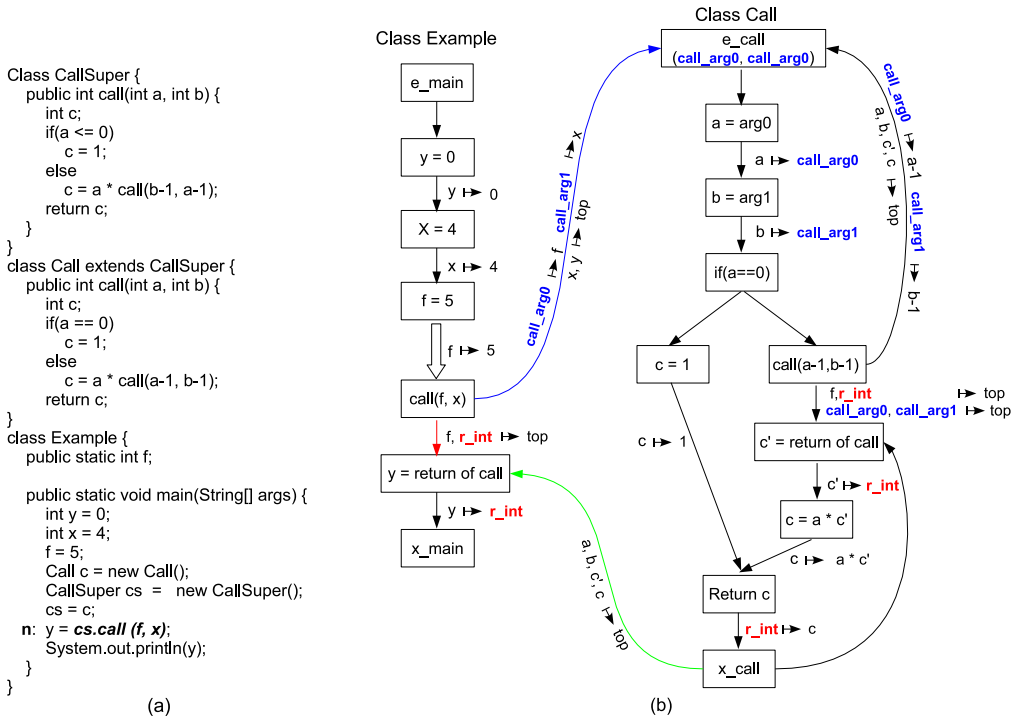

Fig. 1. (a) A Java program fragment with virtual method calls (b) A supergraph with abstract parameter-passing

3.3 Encoding to Pushdown Systems

Recall the usual abstractions for finite model checking on programs, the product of *global variables* and *local variables* and *program control points*(nodes of the CFG) is encoded as a state of the finite transition system. For pushdown model checking, a stack enables simulation of the runtime stack of programs. A usual encoding is, such as in MOPED, *global variables* are encoded as control states, and the product of *local variables* and *program control points* is encoded as stack symbols. After all, the problem to be analyzed decides the encoding choice.

To examine the property in terms of single variables, an edge in the super-

graph is exploded into a set of edges with every variables as ends by a so-called "explosion transformation", which results in an *exploded supergraph*[6]. Especially, there is no edge associated with a variable where it is assigned to be *top*. Provided with an exploded supergraph G of the program to be analyzed, the encoding of the pushdown system for our running example is:

- All program variables (global variables and local variables) are encoded as the set of control states.
- Program control points are encoded as stack symbols.
- Each edge in G is encoded as a pushdown rule according to the following cases:
  - $\langle q, w_i\rangle \hookrightarrow \langle q', w_k\ w_j\rangle$
    A call edge from node w_i to w_k with w_j as return point.
  - $\langle q, w_i\rangle \hookrightarrow \langle q', w_j\rangle$
    An intraprocedural edge from node w_i to w_j.
  - $\langle q, w_i\rangle \hookrightarrow \langle q', \varepsilon\rangle$
    A return edge from exit node w_i to corresponding return points.

4 Interprocedural Dead Code Detection

A dead code detection usually follows the *used-and-defined* approach [7]. That is, some variable x of interest is evaluated with predicates "$Used_x$" or "$Defined_x$" for every states of the transition system, and then whether the transition system violates the property of "$!Used_x$ **W** $Defined_x$" (**W** is weak until) is model checked for all transition sequences.

Provided with pushdown model checking, an interprocedual dead code detection based on this approach usually concerns only on global variables. For example, when variables are passed as parameters to some procedures, these variables are basically considered as "*Used*", regardless of its effect on the result of computation.

