Integrating and Scheduling an Open Set of Static Analyses

Michael Eichberg, Mira Mezini, Sven Kloppenburg, Klaus Ostermann, Benjamin Rank
Darmstadt University of Technology, Germany
{eichberg,mezini,kloppenburg,ostermann,rank}@st.informatik.tu-darmstadt.de

Abstract

To improve the productivity of the development process, more and more tools for static software analysis are tightly integrated into the incremental build process of an IDE. If multiple interdependent analyses are used simultaneously, the coordination between the analyses becomes a major obstacle to keep the set of analyses open. We propose an approach to integrating and scheduling an open set of static analyses which decouples the individual analyses and coordinates the analysis executions such that the overall time and space consumption is minimized. The approach has been implemented for the Eclipse IDE and has been used to integrate a wide range of analyses such as finding bug patterns, detecting violations of design guidelines, or type system extensions for Java.

1 Introduction

Static analyses are used to check that certain desired properties hold before software is deployed. Traditionally, static analyses check properties that are independent of an application's domain, such as array index out of bounds, null-pointer dereferences or buffer overflows. Recently, the attention is shifting towards domain and project specific analyses for, e.g., Web and EJB applications [15, 22, 25, 26], to check the correct usage of specific APIs [3], to find violations of security constraints [23], and to enforce design or programming guidelines [20].

Static analysis tools that support only a fixed set of analyses [1, 16, 22, 24] are not well-suited for project-specific analyses. Hence, we propose an open platform that allows to add or remove analyses as needed. Furthermore, we feature a tight integration of the platform into the incremental build process of an IDE. The user is allowed to select a subset of the available analyses to run along the incremental build process. As a result, the developer receives immediate feedback on the effect of source code changes.

An open platform can replace a multitude of specialized tools. In doing so, it has the potential to reduce the engineering effort for developing new analyses and to support more efficient use of computational resources needed to execute the analyses, which is an important prerequisite for integration into the incremental build. However, in order for this potential to become reality, open platforms require an open data model to store the results of analyses, which in turn requires means of coordination between analyses that write and read the data model. In the following, we elaborate on the statements made in this paragraph in terms of the sample analyses shown in Tab. 1 along with the data they depend on.

The table illustrates that static analyses differ widely in the data they require, but they also share subsets of data. For example, both the SA and the CFT checker require data flow information. Each analysis could of course compute all the data it requires from the raw source code or from a generic representation of the project. However, implementing and running several instances of an algorithm for data flow analysis wastes both engineering effort and computational resources. Furthermore, analyses may consume only information about a part of the project. For example, the EH analysis requires only information about the interfaces of Java classes; method bodies or other artifacts such as deployment descriptors are irrelevant. Hence, it is a waste of resources to reify a generic representation of the entire software.

To cope with the issues stated in the previous paragraph, it is desirable to divide the analyses into small modular producer-consumer units. Analyses such as SA and CFT can share the results produced by a base analysis for data flow information; similarly, EH can consume the results of an analysis that produces information about the interfaces of Java classes only. This requires that analyses are run in a well defined order to satisfy their data-producer-consumer relations. These relations cannot, however, be expressed by a total or-

ID	Description	Required Data
NSF	Searches for `finalize` methods that do not call `super.finalize`.	control flow graph (CFG)
EH	Searches for Java classes overriding either `equals(boolean)` or `hashCode()`, but not both.	interfaces of Java classes
SA	Searches for `String.append(..)` invocations where the return value is ignored.	data flow information
CTAV	Searches for Enterprise Java Beans that use declarative and programmatic transaction demarcation [8].	type hierarchy, method bodies, EJB deployment descriptors
CFT	Realization of Confined Types[12] based on Java annotations.	type hierarchy, type hierarchy changes, data flow information, public interfaces of libraries

Table 1. Sample analyses and the data they depend on

der, since the set of analyses is open. It is also desirable to automatically select and run only analyses that produce information consumed by those analyses directly selected by the user. A base analysis, e.g., for getting the type hierarchy, should only run if its result is needed by a user selected analysis.

The producer-consumer dependencies cannot be represented by a partial order graph either. For better performance, some analyses should be able to transform and modify existing analysis data instead of generating new data. Furthermore, several analyses that generate the same information can co-exist within the platform and it should be ensured that at most one of them is run. Both cases are not expressible by a partial order. Last but not least, to leverage modern multi-processor architectures, it is also desirable to parallelize analysis executions whenever possible.

Our approach coordinates analyses based on solving constraint systems that represent the dependencies between the analyses. The coordination unit, which we call the scheduler, treats analyses as modules that write, read or maintain the open data model. Each analysis describes its properties and dependencies in a special analysis specification language (ASL). These specifications are mapped onto a constraint system which is fed to a constraint solver. Adding objective functions to the set of constraints allows to calculate a schedule that is optimal with regard to the number of internal analyses to run. We have implemented our approach as the core of the open static analysis platform Magellan [10].

