Certificate translation for specification-preserving advices

Gilles Barthe César Kunz

INRIA Sophia-Antipolis Méditerranée

Abstract

Aspect Oriented Programming (AOP) is a paradigm with significant potential to separate functionality and cross-cutting concerns. In particular, AOP supports an incremental development process, in which the expected functionality is provided by a baseline program, that is successively refined, possibly by third parties, with aspects that improve non-functional concerns, such as efficiency and security. Therefore, AOP is a natural enabler for Proof Carrying Code (PCC) scenarios that involve, in addition to the code producer and the code consumer, untrusted intermediaries that modify the code.

The purpose of this article is to explore a PCC architecture that accommodates such an incremental development process. In order to support a wide range of policies, we extend our earlier work on certificate translation, and show in the context of a very simple language that it is possible to generate certificates of executable code from proofs of aspect-oriented programs. To achieve this goal, we introduce a notion of specification-preserving advice, which provides a mild generalization of the notion of harmless advice by Dantas and Walker, and provide a sound verification method for programs with specification-preserving advices.

1. Introduction

While reliability and security of executable code is an important concern in mobile computing scenarios, many program verification techniques and tools target high-level languages, and thus do not address directly the concerns of the code consumers, who require automatic and efficient verification procedures that can be run locally on executable code and that dispense them from trusting code producers (that are potentially malicious), networks (that may be controlled by an attacker), and compilers (that may be buggy).

Proof Carrying Code (PCC) [25, 23, 24] provides a security architecture where executable code is formally verified. In a typical PCC architecture, programs are compiled with a certifying compiler that returns, in addition to executable code, program annotations, which specify program invariants tailored to the desired policy, and a self-explanatory and independently checkable proof, known as certificate, that the code is indeed compliant to the policy. Typically, a certifying compiler will generate both program annotations, as well as proof objects, a.k.a. certificates, that the program is correct. Through its associated verification mechanisms for executable code, PCC suitably addresses the security concerns for mobile code. Nevertheless, current instances of certifying compilers mostly focus on basic safety policies and do not take advantage of the existing methods for verifying source code. Overcoming these limitations is a central theme of the Mobius project [4], and crucial to provide a PCC architecture that accomodates a larger class of security policies, and a larger class of programming idioms.

Earlier work [8, 6, 9, 10, 22] has considered expressive verification methods for executable code and established their adequacy with respect to verification methods for source programs, so as to be able to transfer evidence from source programs to executable code. In particular, Burdy and Pavlova [9] and Charles *et al* [10] have developed a proof compiler for Java, that enables certificates of Java bytecode programs to be constructed from source code verification with JML-based tools such as ESC/Java and Jack. More foundational work on certificate translation [8, 5, 7] has focused on building certificates for executable code from correctness proof of the corresponding source code, in settings where compilation performs aggressive optimizations.

Proof compilation is an important step towards supporting expressive policies since proof compilers allow certificate generation to rely on widely used program verification environments, and thus enables to realistically address (at the cost of interactive verification) expressive policies. Nevertheless, proof compilation currently targets Java programs and does not provide support to generate certificates for programs that have been developed using advanced programming idioms such as aspects.

Contributions The main contribution of this work is to study proof compilation for Aspect Oriented Programming (AOP), and to show in the context of a very simple language that it is possible to generate certificates of executable code from proofs at source level. Being designed to isolate cross-cutting aspects of the software from its main functionality, AOP is a natural enabler for PCC scenarios that involve, besides the code consumer and the code producer, several untrusted intermediaries that enhance the mobile code with specific added value, e.g. related to security and efficiency [21]. Thus our work suggests the feasibility of developing a PCC architecture where untrusted intermediaries modify code using aspects.

In order to realize proof compilation for AOP, we introduce the notion of specification-preserving advice. Informally, an advice a is specification-preserving for an annotated piece of code $\{\Phi\}c\{\Psi\}$, where Φ and Ψ respectively denote the pre- and post-condition for c, if the advised code $a \bowtie c$ satisfies the same specification, i.e. $\{\Phi\}a \bowtie c\{\Psi\}$. Specification-preserving advices are natural in the context of PCC with intermediaries, since many aspects related to security (resource management, logging, *etc.*) and efficiency (e.g. cached functions, optimized code,*etc.*) fall in this category. Moreover, specicification-preserving advices support "separate verification" (as coined by [19]) and allow intermediaries to treat correctness proofs of the baseline code as black-boxes. Concretely, intermediaries will only be required to prove that advices are specification-preserving w.r.t. the code they advise, and an appropriate certificate translator will produce certificates of the weaved code.

In summary, the main technical contributions of this article are:

- the definition of the class of specification-preserving advices that find many uses both for security and efficiency, and that support modular reasoning;
- the relationship between specification-preserving advices and harmless advices [14], which are required to verify the stronger property of preserving the semantics of advised code, except for the possibility of modifying the termination behavior. Inspired by this relationship, we provide a simple static analysis that ensures that advices are specification-preserving;
- the first study of certificate translation in the context of AOP, and the definition of a certificate translator that takes as input an AOP program p and a certificate c of its correctness, and returns a certificate for the compiled program $[\![p]\!]$;
- a mild generalization of the classification of specification-preserving advices to sequences of advices.

2. A basic motivating example

Consider the program p with a procedure main and another procedure twice advised unconditionally by a:

$$\begin{array}{rcl} \mathsf{main}(x) & = & y := \mathsf{twice}(x); z := y + x;\ \mathsf{return}\ z \\ \mathsf{twice}(x) & = & \mathsf{return}\ (x + x) \\ a(x) & = & x := 0; z := \mathsf{proceed}(x);\ \mathsf{return}\ z \end{array}$$

The correctness of the program is established w.r.t. a specification table Γ that associates to each procedure a triple consisting of a precondition, a post-condition, and a modifies clause that states which variables are modified. We choose the obvious specifications for main and twice, i.e.

$$\begin{array}{rcl} \Gamma(\mathsf{main}) & = & (\mathsf{true}, \mathsf{res} = x^\star + x^\star + x^\star, \emptyset) \\ \Gamma(\mathsf{twice}) & = & (\mathsf{true}, \mathsf{res} = x^\star + x^\star, \emptyset) \end{array}$$

(We consider that the variables y and z are local variables, and thus are not declared in the modified clauses).

One can generate for each procedure a verification condition that guarantees, in a traditional setting, that the procedure meets its specification. Both verification conditions hold obviously. Nevertheless all terminating executions of the program will simply return the value given as input, and thus the post-condition will not be satisfied if main is called with an input distinct from 0. In this case, the problem is caused by the fact that a forces twice to be executed with input 0. In other words, a is not parameter-preserving, i.e. causes f to be called with an input different from the one that is declared in the program.

A similar problem shall occur if an advice modifies a global variable that is otherwise unmodified by the procedures it advises. More generally, advices should, in addition to be parameter-preserving, preserve specifications. Consider the modified advice:

$$\begin{array}{rcl} a(x) & = & (\mathtt{if}\ x = 0\ \mathtt{then}\ z := \mathsf{proceed}(x)\ \mathtt{else}\ z := 0); \\ & & \mathsf{return}\ z \end{array}$$

As in the previous case, the post-condition will not be satisfied if main is called with an input distinct from 0. The problem is caused by the fact that a is not specification-preserving. Indeed, consider the function $\hat{a}$ derived from a by replacing the proceed statement by a call to f:

$$\begin{array}{rcl} \hat{a}(x) & = & (\mathtt{if}\ x = 0\ \mathtt{then}\ z := \mathsf{twice}(x)\ \mathtt{else}\ z := 0); \\ & & \mathsf{return}\ z \end{array}$$

One cannot prove that the procedure $\hat{a}$ satisfies the specification of twice, since the proof obligation for $\hat{a}$ with the same pre- and post-condition as twice is logically equivalent to

$$x = 0 \Rightarrow x + x = x + x \land x \neq 0 \Rightarrow 0 = x + x$$

which does not hold.

$$\begin{array}{rrcl} \textbf{Commands} & c & ::= & v{:=}f(e) \\ & & | & v{:=}\ \mathtt{proceed}(e) \\ & & | & v{:=}e \\ & & | & c; c \\ & & | & \mathtt{if}\ b\ \mathtt{then}\ c\ \mathtt{else}\ c \\ & & | & \mathtt{while}\ b\ \mathtt{do}\ c \\ & & | & \mathtt{skip} \\ & & | & \mathtt{return}\ e \\ \textbf{Procedures} & proc & ::= & f\ arg^*\ c_b \\ \textbf{Point-cut descriptors} & ptd & ::= & \mathsf{if}\ b\ \mathsf{around}\ f \\ \textbf{Advices} & advice & ::= & ptd^+\ a\ arg^*\ c_a \\ \textbf{Programs} & Prog & ::= & proc^*\ advice^* \end{array}$$

Figure 1. SYNTAX OF SAL PROGRAMS

Now consider instead the correct advice a:

$$\begin{array}{rcl} a(x) & = & (\mathtt{if}\ x \neq 0\ \mathtt{then}\ z := \mathsf{proceed}(x)\ \mathtt{else}\ z := 0); \\ & & \mathsf{return}\ z \end{array}$$

The function $\hat{a}$ derived from a by replacing the proceed statement by a call to f:

$$\begin{array}{rcl} \hat{a}(x) & = & (\mathtt{if}\ x \neq 0\ \mathtt{then}\ z := \mathsf{twice}(x)\ \mathtt{else}\ z := 0); \\ & & \mathsf{return}\ z \end{array}$$

is specification-preserving, since the proof obligation for $\hat{a}$ with with the same pre- and post-condition as twice is logically equivalent to

$$x \neq 0 \Rightarrow x + x = x + x \land x = 0 \Rightarrow 0 = x + x$$

and it is thus valid. Note that the verification condition generation for $\hat{a}$ relies on the specification of twice, but not on its code.

