Automatic Inference of Abstract Type Behavior

Mihai Balint [*]
LOOSE Research Group
Department of Computer and Software Engineering
Politehnica University of Timişoara, Romania
mihai@cs.upt.ro

ABSTRACT

Type hierarchies are an integral part of the object oriented software reuse machinery. Software flexibility can be increased through type inheritance which, if used in accordance with Liskov Substitution Principle (LSP) enables safe object substitution.

Assuming that formal specifications are available for a set of subtypes, we present our early doctoral research on the automatic inference of an extended deterministic finite automaton that describes the legal usage of abstract supertypes and ensures the behavioral subtyping relation as defined by the Liskov Substitution Principle (LSP). We obtain the supertype interface automata by incrementally exploring the specification of the subtypes, unifying correlated subtype fields, simplifying predicates through quantification, and finally creating new model fields that we associate with the remaining predicates.

The inferred automaton is simulated by the behavior of each subtype and can be used for safe hierarchy extension, verification of new hierarchy clients, and emphasis of LSP non-compliant methods.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specifications; D.2.4 [**Software Engineering**]: Software / Program Verification

General Terms

Verification, Reliability, Documentation

[*]This work was partially supported by the Romanian National Science Foundation (CNCSIS) under project "Methods and Tools for Continuous Quality Assurance in Complex Software Systems" (PNII-IDEI 357/1.10.2007) and by the strategic grant POSDRU 6/1.5/S/13, (2008) of the Ministry of Labour, Family and Social Protection, Romania, co-financed by the European Social Fund - Investing in People.

Keywords

automated specification mining, type specification inference, type inheritance, object oriented software

1. INTRODUCTION

One of the promises of the object-oriented programming paradigm is that of easier extension through encapsulation and type inheritance. However extending a hierarchy while only structurally matching the interface is unsafe and has determined the development of behavioral subtyping [14] which enables safe object substitution. Given the existence of formal behavioral specifications for both subtype and supertype, the behavioral subtyping relation can also be automatically verified.

Writing formal specifications manually is however a laborious process that in the presence of inheritance and polymorphic calls becomes even more difficult. An early paper by Huisman, on the verification of the `AbstractCollection` class from the standard Java API, shows that manual specification is error prone and requires careful refinement [13]. The usefulness of specification for checking reliability, safety, and other properties of software cannot be denied and the automatic generation of specifications is highly sought after using different techniques [2, 10, 19, 20, 21]. Existing approaches typically rely either on static program analysis or on dynamic execution analysis both of which have precision problems. Static analysis based methods are limited due to undecidability and overestimate unlikely paths which cause high rate of false specifications. Dynamic analysis methods are limited by the number and variety of available executions traces and under-represent the exceptional situations that are usually critical to program safety.

In this paper we present our early research for a new specification inference method that relies on existing specifications for the concrete types from an inheritance hierarchy and which is (by construction) at least as precise as the existing specifications. Our approach only works for abstract supertypes, a category that is under-represented by current approaches.

Assuming that the subtypes from a hierarchy have formal first order logic specifications in the form of class invariants and method pre- and post-conditions we (1) construct for the supertype an extended interface automaton from the synchronous exploration of the subtype specifications; (2) identify correlations between subtype fields; (3) unify correlated fields; (4) refine the supertype specification by inserting model fields for the remaining variables or predicates (if

$$\exists v.s_0(v) \Rightarrow \bigwedge_{i/is\cdot a\cdot subtype} (Pre_{m_i}(v) \wedge Inv_i) \quad (1)$$

$$\forall v, v'.s_0(v) \wedge \bigwedge_{i/is\cdot a\cdot subtype} (Inv_i(v) \wedge Pre_{m_i}(v) \wedge Post_{m_i}(v, v')) \Rightarrow s_1(v')) \quad (2)$$

predicate abstraction is available); (5) use approximation to remove any other subtype-specific fields.