The remainder of the paper is organized as follows. Section 2 presents the data model used as the foundation for the analysis specification language (ASL) presented in Section 3. Section 4 discusses how a valid schedule can be calculated. Section 5 evaluates the implementation of the scheduler. Section 6 discusses related work. Section 7 summarizes the paper.

2 The Analysis Data Model

The analysis data in our platform is stored in the *whole-program database* (WPDB). The WPDB is an object graph built-up cooperatively by the executed analyses. The WPDB has a set of designated root objects which are called *facts*. The architecture of the fact objects is shown within the box on the left-hand side of Fig. 1, entitled "Class diagram of the WPDB". We distinguish between three different types of facts.

For each resource (file) in the project a *document fact* is created (an object of class `DocumentFact` in Fig. 1), which keeps a reference to the underlying file. A document fact enables analyses to attach derived information to the set of facts (implementations of the `IFact` interface) it aggregates. For example, a representation of a Java class file is a typical example of a fact aggregated within a document fact. Instances of the class `ClassFile` — within the box in the middle of Fig. 1 — represent individual Java class files produced by the Java Bytecode Analysis Toolkit BAT [11].

A document fact is automatically created, added to, and removed from the database corresponding to the type of action on the underlying file. The set of all document facts that are created or removed from the database in a build is also directly made available to the analyses. This enables analyses which can perform their work incrementally per document to process only the delta to the previous build.

Information that cannot directly be associated with specific documents is stored in the database using *whole program facts*. A whole program fact always needs to be maintained by the analysis that creates it. After a full build, the analysis has to re-create the whole program fact; after an incremental build, the analy-

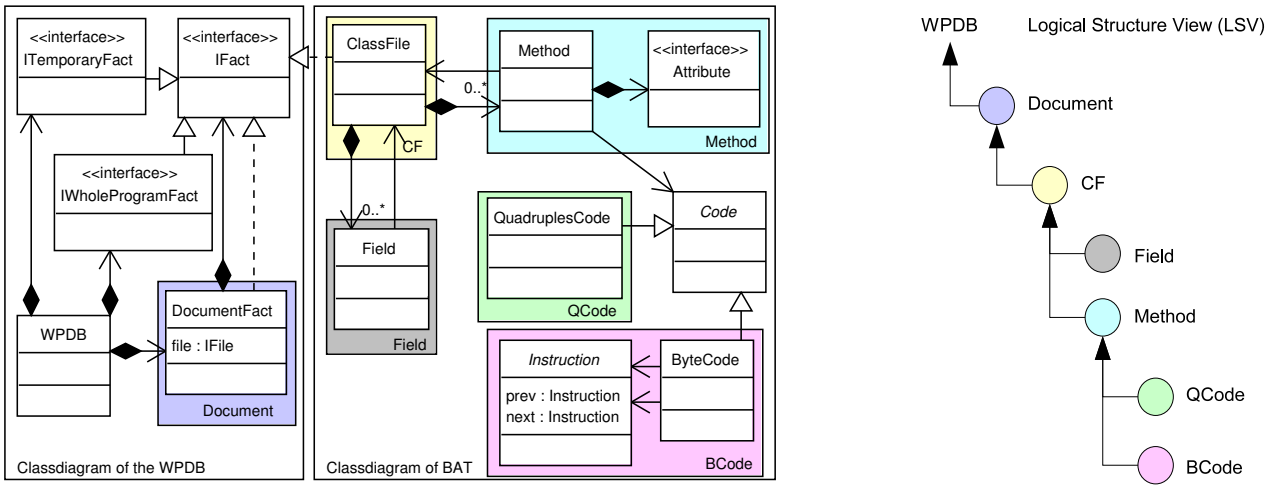

Figure 1. A part of the LSV and its mapping to the WPDB

sis has to bring the information up-to-date to reflect the current project's state. For example, an analysis that makes the type hierarchy information available has to update the type hierarchy whenever the developer makes a change that invalidates the "old" type hierarchy. Information that is only valid during a build step is stored in *temporary facts*. All temporary facts are automatically deleted before each build. For example, a type hierarchy analysis could also make information about the changes to the type hierarchy available for the benefit of subsequent analyses. However, this information is only valid for the current build.

Data dependencies in the WPDB are expressed in the *logical structure view* (LSV). The logical structure view is a directed acyclic graph. Every node in the LSV stands for a part of the WPDB, whereby a part of the WPDB can be a selection of objects or, even more fine-grained, a selection of field values of the objects in the WPDB. We call nodes in the LSV *entities*. Fig. 1 shows a part of the LSV on the right-hand side. Also, its mapping to the corresponding parts of the WPDB is shown by the gray boxes around elements of the WPDB and BAT class diagrams. Consider for an example the gray box labeled "Method" surrounding the class `Method` and `Attribute` in the BAT class diagram. This boxing states that a LSV method entity is mapped to a WPDB method and all its attributes. We refer to entities in the LSV by using paths in the LSV starting at the WPDB vertex; e.g., to refer to the BCode entity we write: *Document*/CF/Method/BCode.