3. A simple AOP language

This section introduces SAL, a simple procedural language with aspects. For simplicity, SAL is restricted to around advices, to point-cuts at procedure calls, and to point-cut descriptors that do not refer to the control-flow graph.

3.1 Syntax

The syntax of commands can be found in Figure 1, where v ranges over the sets $\mathcal{V}$ of local variables and $\mathcal{X}$ of global variables, arg ranges over local variables, f ranges over the set $\mathcal{F}$ of procedure names, and a ranges over the set $\mathcal{A}$ of advice names. A baseline command is a command that does not contain any proceed command. We let c_b and c_a range respectively over baseline and advice commands.

Point-cut descriptors are of the form if b around f, where b is a boolean condition and f is a procedure name. Then, each procedure is composed of an identifier, its formal parameters and a command that represents its body. Each advice is composed of an identifier from a set $\mathcal{A}$ of advice names, a non-empty set of point-cut descriptors, its formal parameters, and an extended command that represents its body. A program is a given by a set of procedures with a distinguished main procedure and a set of advices.

3.2 Semantics

Advice weaving, which enables aspects to influence the execution of programs at designated program points and under certain conditions, is the fundamental mechanism that determines the semantics of AOP programs. Thus, the essence of SAL programs is captured by the transition rules for the commands call and proceed, which are described informally below. For simplicity, we restrict our attention to procedures and advices with a single formal parameter.

$$\begin{array}{llll}
\textbf{Logical expressions} & \bar{e} & ::= & \mathsf{res} \mid x^\star \mid x \mid c \mid \bar{e}\ op\ \bar{e} \\
\textbf{Propositions} & \phi & ::= & \bar{e}\ cmp\ \bar{e} \\
 & & \mid & \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \\
 & & \mid & \exists x.\ \phi \mid \forall x.\ \phi
\end{array}$$

Figure 2. SPECIFICATION LANGUAGE

The semantics of all remaining constructs is defined in the usual way.

Upon reaching a call statement of the form $v{:=}f(e)$, one checks in the order prescribed by the declaration of advices whether the guard of a point-cut descriptor for f is satisfied. If there is no point-cut descriptor for f such that the guard is satisfied, then one starts a new execution frame, initializes the local variable par with the value of e, and executes the body of f; otherwise, if a is the first advice for f whose guard is satisfied, then one starts a new execution frame, initializes the local variable par with the value of e, and executes the body of a.

Upon reaching a proceed statement of the form $v{:=}\ \mathtt{proceed}(e)$, one must examine the call stack to determine the current procedure, say f, and the current advice, say a. Then one checks for all advices that occur after a in the declaration of advices whether the guard of a point-cut descriptor for f is satisfied. If there is no point-cut descriptor for f such that the guard is satisfied, then one starts a new execution frame, initializes the local variable par with the value of e, and executes the body of f; otherwise, if a' is the first advice for f whose guard is satisfied, then one starts a new execution frame, initializes the local variable par with the value of e, and executes the body of a'.

Under such a semantics, the body of f will not be executed whenever a procedure call to f, say $v{:=}f(e)$, triggers an advice that does not contain any proceed statement, or contains a proceed statement that is not reached during execution. Furthemore, if an advice contains two or more proceed statements, then execution will stop upon reaching the second proceed statement.

Formally, the semantics of advice weaving is defined by compilation to an intermediate language SBL, defined in Section 6. For the purpose of the next sections, it is sufficient to know that the semantics of SAL programs can be modeled by judgments of the form

$$p, \mu \Downarrow v, \nu$$

which read: the execution of program p with initial memory μ terminates with final memory ν and returns value v.

4. Verification of baseline code

In this section, we focus on baseline programs, i.e. programs without advices, and introduce for such programs a verification method based on the idea of contract. Therefore, each procedure is specified in terms of a pre-condition, which captures the situations under which the procedure can be called, and a post-condition, which establishes a relationship between the inputs and outputs of the procedure, and a frame condition that specifies which variables that are modified during the execution of f, and that is used by the verification condition generator to improve its context-sensitivity.

The set of propositions is defined in Figure 2, where $x^\star$ is a special, so-called starred, variable representing the initial value of the variable x, and res is a special value representing the final value of the evaluation of the program. Program specifications rely on particular classes of propositions:

- preconditions, which do not refer to starred variables, nor the special variable res;
- postconditions, which do not refer to local variables;
- loop invariants, which do not refer to the special variable res.

Each precondition Φ yields a predicate over states, denoted $\mu \models \Phi$ for a state μ, whereas a post-condition Ψ yields a ternary relation over an initial state, a final state, and a result, denoted $\mu, \nu, v \models \Psi$ for the states μ and ν and the value v. Likewise, invariants yield binary relations over an initial and a current state.

In order to reason effectively about programs, we assume that each procedure is annotated, i.e. that all while loops in its body carries an invariant (we use $\mathsf{while}_I(b)\{s\}$ to denote the loop $\mathsf{while}_I(b)\{s\}$ annotated with invariant I), and that we dispose of a specification table Γ that associates to each procedure f a triple $(\Phi, \Psi, \mathcal{W})$ where Φ is a precondition, Ψ is a postcondition, and $\mathcal{W}$ is a *modifies* clause that declares all variables that are modified during the execution of f.

Given a specification table Γ, one can compute for each annotated procedure f a set $\mathsf{PO}_\Gamma(f)$ of verification conditions. The verification conditions are defined using an extended predicate transformer vcg, which takes as input a baseline command c and a post-condition Ψ, and returns a pre-condition Φ and a set of proof obligations Δ_f. Formally, the set $\mathsf{PO}_\Gamma(f)$ is defined as $\Delta_f \cup \{\Phi \Rightarrow \Phi'[{}^{y}\!/_{y^\star}]\}$, where $\Gamma(f) = (\Phi, \Psi, \mathcal{W})$, y stands for every variable in $\mathcal{V}_\Gamma$ and $\mathsf{vcg}(c, \Psi) = (\Phi', \Delta_f)$, where c is the body of f. We say that a procedure is valid if all its proof obligations are valid formulae, and that a program is valid if all its procedures are. The formal definition of vcg is given in Figure 3.

For the verification method to be sound, we must also check the correctness of the *modifies* clause. Even though we can propose a logic to verify this frame condition, we assume a sound but incomplete automatic analysis that check its correctness.

The weakest precondition calculus is sound in the sense that if a program p is valid w.r.t. a specification table Γ with a main procedure specificied by (Φ, Ψ), then all executions of p initiated with a memory μ satisfying Φ will terminate with a final memory Ψ and value v such that (μ, ν, v) satisfy Ψ.

Lemma 1 (Soundness)**.** *Let p be a baseline program over a set $\mathcal{F}$ of procedures. Let Γ be a specification table for p and let $\Gamma(\mathsf{main}) = (\Phi, \Psi, \mathcal{W})$. Assume that p is valid w.r.t. Γ. Then, if $p, \mu \Downarrow v, \nu$ and $\mu \models \Phi$, then $\mu, \nu, v \models \Psi$.*

In the setting of PCC, we require that proof obligations are certified, i.e. that programs come equipped with independently checkable proofs of their validity. For the purpose of our work, we do not need to commit to any particular format for certificate, nor do we need to specify an algorithm to check certificates. Instead, we rely on an abstract notion of certificate, using the formalism of proof algebra defined in Fig. 4. Finally, we define a certified program as one whose functions are certified, i.e. carry valid certificates for the proof obligations attached to them. Formally, let p be an annotated baseline program and Γ be a specification table. Then, a certificate for the program p w.r.t. Γ is an indexed set of certificates $(c_\delta)_{\delta \in \mathsf{PO}_\Gamma(f), f \in \mathcal{F}}$ such that $c_\delta : \vdash \delta$ for all δ belonging to $\mathsf{PO}_\Gamma(f)$ and for all procedures f. If such a certificate exists, we say that p is certified w.r.t. Γ.

