Contract-based Slicing

Daniela da Cruz, Pedro Rangel Henriques and Jorge Sousa Pinto
{danieladacruz,prh,jsp}@di.uminho.pt

Departamento de Informática / CCTC
Universidade do Minho, 4710-057 Braga, Portugal

Abstract. In the last years, the concern with the correctness of programs has been leading programmers to enrich their programs with annotations following the principles of *design-by-contract*, in order to be able to guarantee their correct behaviour and to facilitate reuse of verified components without having to reconstruct proofs of correctness.
In this paper we adapt the idea of *specification-based slicing* to the scope of (contract-based) program verification systems and behaviour specification languages. In this direction, we introduce the notion of *contract-based slice* of a program and show how any specification-based slicing algorithm can be used as the basis for a contract-based slicing algorithm.

1 Introduction

Program slicing plays an important role in program comprehension, enabling engineers to focus on just a relevant part (with respect to a given criterion) of a program. After Weiser's pioneering work [13], many researchers have searched for more effective or more powerful slicing techniques; since then, many application areas have been identified, including program debugging, software maintenance, software reuse, and so on. See [14] for a fairly recent survey of the area.

Many studies have proposed the use of slicing for software testing. In the context of complex applications, which are by their very nature, size and architecture difficult to comprehend and test, slicing may be an invaluable help when a certification process has to be carried out.

On the other hand a strong demand for formal methods that help programmers developing correct programs has been present in software engineering for some time now. The *Design by Contract* (DbC) approach to software development [12] facilitates modular verification and certified code reuse, and has become a standard approach to the design of architecturally complex systems. The contract for a component (a procedure) can be regarded as a form of enriched software documentation that fully specifies the behavior of that component.

The development and broad adoption of annotation languages for the most popular programming languages reinforces the importance of using DbC principles in the development of programs. These include for instance the Java Modeling Language (JML) [4]; Spec# [2], a formal language for C# API contracts, and the SPARK [1] subset of Ada.

Traditional program slicing is based on control and data dependency analyses, but forms of slicing based on logical assertions have also been studied for over 10 years now, which combine slicing techniques with program verification, to identify synergies and take advantage of good practices on both sides. Comuzzi introduced the concept of *p-slice* [7], which is a slice calculated with respect to the validity of a given postcondition Q. The idea is that all program statements that are not required for the validity of Q upon termination are removed from the program (this only makes sense if Q holds as postcondition for the initial program). Canfora and colleagues used preconditions in their *conditioned slicing* technique [5] as a means to specify a set of initial states for computing a slice, resulting in a mixed form halfway between static and dynamic slicing. Preconditions and postconditions were combined by Chung and colleagues [6] to calculate what they called *specification-based slices*. Finally, *Fox et al* [9] introduced the *backward conditioning* technique, based on symbolic execution, to slice statements which, when executed, always lead to the negation of a given postcondition. The goal here is to use slicing as an aid in the verification of programs, in particular to find bugs.

In this paper we consider programs as sets of contract-annotated procedures, and study notions of assertion-based slicing for such programs with contracts. Specifically, we introduce the concept of *contract-based slice* of a program. Given any specification-based slicing algorithm (working at the level of commands), a contract-based slice can be calculated by slicing the code of each individual procedure independently with respect to its contract (which we call an *open* slice), or taking into consideration the calling contexts of each procedure inside a program (which we call a *closed* slice). We study both notions and then go on to introduce a more general notion of contract-based slice, which encompasses both open and closed slices as extreme cases. We remark that although the language used in this paper to illustrate our ideas is very simple, the principles and algorithms presented here scale up to realistic languages.