Edges in the LSV express data dependencies as follows: *If data in the WPDB is changed that belongs to an LSV entity v, then all data in the WPDB that is invalidated by the change is associated to entities w such that there is a path from w to v in the LSV.* Declaring an entity w as dependent on an entity v implies that there are no conflicts between an analysis that changes the data associated to w or any of its dependent entities and those that just read the data associated to v. Further, analyses that access siblings do not conflict. For example, `Field` and `Method` are declared as dependent entities of `CF`. Hence, an invalidation of the information on a class entity automatically invalidates information on its fields and methods. But, there are no conflicts between analyses that process `Field` and `Method` entities respectively. These properties are leveraged by the scheduler to parallelize analysis executions. Though a fined-grained LSV increases the possibilities for parallelization, it decreases the ease of describing and understanding the dependencies among analysis data.

The LSV is derived from the set of analysis specifications, as detailed in Section 4. The mapping between the LSV and the WPDB is specified informally in the documentation of the respective WPDB elements.

If the user of the platform would like to extend the predefined LSV and WPDB, for example to make the intra-procedural control-dependence graphs (CDG) of methods available, he first needs to determine where to store the information. Our representation for class files enables extension of its object graph by means of attributes. Hence, the user could implement a set of classes for managing the CDG and store instances of them as attributes of the corresponding code object. Since the CDG is derived from the code of the method, the LSV is extended with a new node CDG which is associated with all CDG objects in the WPDB, and an edge to, e.g., BCode to represent the dependency.

```
1  analysis CFP (* creates class file representation *)
2    writes Document/CF, Document/CF/Field, Document/CF/Method, Document/CF/Method/BCode
3  analysis DDP writes Document/EJBDD (* creates EJB Deployment Descriptor representation *)
```

Listing 1. Analyses that make the base representations available (Processors)

```
1  analysis BCFG writes Document/CF/Method/BCode/CFG (* creates the control-flow graph (CFG) *)
2  analysis BtoQ (* transforms the Bytecode in 3-address SSA form *)
3    reads Document/CF/Method/BCode
4    invalidates Document/CF/Method/BCode
5    writes Document/CF/Method/QCodeSSA
6  analysis LIB (* maintains the repository of used library classes *)
7    reads Document/CF/Method/BCode
8    reads-global Document/CF
9    maintains Library/CF/Field_NON_PRIVATE, Library/CF/Method_NON_PRIVATE
10 analysis TH (* maintains the type hierarchy *)
11   reads-global Document/CF, Library/CF
12   writes-temporary TypeHierarchyChange
13   maintains TypeHierarchy
14 analysis CTA1 (* programmatic and declarative transaction demarcation is used *)
15   reads Document/EJBDD
16   reads-global TypeHierarchy, Document/CF/Method/BCode
17   writes CTAViolations
18 analysis CTA2 (* alternative CTA analysis *)
19   reads Document/EJBDD
20   reads-global TypeHierarchy, Document/CF/Method/QCodeSSA
21   writes CTAViolations
```

Listing 2. Analyses that read, create and transform the database (Base Analyses)

```
1  analysis NSF reads Document/CF/Method/QCode/CFG (* finalize does not call super.finalize() *)
2  analysis EH reads Document/CF/Method (* equals and hashcode have to be implemented pairwise *)
3  analysis SA reads Document/CF/Method/QCodeSSA (* String.Append() must not be ignored *)
4  analysis CFT (* realizes Confined Types *)
5    reads TypeHierarchyChange
6    reads-global TypeHierarchy, Document/CF/Method/QCodeSSA,
7           Library/CF/Method_NON_PRIVATE
8  analysis CTAV reads CTAViolations (* wraps CTA and CTA2 *)
```

Listing 3. Analyses that just read the database (Checkers)

Figure 2. Examples of analyses specifications developed to test the framework

AS	::= **analysis** ID STATEMENT*
STATEMENT	::= DEPENDENCY PATH*
DEPENDENCY	::= **reads-global** | **reads** | **writes** | **invalidates** | **maintains** | **writes-temporary**
PATH	::= ID [/ PATH]

Figure 3. The ASL grammar

3 Specifications of Analyses

The *analysis specification language* (ASL) is used to declare the *data* required and provided by each analysis in terms of the logical structure view described in the previous section. The ASL supports six different types of dependencies as shown in the ASL grammar in Fig. 3. Listing 1-3 (in Fig. 2) illustrate the specification of the sample analyses from Tab. 1.

A `reads` dependency on some LSV entities means that the analysis works incrementally on the specified input data. For example, the EH checker (Listing 3, Line 2) specifies that the analysis will read the entities referred to by the path expression *Document*/CF /Method. A `reads−global` dependency, on the other hand, means that the analysis needs data of the specified kind for *all* documents, not just those processed in the current build. The current implementation of the type hierarchy analysis, e.g., needs access to all class files, not just those changed; hence, the corresponding `reads−global` dependency in Listing 2, Line 11.