If a program p is certified w.r.t. a specification table Γ, then it is obviously valid w.r.t. Γ.

5. Verifying programs with advices

As illustrated by the examples of Section 2, soundness fails for programs with advice, as expected since verification condition generation is oblivious to aspects. The purpose of this section is to define a method to verify SAL programs; the verification method is based on the notion of specification-preserving advice, which is introduced formally below.

let $\Gamma(f) = (P_f, Q_f, \mathcal{W})$ in

$$\begin{array}{rcl}
\mathsf{vcg}(\mathtt{skip}, \varphi) & = & (\varphi, \emptyset) \\
\mathsf{vcg}(x{:=}e, \varphi) & = & (\varphi[{}^{e}\!/_{x}], \emptyset) \\
\mathsf{vcg}(c_1;c_2, \varphi) & = & \text{let } (\varphi_2, S_2){=}\mathsf{vcg}(c_2, \varphi) \text{ in let } (\varphi_1, S_1){=}\mathsf{vcg}(c_1, \varphi_2) \text{ in}(\varphi_1, S_1 \cup S_2) \\
\mathsf{vcg}(\mathtt{return}\ e, \varphi) & = & (\varphi[{}^{e}\!/_{\mathsf{res}}], \emptyset) \\
\mathsf{vcg}(\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \varphi) & = & \text{let } (\varphi_1, S_1){=}\mathsf{vcg}(c_1, \varphi) \text{ in let } (\varphi_2, S_2){=}\mathsf{vcg}(c_2, \varphi) \text{ in}(b \Rightarrow \varphi_1 \wedge \neg b \Rightarrow \varphi_2, S_1 \cup S_2) \\
\mathsf{vcg}(\mathtt{while}\ b\ \{Inv\}\ \mathtt{do}\ c, \varphi) & = & \text{let } (\varphi', S){=}\mathsf{vcg}(c, Inv) \text{ in}(Inv, \{Inv \Rightarrow (b \Rightarrow \varphi' \wedge \neg b \Rightarrow \varphi)\} \cup S) \\
\mathsf{vcg}(x{:=}f(e), \varphi) & = & P_f[{}^{e}\!/_{\mathtt{in}_f}] \wedge (\forall_{\mathcal{W}', \mathsf{res}}. Q_f[{}^{e}\!/_{\mathtt{in}_f}][{}^{\mathcal{W}'}\!/_{\mathcal{W}}][{}^{\mathcal{W}}\!/_{\mathcal{W}^\star}] \Rightarrow \varphi[{}^{\mathsf{res}}\!/_{x}][{}^{\mathcal{W}'}\!/_{\mathcal{W}}], \emptyset) \\
\mathsf{vcg}_f(x{:=}\ \mathtt{proceed}(e), \varphi) & = & P_f[{}^{e}\!/_{\mathtt{in}_f}] \wedge (\forall_{\mathcal{W}', \mathsf{res}}. Q_f[{}^{e}\!/_{\mathtt{in}_f}][{}^{\mathcal{W}'}\!/_{\mathcal{W}}][{}^{\mathcal{W}}\!/_{\mathcal{W}^\star}] \Rightarrow \varphi[{}^{\mathsf{res}}\!/_{x}][{}^{\mathcal{W}'}\!/_{\mathcal{W}}], \emptyset)
\end{array}$$

Figure 3. WEAKEST PRECONDITION FUNCTION

$$\begin{array}{ll}
\mathsf{intro}_{\mathsf{true}} & \mathcal{P}(\Gamma \vdash \mathsf{true}) \\
\mathsf{axiom} & \mathcal{P}(\Gamma; A; \Delta \vdash A) \\
\mathsf{ring} & \mathcal{P}(\Gamma \vdash n_1 = n_2) \quad \text{if } n_1 = n_2 \text{ is a ring equality} \\
\mathsf{intro}_{\wedge} & \mathcal{P}(\Gamma \vdash A) \rightarrow \mathcal{P}(\Gamma \vdash B) \rightarrow \mathcal{P}(\Gamma \vdash A \wedge B) \\
\mathsf{elim}^{\mathsf{l}}_{\wedge} & \mathcal{P}(\Gamma \vdash A \wedge B) \rightarrow \mathcal{P}(\Gamma \vdash A) \\
\mathsf{elim}^{\mathsf{r}}_{\wedge} & \mathcal{P}(\Gamma \vdash A \wedge B) \rightarrow \mathcal{P}(\Gamma \vdash B) \\
\mathsf{intro}_{\Rightarrow} & \mathcal{P}(\Gamma; A \vdash B) \rightarrow \mathcal{P}(\Gamma \vdash A \Rightarrow B) \\
\mathsf{elim}_{\Rightarrow} & \mathcal{P}(\Gamma \vdash A \Rightarrow B) \rightarrow \mathcal{P}(\Gamma \vdash A) \rightarrow \mathcal{P}(\Gamma \vdash B) \\
\mathsf{elim}_{=} & \mathcal{P}(\Gamma \vdash e_1 = e_2) \rightarrow \mathcal{P}(\Gamma \vdash A[{}^{e_1}\!/_{r}]) \rightarrow \mathcal{P}(\Gamma \vdash A[{}^{e_2}\!/_{r}]) \\
\mathsf{subst} & \mathcal{P}(\Gamma \vdash A) \rightarrow \mathcal{P}(\Gamma[{}^{e}\!/_{r}] \vdash A[{}^{e}\!/_{r}]) \\
\mathsf{weak} & \mathcal{P}(\Gamma \vdash A) \rightarrow \mathcal{P}(\Gamma; \Delta \vdash A) \\
\mathsf{intro}_{\forall} & \mathcal{P}(\Gamma \vdash A) \rightarrow \mathcal{P}(\Gamma \vdash \forall r.A) \qquad \text{if } r \text{ is not in } \Gamma \\
\mathsf{elim}_{\forall} & \mathcal{P}(\Gamma \vdash \forall r.A) \rightarrow \mathcal{P}(\Gamma \vdash A[{}^{t}\!/_{r}])
\end{array}$$

Figure 4. PROOF ALGEBRA

Throughout this section, we consider a program p in which all procedures are annotated, i.e. have loop invariants, and specified in a table Γ. Furthemore, we let $\mathcal{V}_\Gamma$ be the set of variables that appear in the specification of baseline procedures.

5.1 Specification-preserving advices

In order to reason about advices, we extend the verification condition generator to proceed statements. The extension is parameterized by the name of the advised function, and the proceed statement is interpreted as a call to this function; see Figure 3. Note that when reasoning about an advice a, in order for the verification condition generator to be effective we need one set of loop invariants for each procedure f that a is advising.

Definition 1. *An advice a with guard b is specification-preserving w.r.t. f and Γ if it satisfies the specification*

$$(b \wedge \Phi, \Psi, \mathcal{W}')$$

where $\Gamma(f) = (\Phi, \Psi, \mathcal{W})$, and $\mathcal{W}' \cap \mathcal{V}_\Gamma \subseteq \mathcal{W}$.

The condition $\mathcal{W}' \cap \mathcal{V}_\Gamma \subseteq \mathcal{W}$ states that the advice a only modifies in $\mathcal{W}$, unless they do not appear originally on the specification of the baseline program. We let $\mathsf{PO}_{\Gamma,f}(a)$ stand for the set of proof obligations required to prove that the advice a is specification-preserving w.r.t. f and Γ. Formally, if $\Gamma(f) = (\Phi, \Psi, \mathcal{W})$ and c is the body of a, the set $\mathsf{PO}_{\Gamma,f}$ is defined as $\Delta_{a,f} \cup \{\Phi \Rightarrow \phi[{}^{y}\!/_{y^\star}]\}$ where $(\phi, \delta_{a,f}) = \mathsf{vcg}(c, \Psi)$ and $y^\star$ stands for every starred variable in ϕ.

If all advices are specification-preserving, then baseline program verification is sound. To state this result, one first extends the notion of valid advice, and valid program. Let (p, Γ) be an annotated program. We say that an advice a is valid if for all procedures f that it advises, the set of proof obligations $\mathsf{PO}_{\Gamma,f}(a)$ is valid. Then, we say that the program p is valid if all its procedures and all its advices are valid.