Structure of the Paper. Section 2 introduces a simple imperative language with annotated mutually recursive procedures, and sets up a verification conditions generator (VCGen) for that language. Section 3 then formalizes the notion of specification-based slice for the language. Sections 4 and 5 introduce contract-based slicing in their more specific (open and closed) and general forms respectively; section 6 then shows how a contract-based slicing algorithm (working at the inter-precedural level) can be synthesized from any desired specification-based slicing algorithm (working at the intra-precedural level). Section 7 illustrates our ideas through an example, and Section 8 concludes the paper.

2 Foundations: Verification Conditions and Specification-based Slicing

To illustrate our ideas we use a simple programming language. Its syntax is defined in Figure 1, where x and $\mathbf{p}$ range over sets of variables and procedure

$$
\begin{array}{lll}
\textbf{Exp}[\textbf{int}] & \ni e & ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x \mid x\tilde{} \mid \mathit{result} \mid -e \mid e + e \mid e - e \mid e * e \\
& & \mid \; e \,\texttt{div}\, e \mid e \,\texttt{mod}\, e \\
\textbf{Exp}[\textbf{bool}] & \ni b & ::= \texttt{true} \mid \texttt{false} \mid e == e \mid e < e \mid e <= e \mid e > e \mid e >= e \mid e \mathrel{!}= e \\
& & \mid \; b \mathbin{\&\&} b \mid b \parallel b \mid \mathrel{!} b \\
\textbf{Comm} & \ni C & ::= \textbf{skip} \mid C \,;\, C \mid x := e \mid \textbf{if}\ b\ \textbf{then}\ C\ \textbf{else}\ C \mid \textbf{while}\ b\ \textbf{do}\ \{A\}\, C \mid \textbf{call}\, \textbf{p} \\
\textbf{Assert} & \ni A & ::= \texttt{true} \mid \texttt{false} \mid e == e \mid e < e \mid e <= e \mid e > e \mid e >= e \mid e \mathrel{!}= e \\
& & \mid \; \mathrel{!} A \mid A \mathbin{\&\&} A \mid A \parallel A \mid A \to A \mid \texttt{Forall}\, x.\, A \mid \texttt{Exists}\, x.\, A \\
\textbf{Proc} & \ni \Phi & ::= \textbf{pre}\ A\ \textbf{post}\ A\ \textbf{proc}\ \textbf{p} = C
\end{array}
$$

C	$\mathsf{wp}(C, Q)$	$\mathsf{VC}(C, Q)$
skip	Q	$\emptyset$
$C_1 \,;\, C_2$	$\mathsf{wp}(C_1, \mathsf{wp}(C_2, Q))$	$\mathsf{VC}(C_1, \mathsf{wp}(C_2, Q))$ $\cup\ \mathsf{VC}(C_2, Q)$
$x := e$	$Q[e/x]$	$\emptyset$
if b **then** C_t **else** C_f	$(b \to \mathsf{wp}(C_t, Q)) \mathbin{\&\&} (\mathrel{!} b \to \mathsf{wp}(C_f, Q))$	$\mathsf{VC}(C_t, Q)\ \cup\ \mathsf{VC}(C_f, Q)$
while b **do** $\{I\}\, C$	I	$\{(I \mathbin{\&\&} b) \to \mathsf{wp}(C, I),$ $(I \mathbin{\&\&} \mathrel{!} b) \to Q\}$ $\cup\ \mathsf{VC}(C, I)$
call **p**	$\texttt{Forall}\, \overline{x_f}.$ $\big(\textbf{pre}(\textbf{p}) \to \textbf{post}(\textbf{p})\ [\overline{x}/\overline{x\tilde{}}, \overline{x_f}/\overline{x}]\big)$ $\to Q[\overline{x_f}/\overline{x}]$ with $\overline{x} = \mathcal{N}(\textbf{post}(\textbf{p})) \cup \mathcal{N}(Q)$	$\emptyset$