A `writes` dependency specifies that the analysis provides data of the specified type for documents that are changed in the current build step only. For example, the DDP analysis (Listing 1, Line 3) specifies that it writes the EJBDD entity and implicitly reads the preceding entities, i.e. the *Document* entity. If all path elements would be considered as written it would not be possible to have a second analysis that `writes` a dependent entity, but which does not write the preceding entities; e.g., it would not be possible to specify that an analysis just writes a BCode's CFG and not the BCode. A `writes−temporary` dependency is used for data that is automatically invalidated (and hence removed by the platform) before the next build. For example, the type hierarchy analysis (Listing 2, Line 10) also provides information about changes to the type hierarchy between the current and the previous build. Since this information is only valid for one specific build step, it is declared using `writes−temporary`. As in case of `writes`, only the last entity of the path is written and the previous entities are read.

The `invalidates` dependency specifies that after executing the analysis the last entity referred to by the given path expression is no longer valid. This is usually the case if an analysis provides its result by transforming existing data in the WPDB. For example, the analysis which transforms a method's bytecode representation into the 3-address based representation (Listing 2, Line 2) specifies that the BCode entity will become invalid when the analysis is executed because the analysis changes the existing data in the WPDB.

Finally, `maintains` is used by an analysis to declare that it creates an entity and updates it during the following builds. For example, the type hierarchy analysis declares to maintain (Listing 2, Line 13) the TypeHierarchy entity.

Analyses may overlap in both their input and output data. If multiple analyses are present that can produce the same data, the scheduler decides which of these analyses will be executed. There can also be multiple analysis specifications for the same analysis. This can be used to express that an analysis needs one entity *or* another kind of entity. For example, the checker for detecting conflicting transaction demarcations (CTAV - Listing 3, Line 8) needs either the byte code (BCode - Listing 2, Line 14) or the SSA-transformed code (QCodeSSA - Listing 2, Line 18), hence there are two analysis specifications for this analysis. Such alternatives give the scheduler more leeway in scheduling an analysis.

An analysis specification also serves as a contract on what the analysis implementation is allowed to do with the WPDB. The result of an analysis must only depend on data in the WPDB whose entity in the LSV is read. The analysis must not add any data to WPDB entities which are not marked as `writes` or `writes−temporary` nor change any data that is not marked as `invalidates` or `maintains`, respectively. The schedule computed by the scheduler is correct if and only if all analyses are correct w.r.t. their specification and if the LSV correctly models the dependency relations in the WPDB.

4 Scheduling Analyses

To calculate an execution schedule for a set of analyses, their ASL specifications are mapped onto a constraint system, which is solved by means of integer programming. Before we discuss the calculation of the schedule, we first show how the logical structure view (LSV) of the program is generated from ASL specifications. For this purpose, each ASL statement is parsed and a new entity is created for each path element that is not yet represented in the LSV. The special entity for *Document* is present by default. Moreover, each entity is directly connected with its parent entity. For example, for the path statement *Document*/CF/Method we generate two entities: one for CF and one for Method; furthermore, the entity for Method is added to the LSV graph as a dependent entity of the CF entity. While constructing the LSV we check that the LSV does not contain cycles.

Once the LSV is generated, we record for each entity which analyses accesses it and how. This information is needed for the generation of the constraint system. We

distinguish the following six sets, whereby $\mathcal{A}$ denotes the set of all (installed) analyses a, and $\mathcal{E}$ denotes the set of all entities e in the LSV:

- **R** denotes for each entity — belonging to the set of currently added documents — the set of analyses that read the entity. An analysis is also added to this set for each entity on the paths of `writes` or `invalidates` statements that are not written or invalidated by the analysis. Recall, only the last entity on a `writes` or `invalidates` path is actually written or invalidated.
- **W** denotes the set of analyses that specify to write or to write-temporary the entity. In case of `writes −temporary`, the corresponding entity is marked as `temporary` and it is checked that all dependent entities are also marked as `temporary`.
- **I** denotes the set of analyses that directly invalidate an entity. An analysis a invalidates an entity e in two cases: (1) a explicitly declares e in an `invalidates` statement, and (2) a `reads` e and directly invalidates some other entity, on which e depends.
- $\mathbf{I^P}$ denotes the set of analyses that implicitly invalidate the entity e. An analysis a implicitly invalidates an entity e, if a neither `reads` nor directly invalidates e, but declares to directly invalidate some other entity, on which e depends. For example, an analysis that explicitly invalidates the `Method` entity and which does not read any of its dependent entities implicitly invalidates `BCode`, `QCode` and all other entities depending on `Method`. Nevertheless, the analysis does not conflict with an analysis that previously explicitly invalidates `BCode`.
- $\mathbf{R^G}$ denotes the set of analyses that specify to `reads −global` the entity; i.e., the analysis requires access to the currently added documents as well as documents that have been processed in an earlier build.
- **M** denotes the set of analyses that maintain the information of the entity.