The main contribution of our approach consists of the mechanized generation of an extended interface automaton that describes the correct usage of abstract supertypes. The inferred automaton is constructed in such a way as to define that supertype specification which ensures that the subtypes are true behavior subtypes as defined by Liskov and Wing in [14]. For hierarchies that are not based on the behavioral subtyping relation this restriction causes the inferred automaton to contain only transitions that correspond to the supertype methods which are LSP compliant. If no methods can used uniformly then the trivial single state automaton is generated. Using a program verifier [3, 9], the inferred specification can be used to check the safety properties of new hierarchy subtypes.

The interface automaton inferred using our approach complements existing approaches that generate specifications only for concrete types and ignore or are less precise for abstract types.

In the next sections we describe our approach, its assumptions, the construction procedure and possible refinements. An example is given and the paper concludes with the related work and current and future plans.

2. SUPERTYPE BEHAVIOUR INFERENCE

This section presents the proposed method for inference of supertype interface automata. Before diving into the actual inference procedure we present the assumptions on which our approach relies and we define the extended interface automaton.

2.1 Assumptions

Behavioral subtyping assumptions We assume that that the type hierarchy is constructed using the behavioral subtyping relation as presented in [14]. This is a key property during the automaton construction procedure (Section 2.3) as Liskov's behavioral subtyping rules are used (1) to check whether a method can be invoked from a given state and (2) to determine the relation that describes the destination state of a transition.

Subtype specification assumption We assume that the derived types have formal specifications in the form of type invariants, constraints and method pre- and postconditions. While in a manual specification process it is unlikely that only subtypes will have specifications, our approach is useful when used in conjunction with other tools or techniques that infer specification only for concrete types (i.e.,`Daikon` [8]). Also in a maintenance context, if a new supertype must be extracted from several types then our approach can be applied to also infer a specification for the new abstract type.

Based on the formal specification we construct an interface automaton starting from the initial state and incrementally exploring the all states and transitions as implied by the invariants, pre- and postconditions of each method.

2.2 Extended Interface Automaton Definition

In this section we formally describe our approach and incrementally apply it on an example, to build the behavioral interface for the `Engine` type from Figure 1.

Similar to [1, 12], we have chosen to represent the inferred behavior using interface automata which is automata-based formalism that captures input assumptions about the order in which the methods of an component may be called. We extend interface automata by labeling transitions with the method name and restrictions on method parameters. Formally an behavioral interface is a deterministic automaton $A = (\Sigma, S, s_0, \delta, F)$ where S is the set of all states, $F \subseteq S$ is the set of final states, $s_0 \in S$ is the initial state, Σ is the alphabet accepted by the automaton and $\delta : S \times \Sigma \to S$ is the transition function.

To formalize the input alphabet accepted by our extended automaton, let us consider an initial program state $s \in S$ and a final state $s' \in S$ both defined using relations over V, the program variables. Also, let v and v' be sets of variables values denoting legal values in states s and s' respectively and $Pre_m(p, v)$, $Post_m(p, v, v')$ the pre- and postcondition of method $m(p)$ (with p the set of parameters) which causes the automaton to change states from S to S'. Using these definitions, a transition is defined by S and S' the source and destination states, the input symbol $m(p)$ (the method name and parameters) and can be annotated with pre- and postconditions ($Pre_m(p, v)$ and $Post_m(p, v, v')$ respectively) expressed as relations between state variables and parameters.

2.3 Interface Automata Construction

The extended interface automaton is constructed as the product automaton of the subtype specifications. We start with the initial state of the automaton, which we describe as the disjunction of subtype constructor postconditions. We write s_0, the formula for the initial state of the interface automaton, as $s_0 = \vee(s_{0_i})$ where s_{0_i} itself is a disjunction of the postconditions from each of subtype i's constructors.

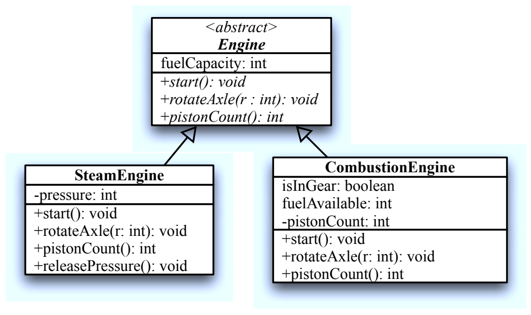

Figure 1: The *Engine* type hierarchy. We assume that *SteamEngine* and *CombustionEngine* are behavioral subtypes of *Engine*.