- The operator $\mathcal{N}(\cdot)$ returns a sequence of the variables occurring free in its argument assertion.
- Given a sequence of variables $\overline{x} = x_1, \ldots, x_n$, we let $\overline{x_f} = x_{1f}, \ldots, x_{nf}$ and $\overline{x\tilde{}} = x_1\tilde{}, \ldots, x_n\tilde{}$
- The expression $t[\overline{e}/\overline{x}]$, with $\overline{x} = x_1, \ldots, x_n$ and $\overline{e} = e_1, \ldots, e_n$ denotes the parallel substitution $t[e_1/x_1, \ldots, e_n/x_n]$

Fig. 1. Abstract syntax of programming language with annotations and VCGen rules

names respectively. A program is a non-empty set of mutually recursive procedure definitions that share a set of global variables (note that this is also an appropriate model for classes in an object-oriented language, whose methods operate on instance attributes). Operationally, an entry point would have to be defined for each such program, but that is not important for our current purpose. For the sake of simplicity we will consider only *parameterless procedures* that work exclusively on global variables, used for input and output, but the ideas presented here can be easily adapted to cope with parameters passed by value or reference, as well as return values.

Each procedure consists of a body of code, annotated with a precondition and a postcondition that form the procedure's specification, or *contract*. The body may additionally be annotated with loop invariants. Occurrences of variables in the precondition and postcondition of a procedure refer to their values in the pre-state and post-state of execution of the procedure respectively; furthermore the postcondition may use the syntax $x^{\sim}$ to refer to the value of variable x in the pre-state (this is inspired by the SPARK syntax). For each program variable x, $x^{\sim}$ is a special variable that can only occur in postconditions of procedures; the use of *auxiliary variables* (that occur in assertions only, not in code) is forbidden.

C-like syntax is used for expressions; the language of annotated assertions (invariants and contracts) extends boolean expressions with first-order quantification. Note that defining the syntax of assertions as a superset of boolean expressions is customary in specification languages based on contracts, as used by verification toolsets for realistic programming languages such as the SPARK toolset [1]. This clearly facilitates the task of programmers when annotating code with contracts.

A *program* is well-defined if all procedures adhere to the above principles, and moreover all defined procedure names are unique and the program is closed with respect to procedure invocation. We will write $\mathcal{P}(\Pi)$ for the set of names of procedures defined in Π. The operators $\mathbf{pre}(\cdot)$, $\mathbf{post}(\cdot)$, and $\mathbf{body}(\cdot)$ return a routine's precondition, postcondition, and body command, respectively, i.e. given the procedure definition $\mathbf{pre}\ P\ \mathbf{post}\ Q\ \mathbf{proc}\ \mathbf{p} = C$ with $\mathbf{p} \in \mathcal{P}(\Pi)$, one has $\mathbf{pre}_{\Pi}(\mathbf{p}) = P$, $\mathbf{post}_{\Pi}(\mathbf{p}) = Q$, and $\mathbf{body}_{\Pi}(\mathbf{p}) = C$. The program name will be omitted when clear from context.

We adopt the common assumptions of modern program verification systems based on the use of a *verification conditions generator* (VCGen for short) that reads in a piece of code together with a specification, and produces a set of first-order proof obligations (*verification conditions*) whose validity implies the partial correctness of the code with respect to its specification. Recall that given a command C and assertions P and Q, the *Hoare triple* $\{P\}\,C\,\{Q\}$ is valid if Q holds after execution of C terminates, starting from an initial state in which P is true [10]. A set of first-order conditions $\mathsf{Verif}(\{P\}\,C\,\{Q\})$ whose validity is sufficient for this is given as

$$\mathsf{Verif}(\{P\}\,C\,\{Q\}) = \{P \rightarrow \mathsf{wp}(C, Q)\} \quad \cup \quad \mathsf{VC}(C, Q)$$

Where the functions $\mathsf{wp}(\cdot,\cdot)$ and $\mathsf{VC}(\cdot,\cdot)$ are defined in Figure 1. The first condition states that P is stronger than the weakest precondition that grants the

validity of postcondition Q, and the remaining verification conditions ensure the adequacy of certain preconditions. For instance, the precondition of a loop command can only be considered to be equal to the annotated invariant if this is indeed an invariant (whose preservation is ensured by a verification condition) and moreover it is sufficient to establish the truth of the required postcondition upon termination of the loop.