We can now state soundness of the verification method in the presence of advice weaving.

Lemma 2 (Soundness)**.** *Let (p, Γ) be a valid annotated program. Then, if $p, \mu \Downarrow v, \nu$ and $\mu \models \Phi$, then $\mu, \nu, v \models \Psi$.*

One can extend the notion of certified baseline program to programs with specification-preserving advices, by requiring that programs come equipped with a certificate that advices are specification-preserving.

Remark. We can extend the scope of this paper to a language with a richer set of point-cut descriptors, for instance to point-cut descriptors that refer to the control-flow graph. To this end, as an alternative to reasoning about the control-flow graph or the call-stack in our logic, we propose a stronger definition of specification preserving advices. An advice a is specification-preserving w.r.t. f and Γ if it satisfies the specification $(\Phi, \Psi, \mathcal{W}')$ where $\Gamma(f) = (\Phi, \Psi, \mathcal{W})$, and $\mathcal{W}' \cap \mathcal{V}_\Gamma \subseteq \mathcal{W}$. Notice that, in contrast to previous definition, the guard b does not appear in the precondition of a.

5.2 Example

To illustrate the approach with a running example we assume an extended program syntax. Consider a procedure $g \doteq \mathtt{slowRetrieve}$ of a SAL program p, that returns the value stored in a slow access memory. That is, given as parameter the integer *Address* $\mathtt{i}$, the procedure g returns the value $\mathtt{mem}[\mathtt{i}]$, where $\mathtt{mem}$ is a global array variable, if $\mathtt{i}$ is within the accessible range.

Since we plan to improve the efficiency of the procedure g, we consider two global array variables $\mathtt{available}$ and $\mathtt{cache}$ and the procedures $f_1 \doteq \mathtt{updateCache}$ and $f_2 \doteq \mathtt{isAvailable}$. Let the proposition ϕ stand for the consistency of the $\mathtt{cache}$ variable with respect to the array $\mathtt{availability}$, i.e.

$$\phi \doteq \forall i. (\mathtt{available}[i] \Rightarrow \mathtt{cache}[i] = \mathtt{mem}[i]) \ .$$

For simplicity, we assume that global variables $\mathtt{available}$ and $\mathtt{cache}$ are only accessible by these procedures.

Consider a specification table Γ such that $\Gamma(g) = (\Phi, \Psi, \mathcal{W})$ where $\Phi \doteq 0 \leq \mathtt{i} < N \wedge \phi$, $\Psi \doteq \mathsf{res} = \mathtt{mem}[\mathtt{i}] \wedge \phi$ and $\mathcal{W} = \emptyset$.

Similarly, we need to specify procedures f_1 and f_2 with their respective pre and post-conditions:

$$\begin{array}{l}
\Phi_1 \doteq \Phi \\
\Psi_1 \doteq \mathtt{cache} = \mathtt{cache}^\star[\mathtt{i} \mapsto \mathtt{v}] \wedge \phi \\
\Phi_2 \doteq 0 \leq \mathtt{i} < N \\
\Psi_2 \doteq \mathsf{res} = \mathtt{available}[i]
\end{array}$$

Consider the introduction of an advice $a \doteq \mathtt{fastRetrieve}$ that improves the store access time by taking advantage of the array variables $\mathtt{available}$ and $\mathtt{cache}$ and the procedures f_1 and f_2. This advice replaces the functionality of method g by receiving as parameter the store address i and returning the *cached* value if available or, otherwise, by permitting the original function g to continue:

```
around slowRetrieve(Address i) fastRetrieve {
    b:= isAvailable(i);
    if b
        return cache[i]
    else
        v:=proceed(i);
        updateCache(i, v);
        return v
}
```

Then, we can prove that a is specification preserving by showing that the proposition

$$\begin{array}{l}
\Phi_2 \wedge \forall_b.(\Psi_2[{}^{b}/_{\mathsf{res}}] \Rightarrow \\
\quad b \Rightarrow \Psi[{}^{\mathtt{cache[i]}}/_{\mathsf{res}}] \wedge \phi \\
\quad \wedge \\
\quad \neg b \Rightarrow \Phi \wedge \forall_{\mathsf{res}}.(\Psi \Rightarrow \Phi_1 \wedge \\
\qquad \forall_{\mathtt{cache}'}.(\Psi_1[{}^{\mathtt{cache}'}/_{\mathtt{cache}}][{}^{\mathtt{cache}}/_{\mathtt{cache}^{\star}}] \Rightarrow \\
\qquad\quad (\Psi \wedge \phi)[{}^{\mathtt{cache}'}/_{\mathtt{cache}}])))
\end{array}$$

is implied by Φ.

5.3 Harmless advices

In general, it is not decidable whether an advice a is specification-preserving w.r.t. a specification table Γ and a procedure f. Therefore, it is of interest to develop automated approximate methods to detect specification-preserving advices. A natural condition is to require that the advice does not modify the variables in $\mathcal{V}_\Gamma$ and always executes a proceed statement. Since such requirements are closely related to the notion of harmless advice, we call such advices specification-harmless.

The set of SAL commands is extended with assertions $\mathsf{assert}(\phi)$ and ghost assignments $\mathsf{set}\ z' := z$, where ϕ is a proposition and z' is a ghost variable not appearing in the original program. The definition of vcg is extended accordingly:

$$\begin{array}{lcl}
\mathsf{vcg}(\mathsf{assert}(\phi), \varphi) & = & (\phi, \{\phi \Rightarrow \varphi\}) \\
\mathsf{vcg}(\mathsf{set}\ z' := e, \varphi) & = & (\phi[{}^{e}/_{z'}], \emptyset)
\end{array}$$

Formally, an advice a with parameters $\vec{y}$ and guard b is specification-harmless w.r.t. f and Γ if the procedure $\hat{a}$ whose body is obtained from the body of a by substituting $x{:=}\ \mathtt{proceed}(\vec{e})$ by

$$\mathsf{assert}(\vec{z^{\star}} = \vec{z}); x{:=}f(\vec{y}); \mathsf{set}\ x', \vec{z'} := x, \vec{z}$$

satisfies the specification

$$(b \wedge \Phi, x' = \mathsf{res} \wedge \vec{z'} = \vec{z}, \mathcal{W}')$$

where $\Gamma(f) = (\Phi, \Psi, \mathcal{W})$, and $\mathcal{W}' \cap \mathcal{V}_\Gamma = \emptyset$, and where $x', \vec{z'}$ are fresh ghost variables, and where $\vec{z}$ is an enumeration of $\mathcal{V}_\Gamma$. We classify an advice as *control flow preserving* if every path in its control flow contains exactly one `proceed` statement. We assume an automated approximate static analysis for this condition.

Lemma 3. *Let a be a control-flow preserving advice. Then, if a is specification-harmless advice with respect to f and Γ, then it is specification-preserving.*

Dantas and Walker [14] propose a mechanism to check that the execution of an advice does not interfere with the final value produced by the computation of the baseline procedure. It consists on a type-effect system inspired on information flow type systems that does not consider timing nor termination behaviour. One can use this type system as a static analysis to detect whether an advice is specification-harmless.

$$\begin{array}{rcl}
instr & ::= & \mathsf{nop} \\
 & & \mid \mathsf{push}\ v \\
 & & \mid \mathsf{load}\ x \\
 & & \mid \mathsf{store}\ x \\
 & & \mid \mathsf{jmp}\ l \\
 & & \mid \mathsf{jmpif}\ cmp\ l \\
 & & \mid \mathsf{invoke} \\
 & & \mid \mathsf{return}
\end{array}$$

Figure 5. INSTRUCTION SET FOR SBL

5.4 Beyond harmless advices

There are many natural examples of advices that do not necessarily trigger a proceed statement. For example, advices that seek to improve efficiency by replacing a procedure call by a semantically equivalent but more efficient computation will not call a proceed statement. For such examples of advices, it is still possible to use the property of specification-harmless to ensure that the advice is specification-preserving for those paths in which a proceed statement is effectively called, and generate a proof obligation for all paths that do not call to proceed.

Recall the advice of the basic example shown in Section 2:

$$\begin{array}{rcl}
a(x) & = & (\mathtt{if}\ x \neq 0\ \mathtt{then}\ z := \mathsf{proceed}(x)\ \mathtt{else}\ z := 0); \\
 & & \mathsf{return}\ z
\end{array}$$

Clearly, we have two possible execution paths depending on whether the input value is equal to 0 or not. To verify that a preserves the specification of f, i.e. $(\mathsf{true}, \mathsf{res} = x^{\star} + x^{\star})$, we consider each possible path separately. In case that the parameter x is not equal to 0 we know that exactly one `proceed` statement will be executed, that no variable is modified and that the expression returned by the proceed statement is passed unchanged by the advice. Thus, we can use a simple static analysis to detect whether this path is specification-harmless. However, the path corresponding to an input equal to 0 does not execute a `proceed` statement, so we need to generate proof obligations that ensures that the specification is still preserved. In this case, it corresponds to the valid proposition $x = 0 \Rightarrow 0 = x + x$.