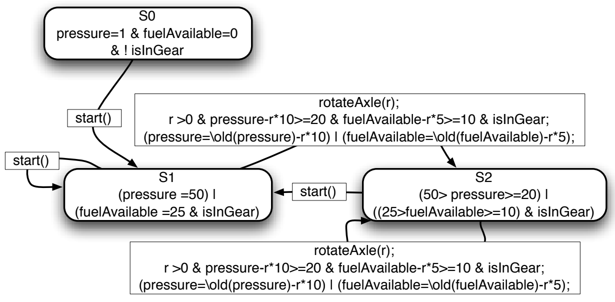
S0
pressure=1 & fuelAvailable=0
& ! isInGear
start()
start()
rotateAxle(r);
r >0 & pressure-r*10>=20 & fuelAvailable-r*5>=10 & isInGear;
(pressure=\old(pressure)-r*10) | (fuelAvailable=\old(fuelAvailable)-r*5);
S1
(pressure =50) |
(fuelAvailable =25 & isInGear)
start()
S2
(50> pressure>=20) |
((25>fuelAvailable>=10) & isInGear)
rotateAxle(r);
r >0 & pressure-r*10>=20 & fuelAvailable-r*5>=10 & isInGear;
(pressure=\old(pressure)-r*10) | (fuelAvailable=\old(fuelAvailable)-r*5);

Figure 2: The interface automaton obtained after the synchronous exploration of the specifications of `SteamEngine` and `CombustionEngine`

Table 1: *CombustionEngine* specification.

Invariants	$fuelCapacity > 0$ $fuelAvail \leq fuelCapacity$ $fuelAvail \geq 0$ $pistonCount > 0$
Method Signature	Preconditions and postconditions
start(): void	
ensures	$isInGear \wedge fuelAvail = 25$
rotateAxle(r:int):void	
requires	$r > 0 \wedge fuelAvail - r \cdot 5 \geq 10 \wedge isInGear$
ensures	$fuelAvail = \backslash old(fuelAvail) - r \cdot 5$
pistonCount(): int	
ensures	$\backslash result = pistonCount$

Next we attempt to identify which supertype methods can be invoked from the initial state by using the preconditions from the subtypes. A method m can be invoked from state s_0 if Equation 1 is satisfiable, given that m_i is the implementation of m in subtype i, Pre_{m_i} is the precondition of m_i, Inv_i is the class invariant for subtype i and V is the set of all variables from all subtypes and v is a valuation for all variables from V.

If a method satisfies Equation 1 then a transition from s_0 to a new state s_1 is possible. Following the subtyping rules of the LSP [14], the new state is described by Equation 2, where $Post_{m_i}$ is the postcondition of m_i and v' is a valuation for all variables in state s_1. Checking and simplifying these equations can be done by employing recent SMT solvers such as [6, 18].

Ideally this iterative process continues until no new transitions can be added to the automaton. However, in practice the iteration will likely have to be limited using over-approximation. For example if it is determined that a new state implies an existing state, it has to be decided whether the new state will be actually added or the transition will be redirected to the existing state.

Figure 1 presents a UML class diagram of a contrived `Engine` hierarchy that we will use to exemplify our approach. The formal specification for each type is provided in tables 2 and 1. We assume that after construction the member fields are initialized with the following values: `Engine`: $\{fuelCapacity := 200\}$, `SteamEngine` : $\{pressure := 1\}$ and `CombustionEngine` : $\{fuelAvail := 0 \wedge isInGear := false\}$.

For this example, the extended interface automaton can be obtained using the construction procedure informally described above. From the initial state $S0$ only *start()* can be called since this state does not satisfy the precondition