Definition 1 (Verif. Conditions of a Program) *For a program Π consisting of the set of procedures $\mathcal{P}(\Pi)$, the set of verification conditions* $\mathsf{Verif}(\Pi)$ *is*

$$\mathsf{Verif}(\Pi) \;=\; \bigcup_{\mathbf{p}\in\mathcal{P}(\Pi)} \mathsf{Verif}(\{\mathbf{pre}(\mathbf{p}) \;\&\&\; \overline{x} == \overline{x^{\sim}}\}\, \mathbf{body}(\mathbf{p})\, \{\mathbf{post}(\mathbf{p})\})$$

Note $\mathbf{pre}(\mathbf{p})$ is strengthened to allow for the use of the $^{\sim}$ notation in postconditions. Let $\models A$ denote the validity of assertion A, and $\models S$, with S a set of assertions, the validity of all $A \in S$. This VCGen algorithm can be shown to be *sound* with respect to an operational semantics for the language, i.e. if $\models \mathsf{Verif}(\Pi)$ then for every $\mathbf{p} \in \mathcal{P}(\Pi)$ the triple $\{\mathbf{pre}(\mathbf{p})\}\, \mathbf{call}\, \mathbf{p}\, \{\mathbf{post}(\mathbf{p})\}$ is valid.

The intra-procedural (command-level) aspects of the VCGen are standard, but the inter-procedural aspects (program-level) are less well-known. We make the following remarks.

- Although this is somewhat hidden (unlike in the underlying program logic), the soundness of the VCGen is based on a *mutual recursion* principle, i.e. the proof of correctness of each routine assumes the correctness of all the routines in the program, including itself. If all verification conditions are valid, correctness is established simultaneously for the entire set of procedures in the program. This is the fundamental principle behind *design-by-contract*.
- The weakest precondition rule for procedure call takes care of what is usually known as the *adaptation* between the procedure's contract and the postcondition required in the present context. The difficulty of reasoning about procedure calls has to do with the need to refer, in a contract's postcondition, to the values that some variables had in the pre-state. We adapt to our context the rule proposed by Kleymann [11] as an extension to Hoare logic, and refer the reader to that paper for a historical discussion of approaches to adaptation.

Lemma 1. *Let $\models Q_1 \rightarrow Q_2$ with Q_1, Q_2 any two assertions. Then $\models \mathsf{wp}(C, Q_1) \rightarrow \mathsf{wp}(C, Q_2)$ and moreover $\models \mathsf{VC}(C, Q_1)$ implies $\models \mathsf{VC}(C, Q_2)$.*

3 Specification-based Slicing

This section reviews the basic notions of specification-based slicing at the command level.

Informally, a command C' is a *specification-based slice* of C if it is a *portion* of C (a syntactic notion) and moreover C can be *refined* to C' with respect to a given specification (a semantic notion). We now give the formal definitions.

$$\frac{}{\textbf{skip} \preceq C} \qquad \frac{C_1 \preceq C_2}{C_1 \,;\, C \preceq C_2 \,;\, C} \qquad \frac{C_1 \preceq C_2}{C \,;\, C_1 \preceq C \,;\, C_2}$$

$$\frac{C_1 \preceq C_2}{\textbf{if } b \textbf{ then } C_1 \textbf{ else } C \preceq \textbf{if } b \textbf{ then } C_2 \textbf{ else } C} \qquad \frac{C_1 \preceq C_2}{\textbf{if } b \textbf{ then } C \textbf{ else } C_1 \preceq \textbf{if } b \textbf{ then } C \textbf{ else } C_2}$$

$$\frac{C_1 \preceq C_2}{\textbf{while } b \textbf{ do } \{I\}\, C_1 \preceq \textbf{while } b \textbf{ do } \{I\}\, C_2}$$