6. Compiling advices

From an applicative perspective, aspect-orientation is transparent and AOP compilers target typical back-ends: indeed, it is the role of the compiler to integrate these concerns into a single executable object, through a weaving mechanism that modifies the code of each procedure depending on the advices that operate over it. In this section, we define the compilation of SAL programs to a stack-based language.

6.1 Target language

The target language is a simple stack-based language (SBL) that can be used to compile the imperative core of SAL. The syntax of SBL instructions is given in Figure 5, where v and l ranges over integers, x ranges over program variables, cmp over relations between integer values, and g ranges over function names. A SBL program consists of a set of function names, and for each function g a declaration of the form $g\ args^{*} = instr^{*}$. The operational semantics of SBL programs is standard, and defined by a small-step relation $\leadsto$ between states. A state is either final, in which case it consists of a global memory μ and a result value v, or intermediary, in which case it consists of a global memory μ and a set of frames lf, each frame consisting of the name of the function being called, of a program counter, of a local memory with a distinguished variable par that stores the parameter of the function being called,

$$\frac{p_f[i] = \mathsf{invoke}\ f}{\langle \mu, \langle f', pc, lm, v : os\rangle :: lf\rangle \rightsquigarrow \langle \mu, \langle f, 1, [par \mapsto v], \epsilon\rangle :: \langle f', pc+1, lm, os\rangle :: lf\rangle}$$

$$\frac{p_f[i] = \mathsf{return}}{\langle \mu, \langle f, pc, lm, v : os\rangle :: \langle f', pc', lm', os'\rangle :: lf\rangle \rightsquigarrow \langle \mu, \langle f', pc', lm', v :: os'\rangle :: lf\rangle}$$

Figure 6. OPERATIONAL SEMANTICS OF SBL

and of an operand stack. Figure 6 gives the rules for invoke and return instructions, where $[par \mapsto v]$ denotes the local memory that only assigns v to par.

6.2 Compiler

The compiler for SAL programs is defined in Figure 7 as a function $[\![]\!]$ that takes a command and returns a list of labeled instructions. It relies on a compiler for integer expressions and a compiler for boolean conditions, namely $[\![]\!]_{\mathsf{e}}$ and $[\![]\!]_{\mathsf{b}}$. The compiler $[\![]\!]_{\mathsf{e}}$ takes an integer expression e and returns a sequence of instructions whose effect is to push on top of the stack the evaluation of the expression e. The compiler $[\![]\!]_{\mathsf{b}}$ takes, in addition to a boolean expression b, a label l and outputs a sequence a instructions that forces the program execution to jump to the program point labeled l if the condition b evaluates to true. The compiler for commands is standard, to the exception of the function call statement, whose compilation involves advice weaving, and the `proceed` statement. Since SBL does not feature a dedicated mechanism for advice weaving, each advice is compiled multiple times, exactly once per procedure it advises, and the procedure call $x{:=}f(e)$ is compiled into

$$[\![e]\!]_{\mathsf{e}} :: \mathsf{invoke}\ \hat{a}_f :: \mathsf{store}\ x$$

where a is the first advice for f, and $\hat{a}_f$ is its specific compilation for f. The code of $\hat{a}_f$ is of the form

$$[\![b, l]\!]_{\mathsf{b}} :: \mathsf{load}\ par :: \mathsf{invoke}\ \hat{a}'_f :: \mathsf{return} :: [l : a_f]$$

where a_f is obtained by compilation from a by translating any proceed statement of the form $x{:=}\ \mathtt{proceed}(e)$ by

$$[\![e]\!] :: \mathsf{invoke}\ a'_f :: \mathsf{store}\ x$$

where a' is the next advice for f. In other words, the code of $\hat{a}_f$ tests if the guard for a holds, and if so proceeds to execute the body of the advice, or lets $\hat{a}'_f$ proceed otherwise.

In order to achieve the desired effect, the compiler is thus parametrized by a procedure (used in the clause for procedure calls to trigger the appropriate advice), or by a procedure and an advice (used in the clause for proceed to trigger the appropriate advice). For readability, we use superscripts to indicate the parameter and omit the superscript in all cases where it is not used.

$$\begin{array}{lll}
[\![\mathtt{skip}]\!] = [l\,{:}\,\mathsf{nop}] & & \\
[\![x{:=}e]\!] = & \mathsf{let\ ins}_e{=}[\![e]\!]_{\mathsf{e}}\ \mathsf{in} & \\
 & \mathsf{ins}_e :: \mathsf{store}\ x & \\
[\![c_1;c_2]\!] = & \mathsf{let\ ins}_1{=}[\![c_1]\!]\ \mathsf{in} & \\
 & \mathsf{let\ ins}_2{=}[\![c_2]\!]\ \mathsf{in} & \\
 & \mathsf{ins}_1 :: \mathsf{ins}_2 & \\
\multicolumn{3}{l}{[\![\mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2]\!] =} \\
\multicolumn{3}{l}{\qquad \mathsf{let\ ins}_1{=}[\![c_1]\!]\ \mathsf{in}} \\
\multicolumn{3}{l}{\qquad \mathsf{let\ ins}_2{=}[\![c_2]\!]\ \mathsf{in}} \\
\multicolumn{3}{l}{\qquad \mathsf{let\ ins}_b{=}[\![b, l_1]\!]_{\mathsf{b}}\ \mathsf{in}} \\
\multicolumn{3}{l}{\qquad \mathsf{ins}_b :: \mathsf{ins}_2 :: \mathsf{jmp}\ l :: [l_1\,{:}\,\mathsf{ins}_1] :: [l\,{:}\,\mathsf{nop}]} \\
\multicolumn{3}{l}{[\![\mathtt{while}\ b\ \mathtt{do}\ c]\!] =} \\
\multicolumn{3}{l}{\quad \mathsf{let\ ins}_c{=}[\![c]\!]\ \mathsf{in}} \\
\multicolumn{3}{l}{\quad \mathsf{let\ ins}_b{=}[\![b, l_c]\!]_{\mathsf{b}}\ \mathsf{in}} \\
\multicolumn{3}{l}{\quad \mathsf{jmp}\ l :: [l_c\,{:}\,\mathsf{ins}_c] :: [l\,{:}\,\mathsf{ins}_b]} \\
[\![x{:=}h(e)]\!]^f = & \mathsf{let\ ins}_e{=}[\![e]\!]_{\mathsf{e}}\ \mathsf{in} & \\
 & \mathsf{ins}_e :: \mathsf{invoke}\ a_f :: \mathsf{store}\ x & \\
[\![\mathtt{return}\ e]\!] = & \mathsf{let\ ins}{=}[\![e]\!]_{\mathsf{e}}\ \mathsf{in} & \\
 & \mathsf{ins} :: \mathsf{return} & \\
[\![x{:=}\ \mathtt{proceed}(e)]\!]^a_f = & \mathsf{let\ ins}_e{=}[\![e]\!]_{\mathsf{e}}\ \mathsf{in} & \\
 & \mathsf{ins}_e :: \mathsf{invoke}\ a'_f :: \mathsf{store}\ x &
\end{array}$$

Figure 7. COMPILER FOR SAL PROGRAMS

$$\begin{array}{lcl}
\textbf{stack expressions} & \overline{os} & ::= \ \mathsf{os} \mid \bar{e} :: \overline{os} \mid \uparrow^k \overline{os} \\
\textbf{logical expressions} & \bar{e} & ::= \ \mathsf{res} \mid x^\star \mid x \mid c \mid \bar{e}\ op\ \bar{e} \mid \overline{os}[k]
\end{array}$$

Figure 8. LOGICAL SBL EXPRESSIONS

7. Certificate translation

In this section, we show that a valid SAL program is compiled into a valid SBL program. To this end, we first define a verification method for SBL programs. The method is strongly inspired from earlier work, and in particular [8].

7.1 Verification of SBL programs

While program annotations are similar to those of SAL programs, the weakest precondition computation will produce propositions that refer to the operand stack, and thus the language of SBL annotations is extended to such propositions.