Based on the LSV, the constraint system is generated to calculate the schedule. The constraint system ensures that every calculated schedule is valid - in fact, the constraint system can be seen as a declarative specification of the semantics of the ASL. A schedule is valid if all requirements of all analyses are met; i.e. the entities the analysis specifies to read were made available in a previous step and are not (yet) invalidated. Further, a dependent entity is available only if the parent is also available. Moreover, every entity is made available at most once and is also explicitly invalidated at most once. The constraints ensure that an analysis that writes an entity is guaranteed to have exclusive access to the entity and race conditions cannot occur. If the constraints have no solution, an error is reported. In the following:

- T_a^S, $a \in \mathcal{A}$ denotes the point in time (execution step) in the schedule S, at which an analysis a is executed. $T_a^S = 0$ means that the analysis a is not scheduled.
- V_e^S, $e \in \mathcal{E}$ denotes the point in time at which e becomes valid. $V_e^S = 0$ means that e will never be available.
- I_e^S, $e \in \mathcal{E}$ denotes the point in time at which e becomes invalid. $V_e^S > 0 \wedge I_e^S = 0$ means that e is available during the next build.

The generated constraints make use of the following definitions: For any entity e the functions w, m, r, r^g, i, i^p return the sets W, M, R, $\mathrm{R^G}$, I, and $\mathrm{I^P}$ respectively. Given an entity e, the predicate $isTemporary(e)$ returns *true* if e is marked as temporary and *false* otherwise.

The range of the variables must be bounded in order to solve the constraint system using integer programming. The domain of the variables T_a^S, V_e^S and I_e^S is $[0, ..., MAX]$ where MAX is $2 * m + n$ ($m = |\mathcal{E}|$ being the number of entities and $n = |\mathcal{A}|$ being the number of installed analyses). MAX defines the theoretical maximum value of the variables T_a^S, V_e^S, and I_e^S. To schedule n analyses that process m entities, we need at most $2*m+n$ time slots. $2*m$, because each entity e is associated with two time slots: V_e^S and I_e^S. This covers the worst-case where all analyses are executed sequentially, all analyses create only one entity, the analyses do not conflict and entities are also invalidated.

The constraints are shown in Fig. 4 and their purpose is explained in the following. The variables V_{Doc}^S and I_{Doc}^S (equation 1) are the variables for the special *Document* entity. The *Document* entity is — by definition — available at the very beginning of the schedule available at the very beginning of the schedule ($V_{\mathrm{Doc}}^S = 1$) and must not be invalidated ($I_{\mathrm{Doc}}^S = 0$).

Implication (2) requires that — except for the document entity which is provided by the framework — every entity that becomes available during the analysis process is actually created by an analysis. The constraint ensures that at least one analysis is scheduled that writes e. The implication (3) ensures that an entity is created at most once. Implication (4) defines that a specific entity e is available in the step immediately following an analysis that writes e and (5) specifies that an entity e is available before an analysis is executed that reads e. Hence, (4) and (5) ensure the correct order between analyses that write and read an entity.

Implication (6) enforces that entities that will be (re-)read or maintained during the next build cycle(s) are not invalidated. Note that we handle an analysis that `maintains` an entity as if the corresponding entity is written and read again by the analysis during

$$V^S_{\text{Doc}} = 1 \wedge I^S_{\text{Doc}} = 0 \tag{1}$$

Availability (validation) of entities:

$$V^S_e > 0 \Rightarrow \sum_{a \in (w(e) \cup m(e))} T^S_a > 0, \quad \text{for each } e \in (\mathcal{E} - \{\text{Doc}\}) \tag{2}$$

$$\forall a \in (w(e) \cup m(e)), T^S_a > 0 \Rightarrow \sum_{x \in (w(e) \cup m(e))} T^S_x = T^S_a, \quad \text{for each } e \in \mathcal{E} \tag{3}$$

$$\forall a \in (w(e) \cup m(e)), T^S_a > 0 \Rightarrow T^S_a + 1 = V^S_e, \quad \text{for each } e \in \mathcal{E} \tag{4}$$

$$\forall a \in (r(e) \cup r^g(e)), T^S_a > 0 \Rightarrow 0 < V^S_e < T^S_a, \quad \text{for each } e \in \mathcal{E} \tag{5}$$