Fig. 2. Definition of portion-of relation

Definition 2 (Portion-of relation) *The $\cdot \preceq \cdot$ relation is the transitive and reflexive closure of the binary relation generated by the set of rules given in Figure 2.*

Definition 3 (Specification-based slice [6]) *Let C, C' be commands and P, Q assertions such that $\models \mathsf{Verif}(\{P\}\, C\, \{Q\})$ holds, thus C is correct with respect to the specification (P, Q). C' is a slice of C with respect to (P, Q), written $C' \lhd_{(P,Q)} C$, if $C' \preceq C$ and $\models \mathsf{Verif}(\{P\}\, C'\, \{Q\})$.*

A *specification-based slicing algorithm* is any function slice that takes a command and a specification, and produces a slice of the command w.r.t. the specification, i.e. $\mathsf{slice}(C, P, Q) \lhd_{(P,Q)} C$.

Given program C correct with respect to the specification (P, Q), one wants to be able to identify portions of C that are still correct w.r.t (P, Q), i.e., portions in which some irrelevant statements (in the sense that they are not required for the program to be correct) are removed. Naturally, many such slices may exist, as well as methods for calculating them. These methods differ with respect to *efficacy* (being able to precisely identify all removable commands and effectively removing the highest possible number of such commands) and *efficiency* (how the execution time of the slicing algorithm varies with the number of lines of code). In [3] we explain the issues involved in detail, survey the existing algorithms, and propose improvements over those algorithms, concerning both precision and efficiency.

We remark that in practice one would not want to slice a block of code C with respect to its own specification (P, Q), unless maybe to confirm that it does not contain unnecessary code; but consider the situation in which one is asked to fulfill a weaker specification (P', Q'), i.e. $\models P' \rightarrow P$ and $\models Q \rightarrow Q'$. Then the code C can be reused, since it is necessarily also correct with respect to (P', Q'), but it may contain code that, while being necessary to satisfy (P, Q), is irrelevant with respect to satisfying (P', Q'). Thus in such a *specialization reuse* context,

it makes sense to slice C with respect to the new specification to eliminate the unnecessary code.

A Specification-based Slicing Algorithm

We have designed a specification-based slicing algorithm that improves on previous algorithms with respect to the number of statements removed from the programs. In fact we show in [3] that this algorithm produces the smallest slice of a program relative to a given specification (modulo an oracle for first-order proof obligations). The algorithm works on a *labelled control-flow graph*, whose edges are labelled with a pair of assertions corresponding to the strongest postcondition (calculated by propagating the specified precondition forward) and the weakest precondition (calculated by propagating the specified postcondition backward) at the program point represented by that edge. The graph is then extended by adding additional edges corresponding to subprograms S such that the strongest postcondition at the point immediately before S is stronger than the weakest precondition immediately after S (this implicative formula can be checked by an external proof tool). The resulting *slice graph* contains as subgraphs representations (in the form of labelled CFGs) of all the specification-based slices of the initial program with respect to its specification, and the smallest such slice can be calculated using standard graph algorithms.

Figure 3 shows an example slice graph for a program. Sliceable sequences are signaled by edges (and possibly **skip** nodes) added to the initial labeled CFG (shown as thick lines). Our online laboratory [8] implements this algorithm. The laboratory also allows for the visualization of these labelled control-flow graphs, which are useful as an aid in debugging, when the verification of a program fails.

4 Open / Closed Contract-based Slicing

How can specification-based slicing be applied in the context of a multi-procedure program? Since a program is a set of procedures carrying their own contract specifications, it makes sense to investigate how the contract information can be used to produce useful slices at the level of individual procedures, and globally at the level of programs. A natural approach consists in simply slicing each procedure based on its own contract information. The idea is to eliminate all spurious code that may be present and does not contribute to that procedure's contract.