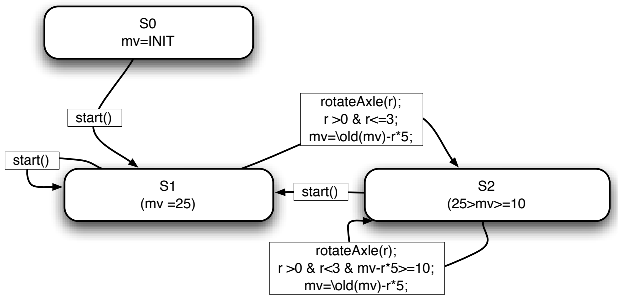

Figure 3: The interface automaton after pruning most of the subtype-specific references

of *rotateAxle(r:int)* . $S1$, the destination state for *start()* is described by the disjunction of the subtype postconditions. From $S1$ calling *start()* simply loops back, however $S1$ satisfies the precondition of *rotateAxle(r:int)* which causes a transition into $S2$. The equations for $S2$ are obtained using Equation 2, which introduces the starting state equations and the precondition within the postcondition and yields the equations for the destination state. The result of this process is presented in Figure 2. To avoid figure clutter, we have omitted the postcondition for the *start()* method since it is identical to $S1$. We have also omitted the invariant $fuelCapacity > 0$ and the transitions associated with *pistonCount()* since it has no side-effects and no preconditions, it can be invoked from any state and would only loop back to that state.

2.4 Abstraction of subtype-specific fields

A problem with the resulting automaton is that it is labeled with lots of subtype member fields that have no meaning in the context of the supertype. This can be alleviated if we simplify the preconditions of each transition using the constraints that define the source and destination states.

Model Field Insertion A model field, as defined for JML in [7], is an abstract type member that is visible only within the specification. It is defined in the abstract supertype and in subtypes it is given a representation in the form of a formula based on other subtype-specific state variables.

We use model fields to replace references to subtype-specific information, either actual fields or more usually information common to all subtypes. In the example from Figure 2, *mv* is the model field that replaces references to *pressure*, *fuelAvail* and *isInGear*.

Table 2: *SteamEngine* specification.

Invariants	$fuelCapacity > 0$ $pressure > 0$
Method Signature	Preconditions and postconditions
start(): void	
ensures	$pressure = 50$
rotateAxle(r: int): void	
requires	$r > 0 \wedge pressure - r \cdot 10 \geq 20$
ensures	$pressure = \backslash old(pressure) - r \cdot 10$
pistonCount(): int	
ensures	$\backslash result = 2$
releasePressure(): void	
requires	$pressure > 20$
ensures	$pressure = 20$

Extensive use of model fields can cause the supertype specification to become overly complicated which in turn results in difficulties when the hierarchy needs to be extended with new subtypes.

Model fields can also be used together with predicate abstraction [11] to further remove specific references from the supertype interface automaton. To limit the proliferation of model fields it is useful to first identify some relations between predicates (similar to field unification but not only across subtypes also within single subtypes). Implementing subtype representations for the predicate abstraction model fields is trivial as the Boolean field takes the predicate expression as representation in the subtypes from which the predicate originated and is undefined in others.

Unification The first step is to attempt to identify correlations between fields from different subtypes. Such correlations function as an invariant for the supertype and may reduce dependency on subtype-specific fields. If there exist a set of fields, one from each subtype and a relation between them then we can unify the set of fields.

The natural representation for the unification is a model field so updating the supertype specification with one for each set of unified fields is straightforward. The model field must also be given a representation within each subtype. The representations are deduced from the equations obtained from the unification process.

In the example from Figure 2 in $S1$ and $S2$ the following correlation exists between *pressure*, *fuelAvail* and *isInGear*: $\mathit{inGear} == \mathit{true} \wedge \mathit{pressure} = 2 \cdot \mathit{fuelAvail}$. Since the correlation does not hold in $S0$ we refine it and obtain Equations 3-5 as the correlations of *pressure*, *fuelAvail*, and *isInGear*. (assuming that the supertype variable that holds the current state is named ST). The model field used for unification mv is defined in $S0$ as $mv = INIT$, in $S1$ and $S2$ as $mv = fuelAvail \wedge mv = pressure/2$, where $INIT$ is some arbitrary value and ST is the supertype model field that holds the current state. The representation of mv in the subtypes can be obtained from Equations (3-5).

$$(ST \neq S0 \wedge \mathit{isInGear}) \wedge (ST = S0 \Rightarrow \neg \mathit{isInGear}) \quad (3)$$
$$(ST \neq S0 \Rightarrow \mathit{pressure} = 2 \cdot mv) \wedge (ST = S0 \Rightarrow \mathit{pressure} = 1) \quad (4)$$
$$(ST \neq S0 \Rightarrow \mathit{fuelAvail} = mv) \wedge (ST = S0 \Rightarrow \mathit{fuelAvail} = 0) \quad (5)$$

Using Equations 3-5 we can replace *pressure*, *fuelAvail* and *isInGear* from all predicate used in the automaton and simplify all formulas.