Our running example targets on dead code detection in a more semantical sense. The basic intention behind is quite simple: A line of code is dead as long as its removal does not affect the final result of interest. Instead of *used-and-defined* approach, We apply PER-based abstraction [8] as a forward abstract intepretation [4]. This approach naturally detects transitivity of dead codes. For instance, our method directly detects L1 in the Example 4.1 is a dead code. While the *used-and-defined* based approach cannot detect it unless L2 is removed.

Example 4.1 A code fragment:

L1 x := 1; L2 y := x+1; L3 x := 3; L4 y : = x+1;

4.1 PER based Data Abstraction

A *partial equivalence relation* R on a set S is a transitive and symmetric relation $S \times S$. If R is reflexive, it is an equivalence relation. Our abstract data domain L is a 2 point domain based on PER [8] [4], defined as

$$L = \{\text{ANY}, \text{ID}\}, \quad \text{with the ordering} \quad \text{ANY} \supset \text{ID}$$

With the original domain as integer $\mathbf{Z}$, the concretisation γ of L is defined as

$$\gamma \ \text{ANY} = \{(x, y) \mid x, y \in \mathbf{Z}\}$$

$$\gamma \ \text{ID} = \{(x, x) \mid x \in \mathbf{Z}\}$$

Where ANY is interpreted as anything, and ID is interpreted as values being fixed. It is easy to see that

$$\forall l \in L, \ \gamma \ l \text{ is a PER}$$

A finite set of *transfer functions* $\mathcal{F} : \ L \to L$ is defined as:

$$\mathcal{F} = \{\lambda x.x, \lambda x.\text{ANY}, \lambda x.\text{ID} \ \mid x \in L\}$$

Let $f_0 = \lambda x.\text{ANY}$, $f_1 = \lambda x.x$, and $f_2 = \lambda x.\text{ID}$, it is obvious that

$$\forall x \in L, \ f_0 \ x \supset f_1 \ x \text{ and } f_1 \ x \supset f_2 \ x$$

We define an ordering $\sqsubset$ on $\mathcal{F}$ as the reverse of $\supset$ as

$$\lambda x.\text{ANY} \sqsubset \lambda x.x \sqsubset \lambda x.\text{ID}$$

The intention for the interprocedural dead code analysis under this data abstraction is: if a variable is assigned to be ANY at some line of code, and the result is still ID, then this line of code is considered as dead.

The abstract interpretation p' of a primitive operation p is derived as $p'(l_1) = l_2$ where $l_2 \in L$ is the least PER that inculdes $\{p(x) \mid x \in l_1\}$ for each $l_1 \in L$. Then, the result is computed by the least fixed point computation under the oredering $\sqsubseteq$, which will be performed by weighted pushdown model checking.

4.2 A bounded idempotent semiring

A bounded idempotent semiring with L as the abstract data domain is defined as:

[4] [11] uses a 3 point abstract domain {ANY,ID,BOT}. Since our focus is on "irrelevance" not on "strictness", BOT is left out.

(i) Weighted domain D is defined as

$$D = \{\lambda x.x, \lambda x.\textsc{any}, \lambda x.\textsc{id}, \textsc{zero} \mid x \in L\}$$

where ZERO is naturally interpreted as that the program execution is interrupted by an error.

(ii) 1 is defined as $\mathrm{id} = \lambda x.x$

(iii) 0 is defined as ZERO

(iv) The $\otimes$ operator composes the effects by transfer functions along one path. The operation $\otimes$ on D is defined as

$$\forall d \in D,\ \textsc{zero} \otimes d = d \otimes \textsc{zero} = \textsc{zero}$$

$$\forall d \in D,\ \lambda x.x \otimes d = d \otimes \lambda x.x = d$$

$$\lambda x.\textsc{any} \otimes \lambda x.\textsc{id} = \lambda x.\textsc{id}$$

$$\lambda x.\textsc{any} \otimes \lambda x.\textsc{any} = \lambda x.\textsc{any}$$

$$\lambda x.\textsc{id} \otimes \lambda x.\textsc{any} = \lambda x.\textsc{any}$$

$$\lambda x.\textsc{id} \otimes \lambda x.\textsc{id} = \lambda x.\textsc{id}$$

(v) The $\oplus$ operator combines effects on the property domain by transfer functions from different branches. The operation $\oplus$ on D is defined as

$$\forall d \in D,\ \textsc{zero} \oplus d = d \oplus \textsc{zero} = d$$

$$\forall d \in D,\ \lambda x.\textsc{any} \oplus d = d \oplus \lambda x.\textsc{any} = \lambda x.\textsc{any}$$

$$\lambda x.\textsc{id} \oplus \lambda x.\textsc{id} = \lambda x.\textsc{id}$$

$$\lambda x.x \oplus \lambda x.x = \lambda x.x$$

$$\lambda x.\textsc{id} \oplus \lambda x.x = \lambda x.x \oplus \lambda x.\textsc{id} = \lambda x.x$$

Distributivity of $\otimes$ over $\oplus$ is easily checked.