Definition 4 (Open Contract-based Slice) *Given programs Π, Π' such that $\models \mathsf{Verif}(\Pi)$ and $\mathcal{P}(\Pi) = \mathcal{P}(\Pi')$, we say that Π' is an open contract-based slice of Π, written $\Pi' \lhd_o \Pi$, if for every procedure $\mathbf{p} \in \mathcal{P}(\Pi)$ the following holds:* $\mathbf{pre}_{\Pi'}(\mathbf{p}) = \mathbf{pre}_{\Pi}(\mathbf{p})$*;* $\mathbf{post}_{\Pi'}(\mathbf{p}) = \mathbf{post}_{\Pi}(\mathbf{p})$*; and*

$$\mathbf{body}_{\Pi'}(\mathbf{p}) \lhd_{(\mathbf{pre}(\mathbf{p}) \,\&\&\, \overline{x}==\overline{x^{\sim}}, \mathbf{post}(\mathbf{p}))} \mathbf{body}_{\Pi}(\mathbf{p})$$

i.e. the body of each routine in Π' is a specification-based slice (with respect to its own annotated contract) of that routine in Π.

```
1 if (y > 0) then x := 100; x := x+50; x := x−100
2           else x := x−150; x := x+100; x := x+100
```

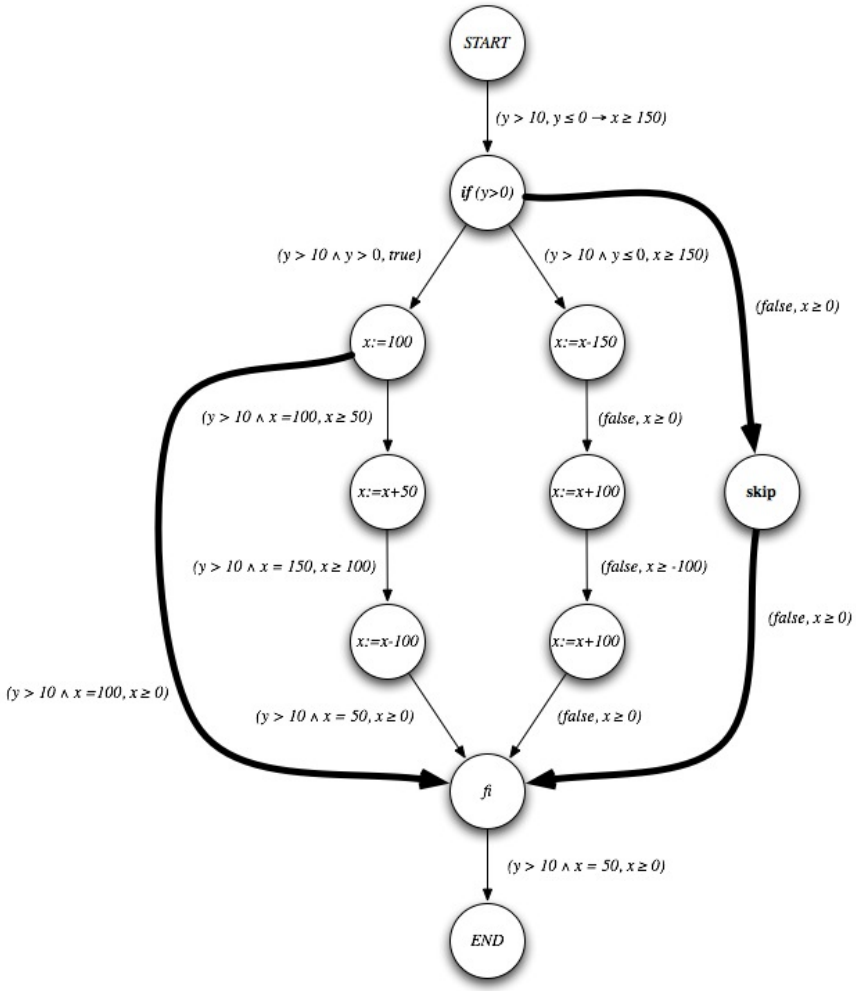

Fig. 3. Example program and its slice graph w.r.t. specification $(y > 10, x \geq 0)$