Transition Constraints Simplification At this stage, the transition pre- and postconditions are fully specified using the relations from the subtypes and any unified variables we may have obtained at the previous step. We can simplify them by introducing the relations from the source state into the relations for the pre- and postconditions. Ideally this step will remove references to subtype fields and leave only parameter related constraints. This simplification may cause different transitions labeled with the same method to have different pre- and postcondition annotations, however each of these is more precise (and likely more compact) than the complete method constraints. If needed, the complete precondition can be constructed from the disjunction of all transition preconditions and source state specifications.

In Figure 3 the *rotateAxle(r:int)* transition from $S1$ to $S2$ has been simplified by using $S1$'s defining relation with the pre- and postconditions of *rotateAxle(r:int)* . The same simplification has been applied to the *rotateAxle(r:int)* transition from $S2$ to $S2$ and as expected the constraints obtained differ from the other *rotateAxle(r:int)* transition that starts in $S1$.

Simplification through quantification Let $Q(v,p)$ be a predicate on state variables and method parameters, then depending on the desired goal we may define $Q^A_p(v)$, $Q^A_v(p)$ as two abstractions of $Q(v,p)$, the first abstracting method parameters (suited for abstracting state relations) and the second abstracting state variables (best suited for transition constraints). One way to obtain such abstractions is through quantification over p: $\exists p \,.\, Q^A_p(v) \Rightarrow Q(v,p)$ and over v: $\exists v \,.\, Q^A_v(p) \Rightarrow Q(v,p)$. Solving the existential quantification is a frequent problem in model-checking and recent tools can handle it very well.

The advantage of this approach is that it simplifies constraints without introducing any artifacts within the abstract supertype specification. The disadvantage is that it could yield an approximative specification and as such may not be useful for a particularly restrictive use-case.

Approximation and predicate abstraction may be leveraged to generate a behavioral interface either very precise or easier to implement in a new subtype. Using coarse approximations will result in an automaton that is simple, contains fewer restrictions but can only be used to prove a limited set of safety properties. Using model fields with predicate abstraction extensively will yield an interface that is more precise, can be used to prove a larger set of safety properties but is more complex and may be more difficult to understand and use for hierarchy extension.

3. RELATED WORK

Program specification mining has been approached from several angles in the literature. Among the first to generate a temporal interface for objects, Whaley *et al.* [21] use static analysis to find all pairs of methods that called in order throw an exception. These facts are augmented with information obtained from a dynamic analysis to create FSMs that model the behavior of Java classes, however for interfaces and abstract types their approach is reduced to a dynamic analysis. Several other static analysis based approaches have been proposed in [2, 10, 4]. These rely on the analysis of the concrete class source code and are not directly usable for the inference of specifications for abstract types.

Recently, researchers have applied static analysis to the client source code hoping to infer usage patterns of components [19, 16]. This approach can also generate specifications for abstract types but is limited by the specific usages that the clients implement.

Another category of approaches is based on the analysis of program execution traces. In [15], Lorenzoli *et al.* uses dynamic analysis to infer extended finite state machines (EFSM) that represent the behavioral interfaces of component. The `Daikon` tool presented in [8] also uses execution traces to infer likely program invariants. Since these tools analyze program execution traces, they only observe the usage of concrete object types and cannot directly infer the specification of abstract types.

Our construction procedure is similar to the witness inference algorithm from [12] if we consider Liskov's behavioral subtyping constraints [14] as a safety property. Their approach is incomplete for our case since it produces a supertype specification that also references subtype specific fields and methods.

The interface automata construction procedure presented in this paper is similar to the state exploration of bounded model checking [5]. However the purpose is different, model checking explores states trying to find property violations, we on the other hand construct a specification.