With the bounded idempotent semiring, the interprocedural dead code detection works as follows: select some line of code and assign the weight of its transition associated to be $\lambda x.\textsc{any}$, this line of code is dead if the weight of the result is either $\lambda x.x$, or $\lambda x.\textsc{id}$. The soundness of this analysis is guaranteed by the facts that:

- the construction of PER-based forward abstract interpretation, and
- the conservative approximation of the definition of $\oplus$.

5 The Prototype Framework and Implementation

Our prototype is implemented as shown in Figure 2. It is developed with SOOT for Java preprocessing and the WPDS library as the back-end model

checking engine. Jimple, a typed stackless 3-address intermediate representation, is the analysis target. To make use of the existing tools enables us a rapid prototyping for our analysis design.

The analysis procedure is illustrated with Example 3.1. In *phase 1*, Java programs are first translated into Jimple with SOOT. Some sample result is shown in (b). The *phase 2* is for abstraction. The *call graph*, a set of possible call edges among procedures generated by SOOT, is borrowed here. As shown in (c), the virtual method call in Example 3.1 is resolved and the corresponding call edge is given. The output of abstraction, as shown graphically [5] in Figure 3, is a weighted pushdown system that will be model checked in *phase 3* by the back-end model checking engine. A bounded idempotent semiring, specific to the analysis of interest, is designed (Section 4.2) and implemented with the weighted PDS library beforehand.

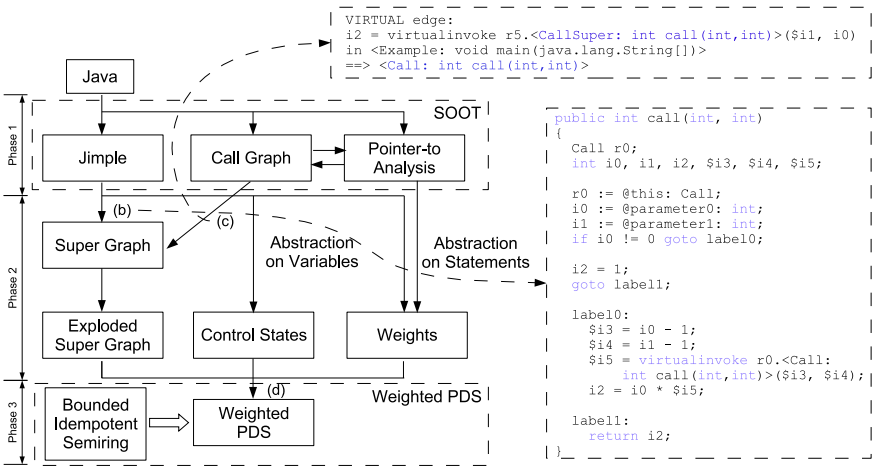

Fig. 2. A Prototype Framework

6 Related Work

Our abstraction from Java programs to weighted pushdown systems basically follows the approach proposed in [5]. In [5], several examples of interprocedural analyses based on weighted pushdown model checking have been investigated, such as live variables, linear constant propagation. For instance, WPDS++ library provides an example for interprocedural live variable analysis. The weighted domain D is defined as $D = \{\lambda S.S \setminus KillSet(i) \cup GenSet(i) \mid i \in \mathbf{N}\}$, where S is the finite set of variable alphabet, $\mathbf{N}$ is labels of all the program statements. These previous case studies basically consider global variables only, whereas our analysis handles local variables and interactions among procedures.

[5] This graphical drawing is part of our implementation. It is automatically generated for debugging purpose.