As expected, open contract-based slicing produces correct programs:

Proposition 1 *If* $\models \mathsf{Verif}(\Pi)$ *and* $\Pi' \lhd_o \Pi$ *then* $\models \mathsf{Verif}(\Pi')$.

Given some specification-based slicing algorithm $\mathsf{slice}(\cdot,\cdot,\cdot)$, program Π, and procedure $\mathbf{p} \in \mathcal{P}(\Pi)$, it is straightforward to lift it to an algorithm that calculates contract-based slices. Let

$$\begin{aligned}\mathsf{procslice_o}(\mathbf{p}) \doteq\ & \mathbf{pre}\ (\mathbf{pre}_\Pi(\mathbf{p}))\\ & \mathbf{post}\ (\mathbf{post}_\Pi(\mathbf{p})\\ & \mathbf{proc}\ \mathbf{p} = \mathsf{slice}(\mathbf{body}_\Pi(\mathbf{p}), \mathbf{pre}_\Pi(\mathbf{p}), \mathbf{post}_\Pi(\mathbf{p}))\end{aligned}$$

Then $\mathsf{progslice_o}(\Pi) \doteq \{\ \mathsf{procslice_o}(\mathbf{p}) \mid \mathbf{p} \in \mathcal{P}(\Pi)\ \}$.

Proposition 2 *For any program* Π *such that* $\models \mathsf{Verif}(\Pi)$, $\mathsf{progslice_o}(\Pi) \lhd_o \Pi$.

As was the case with specification-based slicing, one may want to calculate open contract-based slices just to make sure that a program (already proved correct) does not contain irrelevant code. This notion of slice of a program assumes that all the procedures are public and may be externally invoked – the program is *open*. But consider now the opposite case of a program whose procedures are only invoked by other procedures in the same program, which we thus call a *closed* program (this makes even more sense if one substitutes *class* for *program* and *method* for *procedure* in this reasoning). In this situation, since the set of callers of each procedure is known in advance (it is a subset of the procedures in the program), it is possible that *weakening* the procedures' contracts may still result in a correct program, as long as the assumptions required by each procedure call are all still respected. In other words the procedures may be doing more work than is actually required. Such a program may then be sliced in a more aggressive way, defined as follows.

Definition 5 (Closed Contract-based Slice) *Let* Π, Π' *be programs such that* $\models \mathsf{Verif}(\Pi)$ *and* $\mathcal{P}(\Pi) = \mathcal{P}(\Pi')$. Π' *is a* closed contract-based slice *of* Π, *written* $\Pi' \lhd_c \Pi$, *if* $\models \mathsf{Verif}(\Pi')$ *and additionally for every procedure* $\mathbf{p} \in \mathcal{P}(\Pi)$

1. $\models \mathbf{pre}_{\Pi'}(\mathbf{p}) \rightarrow \mathbf{pre}_\Pi(\mathbf{p})$;
2. $\models \mathbf{post}_\Pi(\mathbf{p}) \rightarrow \mathbf{post}_{\Pi'}(\mathbf{p})$; *and*
3. $\mathbf{body}_{\Pi'}(\mathbf{p}) \lhd_{(\mathbf{pre}_{\Pi'}(\mathbf{p})\ \&\&\ \overline{x}==\overline{x^\sim}, \mathbf{post}_{\Pi'}(\mathbf{p}))}\ \mathbf{body}_\Pi(\mathbf{p})$

Note that in general weakening the contracts of some procedures in a correct program may result in an incorrect program, since the correctness of each procedure may depend on the assumed correctness of other procedures; thus the required condition $\models \mathsf{Verif}(\Pi')$ in the definition of closed contract-based slice.