- The extended set of logical expressions is defined in Figure 8; the logical propositions are built as before. In the definition, os is a special variable representing the current operand stack and $\uparrow^k\ \overline{os}$ denotes the stack $\overline{os}$ minus its k-first elements. An annotation is a proposition that does not contain stack subexpressions.
- An annotated bytecode instruction is either a bytecode instruction or a proposition and a bytecode instruction:

$$\bar{i} \quad ::= \quad i \mid (\phi, i)$$

- An annotated program is a pair (p, Γ), where p is a bytecode program in which some instructions are annotated and Γ is a specification table that associates to each procedure f a triple $(\Phi, \Psi, \mathcal{W})$ where Φ is a precondition, Ψ is a postcondition, and $\mathcal{W}$ is a *modifies* clause that declares all variables that may be modified during the execution of f.

Verification of SBL programs is defined in terms of a weakest precondition function wp that operates on annotated programs. In order for the wp function to be well-defined, we must restrict our attention to well-annotated programs [5, 8, 26], i.e. programs in which all cycles in the control-flow graph must pass through an annotated instruction. We characterize such programs by an inductive and decidable definition.

An annotated program p is well-annotated if every procedure is well annotated. A procedure g is well-annotated if every program point satisfies the inductive predicate $\mathsf{reachAnnot}_g$ defined by the clauses:

$$\frac{g[k] = (\phi, i)}{k \in \mathsf{reachAnnot}_g} \qquad \frac{g[k] = \mathsf{return}}{k \in \mathsf{reachAnnot}_g}$$
$$\frac{\forall k'.\ k \mapsto k' \Rightarrow k' \in \mathsf{reachAnnot}_g}{k \in \mathsf{reachAnnot}_g}$$

Given a well-annotated procedure, one can generate an assertion for each label, using the assertions that were given or previously computed for its successors. This assertion represents the pre-condition that an initial state before the execution of the corresponding label should satisfy for the procedure to terminate only in a state satisfying its post-condition.

Let (p, Γ) be a well-annotated program.

- The weakest precondition calculus over (p, Γ) is defined in Figure 9. Formally, the result of the weakest precondition calculus is a program in which all instructions are annotated.
- The set $\mathsf{PO}(f)$ of verification conditions of the procedure f is defined by the clauses:

$$\frac{}{\Phi \Rightarrow \mathsf{wp}_{\mathcal{L}}(0)[^{\vec{x^\star}}/_{\vec{x}}] \in \mathsf{PO}_\Gamma(f)} \qquad \frac{f[k] = (\phi, i)}{\phi \Rightarrow \mathsf{wp}_i(k) \in \mathsf{PO}_\Gamma(f)}$$

As before, an annotated SBL program is valid with respect to a specification table Γ if all its sets proof obligations $\mathsf{PO}_\Gamma(f)$ are valid.

7.2 Preservation of validity

The purpose of this section is to prove that valid SAL programs are compiled into valid SBL programs. To this end, we first extend the compiler of Section 6 so that compiled programs are well-annotated. This is achieved by modifying the compiler clause for loops:

$$\begin{array}{rl} [\![\mathsf{while}_I(b)\{c\}]\!] = & \mathsf{let}\ \mathsf{ins}_c{=}[\![c]\!]\ \mathsf{in} \\ & \mathsf{let}\ \mathsf{ins}_b{=}[\![b, l_c]\!]\ \mathsf{in} \\ & \mathsf{jmp}\ l :: [l_c : \mathsf{ins}_c] :: [l : (I, \mathsf{ins}_b)] \end{array}$$

where we denote (I, ins_b) the sequence of instructions obtained by annotating the first instruction of ins_b with I. In the rest of this section, for any SBL function g, we denote $g[l, l']$ the sequence of instructions $g[l] :: g[l+1] :: \ldots :: g[l'-1]$.

Lemma 4. *Assuming the axioms for stacks $(v :: \mathsf{os})[0] = v$ and $\uparrow (v :: \mathsf{os}) = \mathsf{os}$, he auxiliary compilers $[\![]\!]_\mathsf{e}$ and $[\![.]\!]_\mathsf{b}$ satisfy the following properties:*

i) for every integer expression e and function g such that $g[l, l'] = [\![e]\!]_\mathsf{e}$, $\mathsf{wp}_{\mathcal{L}}(l)$ is equivalent to $\mathsf{wp}_{\mathcal{L}}(l')[^{e::\mathsf{os}}/_{\mathsf{os}}]$;

ii) for every boolean expression b and function f such that $g[l, l''] = [\![b, l']\!]_\mathsf{b}$, $\mathsf{wp}_{\mathcal{L}}(l)$ is equivalent to

$$b \Rightarrow \mathsf{wp}_{\mathcal{L}}(l') \wedge \neg b \Rightarrow \mathsf{wp}_{\mathcal{L}}(l'')$$

Given a specification table Γ for SAL programs, we say that Γ' is a specification table for SBL programs extending Γ if for every advice a and any procedure f advised by a, $\Gamma'(\hat{a}_f) = (\Phi_f, \Psi_f, \mathcal{W}_f)$ and $\Gamma'(a_f) = (\Phi_f \wedge b, \Psi_f, \mathcal{W}_f)$, where $\Gamma(f) = (\Phi_f, \Psi_f, \mathcal{W}_f)$. In the following paragraphs, we implicitly consider the specification tables Γ and Γ' respectively for the verification of SAL and SBL programs.

Lemma 5. *Let g be a SBL function such that $g[l, l'] = [\![c]\!]$, and let $(\phi, S) = \mathsf{vcg}(c, \mathsf{wp}_{\mathcal{L}}(l'))$. Then, $\phi' \equiv \mathsf{wp}_{\mathcal{L}}(l)$ and the proof obligations in S are equivalent to the proof obligations corresponding to the annotated instructions in $g[l, l']$.*

Consider a SBL program p' compiled from an annotated SAL program p. The following result states that if p is a valid SAL program with respect to Γ, then p' is a valid SBL program with respect to Γ'.

Theorem 1. Suppose that (p, Γ) is a valid annotated program. That is, for every procedure f and for every advice a, the sets of proof obligations Δ_f and $\mathsf{PO}_{\Gamma,f}(a)$ are valid. Then, for every function f, a_f and $\hat{a}_f$, the sets $\mathsf{PO}_{\Gamma'}(f)$, $\mathsf{PO}_{\Gamma'}(a_f)$ and $\mathsf{PO}_{\Gamma'}(\hat{a}_f)$ contain valid proof obligations.

Furthermore, we can prove that a SAL programs certified with respect to Γ is compiled into a SBL program certified with respect to Γ'. More precisely, using the rules of the proof algebra extended with the axioms $(v :: \mathsf{os})[0] = v$ and $\uparrow (v :: \mathsf{os}) = \mathsf{os}$, for every equivalent proof obligations δ and δ', we can transform a certificate c_δ for δ to a certificate $c_{\delta'}$ for δ'. Therefore, if for every procedure $f \in \mathcal{F}$, $(c_\delta)_{\delta \in \mathsf{PO}_\Gamma(f)}$ and $(c_\delta)_{\delta \in \mathsf{PO}_{\Gamma,f}(a)}$ are indexed sets of certificates for a SAL program p, then for every function g of p' we can generate a certificate for the proof obligation $\delta \in \mathsf{PO}_{\Gamma'}(g)$.

8. Increasing the Power of Verification

Consider a procedure f executing under the advice of a_1 and a_2, and suppose that neither a_1 nor a_2, when executed in isolation, preserve the specification of f. However, it can be the case that the execution of each advice, when complemented by the execution of the other one, preserves the specification of f. Then, since it may seem a bit restrictive to require that every advice in its own is specification-preserving, we propose to study instead whether a sequence of advices is specification preserving.

While gaining in completeness, this more general verification method makes more difficult to find an automated approximate procedure.

Verification of advices in isolation. We extend the specification of advices such that for every advice a we have, in addition to the tuple $(\Phi, \Psi, \mathcal{W})$, a specification for the code invoked by a $\mathsf{proceed}$ statement. That enables to reason about the correctness of an advice without considering the possible contexts in which this advice may be invoked. More precisely, the specification extension for an advice a consists on an extra and distinct tuple $(\Phi', \Psi', \mathcal{W}')$, in addition to the tuple $(\Phi, \Psi, \mathcal{W})$. The tuple $(\Phi', \Psi', \mathcal{W}')$ is such that $\mathcal{W}'$ specifies the set of variables that the invoked code is allowed to modify, and Φ' and Ψ' are respectively the pre and post-conditions of such invocation. The propositions Φ' and Ψ' may refer, in addition to the input and output arguments of a (`in` and `res`), to the input and output arguments of the invoked code, respectively represented with the new variables `in'` and `res'`. To complete the proof, a second phase explained in the next paragraphs checks, for every context in which the advice a may be executed, that the code allowed to proceed satisfies a specification that is consistent with the extension of the specification of a.