Regarding the analysis of type hierarchies, the work of Mihancea [17] is related in the sense that he is able to identify inconsistent subtype usage by analyzing hierarchy usages. However he's results are focused towards reverse-engineering and program understanding and cannot be used for automatic verification.

4. ONGOING WORK AND CONCLUSIONS

Evaluation. A prototype implementation that uses JML specifications for the Java programming language is under development. We plan to evaluate the quality of our results using the subtype specifications from ESC/Java2's jmlspecs project and comparing our results with the jmlspecs supertype specifications.

To evaluate the effectiveness of our approach together with existing specification inference tools, we will use the subtype specifications obtained using tools such as `Daikon` [8] and `SAFE` [19] and again compare our results with the supertype specifications from the jmlspecs project.

To evaluate the usefulness of our results we will use a program verifier to find specification violations in several large software applications. We will then compare the results obtained using specification inferred using our approach with the results obtained using the specifications from the jmlspecs project.

Future Work. Further work will involve measuring the effectiveness of our results in advancing object oriented software engineering and maintenance by providing an automated behavioral safety net to developers that have to use or extend type hierarchies. If integrated with and IDE we could identify related types that should have a common base class and methods that are safe to "pull-up" in the base class. Adding a static checker would provide a maintainer with information about mis-behaving methods from subtypes or mis-behaving hierarchy clients. Our approach complements existing techniques that cannot obtain specification for abstract classes.

Conclusions. In this paper we have presented a novel method for the inference of behavioral interfaces for abstract supertypes. By construction the resulting interface automaton ensures the behavioral subtyping relation between supertypes and subtypes, a property that enables the emphasis of interface methods that adhere to the LSP which can be used uniformly across subtypes. This is a highly desirable feature that opens several new alleys for software engineering and maintenance research.

Acknowledgments. We gratefully acknowledge Marius Minea for his extensive feedback. We also acknowledge Petru Mihacea, Daniel Ratiu and Casandra Holotescu for their suggestions and editorial advice. Finally we thank the anonymous reviewers for their useful comments on our research scope and topic in general and on this proposal in particular.

5. REFERENCES

[1] L. d. Alfaro and T. A. Henzinger. Interface automata. In *9th ACM SIGSOFT ESEC/FSE-9*, 2001.

[2] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. *SIGPLAN Not.*, 40(1):98–109, 2005.

[3] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[4] D. Beyer, T. A. Henzinger, and V. Singh. Algorithms for interface synthesis. *LNCS 4590*, pages 4–19, 2007.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *5th TACAS '99*, pages 193–207.

[6] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. In *Proc. of IJCAR*, 2010.

[7] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract: Research articles. *Softw. Pract. Exper.*, 35(6):583–599, 2005.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *TSE*, 27(2):99–123, Feb. 2001.

[9] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI 2002*, 234–245, 2002.

[10] C. Goues and W. Weimer. Specification mining with few false positives. In *15th TACAS '09*, pages 292–306.

[11] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *9th CAV '97*, pages 72–83, 1997.

[12] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *ESEC/FSE-13*, 2005.

[13] M. Huisman. Verification of Java's AbstractCollection class: A case study. In *MPC '02: Proc. of the 6th Intl. Conf. on Math. of Progr. Constr.* Springer, 2002.

[14] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[15] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *30th Intl. Conf. on Sw. Eng (ICSE)*. IEEE, 2008.

[16] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro. SEIM: static extraction of interaction models. In *Proc. of the 2nd Intl. Works.on Princ.of Eng. Serv.Orie.Sys. (PESOS), coloc. with 32nd ICSE.* ACM/IEEE, 2010.

[17] P. F. Mihancea. Type highlighting: A client-driven visual approach for class hierarchies reengineering. *Works. on Src. Code Ana. and Manipulation*, 2008.

[18] D. Monniaux. A quantifier elimination algorithm for linear real arithmetic. In *Proc. of the 15th Intl.Conf. on Logic for Program., Artif.Intell. and Reas.*, 2008.

[19] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA '07.* ACM, 2007.

[20] W. Weimer and N. Mishra. Privately finding specifications. *IEEE TSE.*, 34(1):21–32, 2008.

[21] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4):218–228, 2002.