Invalidation of entities:

$$\forall a \in (r^g(e) \cup m(e)), T^S_a > 0 \Rightarrow I^S_e = 0, \quad \text{for each } e \in \mathcal{E} \tag{6}$$

$$\mathit{isTemporary}(e) \Rightarrow V^S_e \leq I^S_e, \quad \text{for each } e \in \mathcal{E} \tag{7}$$

$$I^S_e > 0 \Rightarrow 0 < V^S_e < I^S_e, \quad \text{for each } e \in \mathcal{E} \tag{8}$$

$$\forall a \in i(e), T^S_a > 0 \Rightarrow I^S_e = T^S_a \wedge \sum_{x \in i(e)} T^S_x = T^S_a, \quad \text{for each } e \in \mathcal{E} \tag{9}$$

$$\forall a \in i^p(e), T^S_a > 0 \wedge V^S_e > 0 \Rightarrow 0 < I^S_e < T^S_a, \quad \text{for each } e \in \mathcal{E} \tag{10}$$

$$\forall a \in (r(e) - i(e)), T^S_a > 0 \wedge I^S_e > 0 \Rightarrow T^S_a < I^S_e, \quad \text{for each } e \in \mathcal{E} \tag{11}$$

Objective function:

$$\mathit{minimize} \left(\sum_{a \in \mathcal{A}} T^S_a + \sum_{e \in \mathcal{E}} V^S_e \right) \tag{12}$$

Figure 4. Constraint system for calculating the schedule

the next build; but the write does not have to precede the read operation. The analysis reads and writes the entity in the same step. For an entity that is marked as temporary, constraint (7) ensures, that we are able to determine a point in time at which the entity can become invalid.

Constraint (8) ensures that only entities are invalidated that were created previously. Constraint (9) enforces that the invalidation of an entity happens in the same step as the analysis that explicitly declares the invalidation and that only one analysis explicitly invalidates the entity.

Constraint (10) states the relation between the point in time where an analysis a is executed and the time where entities e are invalidated that are implicitly invalidated by a. If e exists ($V^S_e > 0$), we require that e is invalidated at an earlier point in time than the execution of a. This allows another analysis to explicitly invalidate e before a is scheduled.

Constraint (11) specifies that an analysis is executed before the invalidation of the entities that the analysis declares to read.

The objective function (12) is the minimum of the sum of all analysis times and points in time where the entities become valid. Minimizing the sum of the analyses times is equivalent to finding a schedule that executes only necessary analyses as early as possible. By including the points in time at which an entity becomes available we make sure that only those analyses are scheduled that create the minimum number of entities necessary for satisfying all constraints.

If we directly solve the constraint system in Fig. 4, no analysis is scheduled; the T^S_a values for all analyses will be zero as this minimizes the objective function. To calculate a meaningful schedule, for any user selected analysis a we add the constraint: $T^S_a > 0$

Tab. 2 shows an example schedule that is calculated when the user selects all analyses in Fig. 2, Listing 3 except for the CTAV analysis. The schedule shows which analysis has to be executed in which step and which entity becomes valid, respectively invalid, in a step. In step 1 the *Document* (Doc) entity becomes valid. In step 2 the CFP analysis is executed; as a result, in step 3 the CF, Field (F), Method (M) and, BCode

step	1	2	3	4	5	6	7	8	9	10	11
	V_{Doc}	T_{CFP}	V_{CF} V_{F} V_{M} V_{BC}	T_{BCFG} T_{LIB} T_{EH}	V_{CFG} V_{L} V_{LCF} V_{LF} V_{LM}	T_{NSF} T_{TH}	V_{THC} V_{THF} I_{CFG}	T_{BtoQ} I_{BC}	V_{QC}	T_{NSF} T_{CFT}	I_{THC}

Table 2. Example schedule

(BC) entities are available for all documents added in the current build step. Next, the BCFG analysis can run in parallel with the LIB and the EH analysis. As a result, the CFG for each method of all added class files is available in step 5, as well as information about the used libraries (library L, library class file LCF, public fields in the library LF and public methods of the library LM). Next, the TH and NSF analyses can run in parallel in step 6. The type hierarchy (THF) and the information about type hierarchy changes (THC) is then available in step 7. Further, the CFG entity is invalidated. In step 8, the analysis that transforms the method bodies in the 3-address based representation (BtoQ) is executed which directly invalidates the BCode entity (BC)[1]. When the 3-address based information (QC) is available the CFT analysis and the NSF checker is executed. In the last step (11), the type hierarchy change information (THC) is invalidated, because it is marked as temporary. The values of all other variables, i.e., the variables not shown in the schedule, such as e.g., $T_{\mathrm{CTAV}}, I_{\mathrm{CF}}, I_{\mathrm{F}}$, etc. are zero.

5 Evaluation

The approach has been implemented as the scheduler of Magellan [10] — an open platform for static analyses tightly integrated into Eclipse [9]. The constraint systems is realized using ZIMPL [21] as the mathematical programming language and lp_solve [4] for solving it. The set of analyses used for evaluation includes: analyses listed in Fig. 2, 20 other analyses that check the use of the standard Java API, as well as an incremental inter-procedural call-graph analysis, which is used by some of the checkers.

The evaluation considers two aspects. First, our primary concern is the time needed to complete an incremental build process. That is, to determine how efficient the calculated schedules are. Second, the efficiency of the scheduling process itself is also assessed to determine how well the scheduling approach scales in terms of the number of analyses that it can support.