The predicate transformer wp is, thus, modified accordingly for `proceed` statements:

$$\begin{array}{l} \mathsf{wp}_a(x{:=}\ \mathtt{proceed}(e), \phi) = \\ \quad (\Phi'_a[^e/_{\mathtt{in}'_a}] \\ \qquad \wedge \forall_{y', \mathsf{res}'}.\Psi'_a[^e/_{\mathtt{in}'_a}][^{y'}/_y][^y/_{y^{\star}{}'}] \Rightarrow \phi[^{res'}/_x][^{y'}/_y][^e/_{\mathtt{in}'_a}], S) \end{array}$$

where $(\Phi', \Psi', \mathcal{W}')$ correspond to the specification of the `proceed` statement and $y \in \mathcal{W}'$.

By using this modified wp function we can prove that the body of an advice satisfies its specification as long as the code invoked by a `proceed` statement satisfies the specification $(\Phi', \Psi', \mathcal{W}')$.

Verifying weaved code. After statically determining the sequence of advices $\vec{a}_f$ executing around a procedure f, we are interested in identifying a set of sufficient proof obligations that ensures that the sequence $\vec{a}_f$ is specification-preserving.

let $\Gamma(f) = (P_f, Q_f, \mathcal{W})$ and y represent every variable in $\mathcal{W}$:

$$\begin{array}{rcll}
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)[^{c::\mathsf{os}}/_{\mathsf{os}}] & \text{if } g[k] = \mathsf{push}\ c \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)[^{(\mathsf{os}[0]\ op\ \mathsf{os}[1])::\uparrow^2\mathsf{os}}/_{\mathsf{os}}] & \text{if } g[k] = \mathsf{binop}\ op \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)[^{x::\mathsf{os}}/_{\mathsf{os}}] & \text{if } g[k] = \mathsf{load}\ x \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)[^{\uparrow\mathsf{os},\mathsf{os}[0]}/_{\mathsf{os},x}] & \text{if } g[k] = \mathsf{store}\ x \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(l) & \text{if } g[k] = \mathsf{jmp}\ l \\
\mathsf{wp}_i(k) & = & \qquad (\mathsf{os}[0]\ \neq 0 \Rightarrow \mathsf{wp}_{\mathcal{L}}(k+1)[^{\uparrow^1\mathsf{os}}/_{\mathsf{os}}]) & \text{if } g[k] = \mathsf{jmpif}\ l \\
 & & \wedge \quad \mathsf{os}[0] = 0 \Rightarrow \mathsf{wp}_{\mathcal{L}}(l)[^{\uparrow^1\mathsf{os}}/_{\mathsf{os}}]) & \\
\mathsf{wp}_i(k) & = & \Psi[^{\mathsf{os}[0]}/_{\mathsf{res}}] & \text{if } g[k] = \mathsf{return} \\
\mathsf{wp}_i(k) & = & P_f[^{\mathsf{os}[0]}/_{\mathtt{in}}] & \text{if } g[k] = \mathsf{invoke}\ f \\
 & & \wedge & \\
 & & (\forall \mathsf{res}, y'.Q_f[^{\mathsf{os}[0]}/_{\mathtt{in}}][^{y}/_{y^\star}][^{y'}/_{y}] \Rightarrow \mathsf{wp}_{\mathcal{L}}(k+1)[^{\mathsf{res}::\mathsf{os}}/_{\mathsf{os}}][^{y'}/_{y}]) & \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } g[k] = \phi : i \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise}
\end{array}$$

Figure 9. WEAKEST PRECONDITION FOR SBL PROGRAMS

Since we do not require that every sub-sequence of advices preserves the specification, we must consider a judgement of the form

$$\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\} a_i \ldots a_j \{\Psi\}$$

for every $a_i \ldots a_j$ sub-sequence of $\vec{a}_f$.

To verify a judgement $\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\}\vec{a}_g\{\Psi\}$, we proceed by induction on the length of the sequence $\vec{a}_g$ to identify the set of proof obligations $\Delta_{\vec{a}_g}(\Phi, \Psi)$.

Given a non-trivial sequence $\vec{a}_g = a :: \vec{a}'_g$, we consider two alternative sets of verification conditions, depending on whether we can statically ensure that the code of the advice a *control flow preserving*. We assume an automated static mechanism to check this condition.

In case that it cannot be checked whether a is control-flow preserving we apply the following rule:

$$\frac{\begin{array}{c}\Gamma_{\mathsf{a}}(a) = \langle(\Phi_a, \Psi_a, \mathcal{W}_a), (\Phi'_a, \Psi'_a, \mathcal{W}'_a)\rangle \\ \Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi'\}\vec{a}'_g\{\Psi'\} \\ \Phi'_a \Rightarrow \Phi'[^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}] \qquad \Psi'[^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][^{\mathsf{res}'}/_{\mathsf{res}}] \Rightarrow \Psi'_a \qquad \mathcal{W}_g \cup \mathcal{W}_{\vec{a}'_g} \subseteq \mathcal{W}'_a\end{array}}{\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi_a\} a :: \vec{a}'_g \{\Psi_a\}}$$

For simplicity, we are not considering the boolean condition specified in the point-cut descriptor.

Unfortunately, the rule above makes hard to propagate the information carried by the specification (Φ', Ψ'), unless it is explicitly stated in the specification (Φ_a, Ψ_a) of a. However, under the hypothesis that a is a *control flow preserving* advice we can apply the following alternative rule:

$$\frac{\begin{array}{c}\Gamma_{\mathsf{a}}(a) = \langle(\Phi_a, \Psi_a, \mathcal{W}_a), (\Phi'_a, \Psi'_a, \mathcal{W}'_a)\rangle \\ \Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi'\}\vec{a}'_g\{\Psi\} \\ \Phi \Rightarrow \Phi_a \wedge \forall x'.(\Phi'_a[^{x'}/_{x}] \Rightarrow \Phi'[^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][^{x'}/_{x}]) \qquad \mathcal{W}_g \cup \mathcal{W}_{\vec{a}'_g} \subseteq \mathcal{W}'_a \\ \Psi'[^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][^{\mathsf{res}'}/_{\mathsf{res}}][^{y^{\star\prime}}/_{y^\star}] \Rightarrow \Psi'_a \wedge \forall x'.(\Psi_a[^{\mathtt{in}}/_{\mathtt{in}_a}][^{x'}/_{x}] \Rightarrow \Psi[^{x'}/_{x}])\end{array}}{\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\} a :: \vec{a}'_g \{\Psi\}}$$

where x' represents the global variables potentially modified by a, and W'_a specifies the variables that are allowed to be modified by the execution triggered by the `proceed` statement.

For every procedure f advised by $\vec{a}_f$, we define $\Delta_{\vec{a}_f}(\Phi, \Psi)$ as the set of proof obligations required to derive the judgement $\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\}\vec{a}_f\{\Psi\}$. Assume the specification table Γ is such that $\Gamma(f) = (\Phi_f, \Psi_f, \mathcal{W})$. Then, we say that the sequence $\vec{a}_f$ is specification preserving with respect to f, Γ and Γ_{a}, if $\Phi_f \Rightarrow \Phi$, $\Psi \Rightarrow \Psi_f$ and the proof obligations in $\Delta_{\vec{a}_f}(\Phi, \Psi)$ are valid.

Lemma 6. *Let p be a* SAL *program over a set $\mathcal{F}$ of procedures and a set $\mathcal{A}$ of advices. Let Γ be a specification table for $\mathcal{F}$ and Γ_{a} be a specification table for $\mathcal{A}$. Assume that for every procedure f that is advised by $\vec{a}_f$, the sequence $\vec{a}_f$ is specification preserving with respect to f, Γ and Γ_{a}. Then, if $p, \mu \Downarrow v, \nu$ and $\mu \models \Phi$, then $\mu, \nu, v \models \Psi$.*

The dynamic nature of some point-cut descriptors can make static verification a difficult task. Consider for example a cflow point-cut descriptor, for which program semantics must refer to a collecting call stack to decide whether a cflow condition is valid.

Although possible, it is cumbersome to reason explicitly about the call stack in the program logic. Furthermore, we cannot associate a priori a condition specifiable in our logic to each cflow declaration.