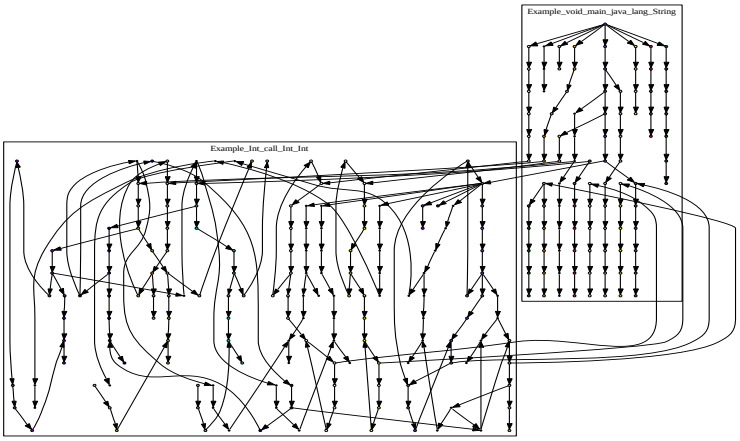

Fig. 3. An exploded Interprocedural Control Flow Graph

Bandera [14] is a a tool set for analysis generation for Java programs. It provides model constructions from Java programs to the inputs of several popular finite model checkers. Bandera also works on Jimple and it borrows the implementation framework of SOOT to realize their own JCCC compiler. JCCC provides extra functions for tracking errors back to Java source codes. Currently, our analysis does not support such facility to pull back the result of an analysis on Jimple to that on Java. By substituting Jimple compiler from SOOT to JCCC, this will be covered.

jMoped [15] is a translator from Java bytecode to pushdown systems that are the input of MOPED. Compared with Java bytecode, Jimple would be more suitable for designing abstraction by its features, such as explicit statements and control flow structures.

Weighted pushdown systems are further extended to *extended weighted pushdown systems*(EWPDS) [9]. EWPDS excludes the *call to return edge* in the supergraph abstraction. Instead, a merge function is proposed to restore the local variables of caller procedures when callee procedures return.

7 Conclusions and Future Work

This paper investigated interprocedural program analyses for mono-thread Java based on weighted pushdown model checking. Our prototype implementation using existing tools enables us a rapid prototyping of an interprocedural analysis design. The running example is an interprocedural dead code detection under PER based abstraction [8].

Future work will be:

- More case studies of interprocedual program analyses; the first target will be *call graph generation*. Current our prototyp borrows the *call graph generation* from SOOT for more precise abstraction. However, this call graph generation is somewhat ad-hoc with the aid of *pointer-to analysis* in SOOT, and is a time-consuming task. We expect that this can be efficiently com-

puted by weighted pushdown model checking.

- Based on case studies, clarify suitable (hopefully automatic) supports for designing and implementing abstraction design. For instance, Bandera [14] has a facility for automatic generation of an intraprocedural analysis from user-specified abstraction.
- Applying similar methodology to other languages, such as ML. We are planning to cooperate with Interoperable ML project. [6] .
- More theoretical study on systematic derivation of a bounded idempotent semiring from an abstraction. That is, what kind of abstractions can be encoded into weighted pushdown systems by keeping sound property need to be further examined.

References

[1] B. Steffen. Data Flow Analysis as Model Checking. In TACS '91, LNCS 526, pages 346–365. Springer, 1991.

[2] D.A. Schmidt. Data ow analysis is model checking of abstract interpretation. *In Proceedings of the Twenty Fifth Annual Symposium on Principles of Programming Languages*, pages 38-48. ACM Press, 1998.

[3] Aho, A. and Sethi, R. and Ullman, J.D. Compilers: Principles, Techniques, and Tools. Addison–Wesley, 1986.

[4] Cousot, P. and Cousot, R., Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints, *Proc. 4th ACM Symposium on Principles of Programming Languages*, pp.238–252, 1977.

[5] Reps, T., Schwoon, S., Jha, S., and Melski, D., Weighted pushdown systems and their application to interprocedural dataflow analysis. Science of Computer Programming, 58(1C2):206–263, October 2005.

[6] Sagiv, M., Reps, T., and Horwitz, S., Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167 (1996), 131–170.

[7] Lacey, D., Jones, N. D., Wyk, E. V. and Frederiksen, C. C., Proving correctness of compiler optimizations by temporal logic, *Proc. 29th ACM Symposium on Principles of Programming Languages*, *Symposium on Principles of Programming Languages*, pp. 283–294, 2002.

[8] Hunt, S., PERs Generalize Projections for Strictness Analysis (Extended Abstract), *Functional Programming: Proc. 1990 Glasgow Workshop*, pp. 114–125 (1991), Springer-Verlag.

[6] `http://www.pllab.riec.tohoku.ac.jp/õhori/research/iml-e.html`

[9] Lal, A., Balakrishnan, G., and Reps, T., Extended weighted pushdown systems. In Proc. Computer-Aided Verification, 2005.

[10] http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/

[11] http://www.fmi.uni-stuttgart.de/szs/tools/wpds/

[12] http://www.cs.wisc.edu/wpis/wpds++/

[13] http://www.sable.mcgill.ca/soot/

[14] http://bandera.projects.cis.ksu.edu/

[15] http://www.fmi.uni-stuttgart.de/szs/tools/moped/jmoped/