[1]Direct invalidations happen in the same step to prevent other analysis to still use them.

By construction, the produced schedules, beyond being feasible in terms of inter-analyses dependencies, also minimize the number of analyses performed, as well as the number of data produced. Furthermore, the scheduler is able to determine analyses that can run in parallel because the data they depend on is disjoint. However, the approach does not take into consideration the execution time of analyses: each analysis is assigned one time slot. As a result, it is possible that — on a multiprocessor system — a long running analysis uses one processor while the other processor is idle.

To make better use of multi-processor architectures, Magellan implements data parallelism for all so-called resource-based analyses — analyses that process one document after another and do not need write access to other entities. For example, the NSF checker can be executed for all class files in parallel since it analyzes the finalize methods and does not write any entities. Other candidates are analyses that do not write access the same entity, in particular those that make no use of `maintains`, or `writes-temporary`. For such analyses, multiple instances run in parallel on different partitions of the data set to be analyzed.

For analyses that are not automatically parallelized (e.g. the inter-procedural call graph analysis), Magellan provides additional functionality to help the developer parallelize the execution. Automatic and manual parallelizations are complemented by the strategy that whole program analyses are executed before resource-based analysis in the same time slot.

To evaluate our strategy, we ran 25 checkers on a project with approximately 100K lines of code (Jedit) on single and dual CPU systems. On a single processor system with a P4/3GHz CPU, the time for a full build was 16 seconds. Using a dual processor system with two P4/3GHZ CPUs the time is 11 seconds. The time for an incremental build, when only one class is changed at a time, ranges between 30ms and 62ms on the dual CPU system and between 53ms and 100ms on the single CPU system, depending on the size of the edited class and the type of change.

The evaluation suggests that the scheduler makes good usage of multi-processor architectures; compensating for assigning only one time slot to each analysis.

However, a more detailed evaluation of the choice of the objective function — w.r.t. the minimization of the overall analysis time — is left for future work. A larger number of analyses needs to be available to thoroughly compare different objective functions.

We also briefly consider the second aspect of our evaluation - the performance of the scheduling process expressed in terms of the number of analyses that can be scheduled in a reasonable time. Scheduling a set of more than 60 analyses takes less than 10 seconds on a P4/3GHz and is done only once everytime the user changes the set of analyses that should be executed along with the build process. Using a commercial grade integer programming solver the schedule is even calculated in less than 0.5 seconds. Hence, the calculation of the schedule is not a limiting factor even if we would have several hundred analyses [2].

6 Related Work

Several extensible tools for analyzing software projects have been developed. These tools can be divided in two broad categories. In the first category, there are tools that enable the developer to implement new analyses using declarative query languages. For instance, PQL [23] is a specially developed query language, CodeQuest [18] uses Datalog [27], XIRC [13] uses XQuery [5] and Xgcc [17, 2, 19] uses its own state-machine based language Metal. The second category consists of tools that provide an API for developing analyses, such as IRC [14], FindBugs [20] or PMD [6].

The tools of the first category have in common that the information that is made available about the programs is fixed. Analyses are strictly divided into two categories. (1) Tool internal analyses to build up the information about the project. (2) User defined queries executed in a second step. In PQL [23], for example, the source of information is a context-sensitive, flow-insensitve, inclusion-based pointer alias analysis. However, the analyses that create the program database are executed independently of the needs of the actual queries.

Further, since the set of analyses that make up the program database is fixed, these tools are targeted toward a specific type of analysis. For example, PQL [23] is particularly well-suited for data-flow related analyses. XIRC, on the other hand, was designed to check structural properties of classes. While being very useful for detecting certain types of errors, and being extensible within a particular problem class, these tools cannot be used as platforms for the implementation of a broad range of analyses. A second consequence of always executing a fixed set of base analyses is that, if sophisticated non-incremental analyses, e.g., as in case of PQL, are executed, the time to update the database is too lengthy to enable an integration with the incremental build process. An advantage of these approaches is that conflicts between analyses that are executed in parallel cannot occur; the queries perform read-only access to the program database. Hence, an explicit scheduling of analyses is not necessary.

Tools of the second category, i.e., tools that provide an explicit API for the development of new analyses, also provide a specific representation of the program's code that is to be used for the implementation of the analyses. For example, FindBugs [20] uses the Java bytecode library BCEL [7] as the basis for the representation of the program's code. BCEL provides an object-oriented representation of a Java class file and implements a basic intra-procedural data-flow analysis. IRC [14] uses an approach comparable to Findbugs; PMD [6] uses the abstract syntax tree. Though, it is technically possible that analyses implemented in Java / C++ that operate on an object graph can refine or transform the graph, these operations are not supported by the frameworks. A transformed representation might conflict with other analyses executed thereafter. However, even when a developer decides to extend a framework's representation by implementing a new analysis that additionally provides a higher-level intermediate representation, the execution of analyses that operate on top of the new intermediate representation is not supported. The tools do not provide basic functionality for dependency management of analyses. Handling dependencies between analyses is, however, required to ensure the execution of an analysis that provides additional information before the analyses that want to access the information.