We propose the following simple derivation rule to reason in the presence of cflow point-cut descriptors:

$$\frac{\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\} a \triangleright \vec{a}'_g \{\Psi\} \qquad \Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\}\vec{a}'_g\{\Psi\}}{\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\} a \overset{\mathsf{cflow}}{\triangleright} \vec{a}'_g \{\Psi\}}$$

where $a \overset{\mathsf{cflow}}{\triangleright} \vec{a}_g$ denotes that the execution of the advice a is conditional on cflow statement. The rule can be interpreted as the fact that, regardless of whether the cflow condition is valid, the specification (Φ, Ψ) will still be verifiable with respect to the sequence of advices $a \overset{\mathsf{cflow}}{\triangleright} \vec{a}'_g$. Although incomplete, this rule may prove to be useful as long as the advice a is specification preserving with respect to (Φ, Ψ).

We have formally proved the soundness of the verification method proposed in this section. In addition, we have shown how to extended the compiler with a mechanism to translate a formal certificate of correctness of a SAL program to a certificate for the compiled code.

9. Related work

Reasoning about advices As the invasive nature of aspects cause them to break modularity, the design of sound and practical verification methods for aspect-oriented programs is particularly challenging. There have been many works that explore the design space for such verification methods, and propose different tradeoffs between the modularity of verification and the generality of the method. In addition, there are been many works that isolate particular classes of aspects that are well-suited for modular reasoning and provide automatic analysis methods to detect when an advice fits in one of these classes. We mention some of the most relevant work below.

Clifton and Leavens [12] define a notion of modular reasoning and show why modularity is not a general property in AspectJ and how this can be improved. They define a classification for aspects as *spectators* or *assistants*: the former include aspects that only modify the state space they own and do not alter the control flow, whereas *assistants* can interfere with the original behavior of the program but only if explicitly accepted by the original program. Based on this classification, Clifton and Leavens suggest a verification method, which is described in more detail in [11]. More recently, Clifton, Leavens and Noble [13] have developed an effect system to specify and verify the control and heap effect of aspects in the MAO language. Their system helps to verify whether an advice is a spectator, and provides valuable static information exploitable by subsequent verification. To our best knowledge, there is however no sound program verification method based on these ideas, although Clifton [11] argues informally that the method he proposes is indeed sound. In a similar vein, Rinard *et al* [27] provide a classification of advices, and a static analysis that automatically classifies aspects. They illustrate the usefulness of their static analysis, but do not develop any verification mechanism based on it.

In addition to these works, here have been several efforts to develop modular model-checking techniques for AOP. The prevailing trend to achieve modularity is to isolate specific classes of aspects that exhibit an appropriate behavior. Early work by Katz et al. [18] proposes a classification of aspects as *spectative*, *regulative* or *invasive*, and analyze the class of temporal properties that are preserved by aspects that fall in these categories. In a subsequent work, Goldman and Katz [17] have formalized the idea that *weakly invasive* aspects preserve temporal properties. More recently, Djoko Djoko *et al* [15] have given a formal treatment of similar ideas based on a slightly different classification. These works resembles our own in the sense that they favor modularity of the verification process and makes emphasis on the preservation of original properties. Krishnamurthi *et al* [19] propose an alternative method where modularity is achieved by requiring that the set of point-cut designators is known statically.

While the above works consider different classes in which advices are allowed to interfere more or less with baseline programs, Dantas and Walker [14] choose to consider advices that are optimally suited for modular verification. They define the notion of *harmless advice*, which may interfere with the control flow (by preventing termination) and may also perform I/O, but it does not interfere with the final result of the underlying code. This weak interference property is an instance of specification-preserving advice, and thus permits to reason about the original program independently. They propose an information-flow type system over a core AOP language [28] to check harmlessness with respect to the main program. As discussed in Section 5.3, their type system can be combined to form part of our hybrid logic to certify and check that an advice does not interfere with the original global state.

Aldrich [1] has proposed a module system called "Open Modules" that enables class interfaces to explicitly control the visibility of internal control-flow points. Thus, it provides a mechanism to restrict the interference of external advice, by forbidding the attachement of advices to hidden internal join-points.

Proof compilation As discussed in the introduction, there have been several efforts to study proof compilation for non-optimizing and optimizing compilers. Our work is most closely based on the work of [8], who show that, given a specific VCGen, a sufficiently simple compiler generates, from an imperative source program, a stack based low-level piece of code, whose proof obligations are syntactically equal to that of the source program. Similar results on a wider verification framework are detailed by Pavlova [26], for a significant subset of Java Bytecode.

There has been a closely related effort by Zhao and Rinard [29] to provide state-of-the-art specification and verification tools for AOP, and to relate them to standard verification. They have defined Pipa [29], an extension to JML [20] for AspectJ [2], to support specification for aspects invariants, pre and post-conditions for advices and variable introductions, and provided a compiler that transforms a Pipa-annotated AspectJ program into a JML-annotated Java program. However, they do not provide any formal treatment to support their approach.

10. Conclusion

We have introduced the notion of specification-preserving advice, that mildly generalizes the notion of harmless advice of Dantas and Walker, and that is expressive enough to capture many advices related to security and efficiency. In addition, we have developed a modular verification method for programs with specification-preserving advices, and shown how proof compilation extends naturally to this setting. Our results, while preliminary, establish the feasibility of a Proof Carrying Code scenario with untrusted intermediaries modifying the code by aspects. In future work, we intend to build on the theoretical and practical efforts of the Mobius project on proof compilation for Java and extend our results towards an expressive fragment of AspectJ, taking into account recent developments in optimizing compilation for aspects [3]. In addition, it would be interesting to target our compiler to low level languages with appropriate support for aspects [16], and investigate certificate translation in that setting.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In A. P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer, 2005.

[2] AspectJ Team. The AspectJ programming guide. Version 1.5.3. Available from `http://eclipse.org/aspectj`, 2006.

[3] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc* : An extensible aspectj compiler. 3880:293–334, 2006.

[4] G. Barthe, L. Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. MOBIUS: Mobility, ubiquity, security. objectives and progress report. In *Trustworthy Global Computing*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[5] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2006.

[6] G. Barthe, B. Grégoire, and M. Pavlova. Preservation of proof obligations for java. Draft paper, 2008.

[7] G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In S. Drossopoulou, editor, *ESOP*, Lecture Notes in Computer Science. Springer, 2008. To appear.

[8] G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In T. Dimitrakos, F. Martinelli, P. Y. A. Ryan, and S. A. Schneider, editors, *Formal Aspects in Security and Trust*, volume 3866 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2005.

[9] L. Burdy and M. Pavlova. Java bytecode specification and verification. In *Symposium on Applied Computing*, pages 1835–1839. ACM Press, 2006.

[10] J. Charles, B. Gregoire, and P. Müller H. Lehner. Automatic certificate translation for proof carrying code. Draft paper, 2008.

[11] C. Clifton. *A design discipline and language features for modular reasoning in aspect-oriented programs*. Ph.d. thesis, Iowa State University, 2005.

[12] C. Clifton and G. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning. Technical report, Iowa State University, 2002.

[13] C. Clifton, G. T. Leavens, and J. Noble. Mao: Ownership and effects for more effective reasoning about aspects. In E. Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 451–475. Springer, 2007.

[14] D. S. Dantas and D. Walker. Harmless advice. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 383–396, New York, NY, USA, 2006. ACM Press.

[15] S. Djoko Djoko, R. Douence, and P. Fradet. Aspects preserving properties. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 135–145, New York, NY, USA, 2008. ACM.

[16] R. M. Golbeck and G. Kiczales. A machine code model for efficient advice dispatch. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 2, New York, NY, USA, 2007. ACM.

[17] M. Goldman and S. Katz. Modular generic verification of LTL properties for aspects. In *Foundations of Aspect Languages Workshop (FOAL06)*, 2006.

[18] S. Katz. Aspect categories and classes of temporal properties. In A. Rashid and M. Aksit, editors, *T. Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 106–134. Springer, 2006.

[19] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 137–146, New York, NY, USA, 2004. ACM Press.

[20] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, February 2007.

[21] MOBIUS Consortium. Deliverable 4.1: Scenarios for proof-carrying code. Available online from `http://mobius.inria.fr`, 2006.

[22] P. Müller and M. Nordio. Proof-transforming compilation of programs with abrupt termination. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 39–46, New York, NY, USA, 2007. ACM.

[23] G.C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.

[24] G.C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[25] G.C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI'96*, pages 229–243. Usenix, 1996.

[26] M. Pavlova. *Java bytecode verification and its applications.* Thése de doctorat, spécialité informatique, Université Nice Sophia Antipolis, France, January 2007.

[27] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 147–158, New York, NY, USA, 2004. ACM Press.

[28] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In C. Runciman and O. Shivers, editors, *ICFP*, pages 127–139. ACM, 2003.

[29] J. Zhao and M. C. Rinard. Pipa: A behavioral interface specification language for aspectj. In M. Pezzè, editor, *FASE*, volume 2621 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2003.