Though the proposed approach can also be used to realize a build management tool, such as Make, this is not in the focus of our work. In case of a static analysis platform the execution of the user selected analyses is the focus. The effect of the analyses on the underlying data is not a concern of the user. In case of a build management tool the user is just interested in getting the result, e.g., the executable, which tasks generated the result is irrelevant. However, when compared with Make our approach provides a more fine grained data model that also enables reasoning about the inner structure of a file. Further, the proposed model also supports the explicit invalidation of entities. In Make every entity is a file and a task must not

[2] Though, the time for analyzing a project rises along with the project's size, the time for calculating the schedule is independent of the size and just depends on the number of installed and selected analyses.

invalidate (delete) files. Make on the other hand automatically determines which tasks need to be executed to create the new result, whereas our platform just calls every analysis in case of a change and, basically, each analysis has to determine the scope of entities that need to be processed on its own.

7 Summary

This paper presented an approach for scheduling an open set of analyses in a static analysis platform. Two forces have driven this work. First, the ability to define analyses independent from each other, which is crucial in an open platform. Second, the ability to run static analyses along with the incremental build process offered by the Eclipse IDE.

To enable the integration of independently developed analyses, we have proposed a specification language for analyses to describe the dependencies among analyses. Given the specifications and a selection of analyses by the end-user, we calculate a schedule using integer programming that (a) derives a minimal set of all analyses necessary to satisfy the requirements of the end-user analyses, and (b) parallelizes scheduled analyses. As the evaluation has shown, the specification language provides sufficient means to specify analyses and the calculated schedules can be efficiently executed.

References

[1] C. Artho and K. Knizhnik. Jlint. `http://artho.com/jlint/`, June 2004.

[2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society.

[3] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of POPL 2002*. ACM Press.

[4] M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve, version 5.5.0.6; open source (mixed-integer) linear programming system. `http://lpsolve.sourceforge.net/5.5/`, 2005.

[5] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. Xquery 1.0: An xml query language. W3c working draft 04 april 2005. `http://www.w3.org/TR/xquery/`.

[6] T. Copeland, et. al. Pmd 3.0. `http://pmd.sourceforge.net/`, March 2005.

[7] M. Dahm, J. van Zyl, and E. Haase. Bytecode engineering library. `http://jakarta.apache.org/bcel/`, 2006.

[8] L. G. DeMichiel. *Enterprise JavaBeans Specification, Version 2.1*. SUN Microsystems, 2003.

[9] Eclipse 3.1. `http://www.eclipse.org`, 2005.

[10] M. Eichberg and C. Bockisch. Magellan, http://www.st.informatik.tu-darmstadt.de/magellan, 2005.

[11] M. Eichberg and C. Bockisch. The bytecode analysis toolkit (BAT). `http://www.st.informatik.tu-darmstadt.de/BAT`, 2006.

[12] M. Eichberg, S. Kanthak, S. Kloppenburg, M. Mezini, and T. Schuh. Incremental confined types analysis. In *Proceedings of LDTA 2006*. Elsevier.

[13] M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer. Xirc: A kernel for cross-artifact information engineering in software development environments. In *Proceedings of WCRE 20004*. IEEE Computer Society.

[14] M. Eichberg, M. Mezini, T. Schäfer, C. Beringer, and K.-M. Hamel. Enforcing system-wide properties. In *Proceedings of ASWEC 2004*. IEEE Computer Society.

[15] M. Eichberg, T. Schäfer, and M. Mezini. Using annotations to check structural properties of classes. In *Proceedings of FASE 2005*. Springer.

[16] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of SOSP 2003*. ACM Press.

[17] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of OSDI 2000*. Usenix.

[18] E. Hajiyev, M. Verbaere, O. de Moor, and K. de Volder. Codequest: querying source code with datalog. In *Companion to the Proceedings of OOPSLA 2005*. ACM Press.

[19] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of PLDI 2002*. ACM Press.

[20] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12), 2004.

[21] T. Koch. *Rapid Mathematical Programming*. PhD thesis, Technische Universität Berlin, 2004.

[22] V. B. Livshits. Findings security errors in java applications using lightweight static analysis. In ACSAC, Work-in-Progress Report, November 2004.

[23] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *Proceedings of OOPSLA 2005*. ACM Press.

[24] Microsoft. Prefast with driver-specific rules. `http://www.microsoft.com/whdc/devtools/tools/PREfast-drv.mspx`, October 2004.

[25] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. D. Johnson, A. Kershenbaum, and L. Koved. Saber: smart analysis based error reduction. *SIGSOFT Software Engineering Notes*, 29(4), 2004.

[26] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, J. Dolby, A. Kershenbaum, and L. Koved. Validating structural properties of nested objects. In *Companion to the Proceedings of OOPSLA 2004*. ACM Press.

[27] J. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.