5 Contract-based Slicing: General Case

Clearly the notion of closed contract-based slicing admits trivial solutions: since all contracts can be weakened, any precondition (resp. postcondition) can be set

to *false* (resp. *true*), and thus any procedure body can be sliced to **skip**. A more interesting and realistic notion is obtained by fixing a subset of procedures of the program, whose contracts must be preserved. All other contracts may be weakened as long as the resulting program is still correct.

Definition 6 (Contract-based Slice) *Let Π, Π' be programs such that $\mathcal{P}(\Pi) = \mathcal{P}(\Pi')$ and $\mathcal{S} \subseteq \mathcal{P}(\Pi)$; Π' is a* contract-based slice *of Π, written $\Pi' \lhd_{\mathcal{S}} \Pi$, if the following all hold:*

1. $\models \mathsf{Verif}(\Pi')$.
2. *for every procedure* $\mathbf{p} \in \mathcal{S}$,
   - $\mathbf{pre}_{\Pi'}(\mathbf{p}) = \mathbf{pre}_{\Pi}(\mathbf{p})$ *and* $\mathbf{post}_{\Pi'}(\mathbf{p}) = \mathbf{post}_{\Pi}(\mathbf{p})$;
   - $\mathbf{body}_{\Pi'}(\mathbf{p}) \lhd_{(\mathbf{pre}(\mathbf{p}) \,\&\&\, \overline{x}==\overline{x^{\sim}}, \mathbf{post}(\mathbf{p}))} \mathbf{body}_{\Pi}(\mathbf{p})$
3. *for every procedure* $\mathbf{p} \in \mathcal{P}(\Pi) \setminus \mathcal{S}$,
   - $\models \mathbf{pre}_{\Pi'}(\mathbf{p}) \rightarrow \mathbf{pre}_{\Pi}(\mathbf{p})$;
   - $\models \mathbf{post}_{\Pi}(\mathbf{p}) \rightarrow \mathbf{post}_{\Pi'}(\mathbf{p})$; *and*
   - $\mathbf{body}_{\Pi'}(\mathbf{p}) \lhd_{(\mathbf{pre}_{\Pi'}(\mathbf{p}) \,\&\&\, \overline{x}==\overline{x^{\sim}}, \mathbf{post}_{\Pi'}(\mathbf{p}))} \mathbf{body}_{\Pi}(\mathbf{p})$

This notion is very adequate to model slicing when applied to code reuse. When program (or class) Π is reused, some of its procedures may not need to be public, since they will not be externally invoked (but they may be invoked by other, public procedures in the program). In this case the contracts of the private procedures may be weakened according to their usage inside the program, i.e. the actual required specification for each private procedure may be calculated from the set of internal calls, since it is guaranteed that no external calls will be made to private procedures. One may then want to reflect this in the annotated contracts, in order to produce a contract-based slice stripped of the redundant code. Private procedures whose contracts are not required internally may indeed see their bodies sliced to **skip**.

Note that even for closed programs this notion makes more sense. Since its procedures are not invoked externally from other programs' procedures, every closed program will naturally have a main procedure to be executed as an entry point, whose contract is fixed (cannot be weakened).

Finally, it is easy to see that both open and closed contract-based slicing are special cases of Definiton 6: $\Pi' \lhd_o \Pi \Leftrightarrow \Pi' \lhd_{\mathcal{P}(\Pi)} \Pi$ and $\Pi' \lhd_c \Pi \Leftrightarrow \Pi' \lhd_{\emptyset} \Pi$.

6 A Contract-based Slicing Algorithm

Again any specification-based slicing algorithm (at the command level) can be used for calculating contract-based slices according to Definition 6. A contract-based slice of program Π can be calculated by analyzing Π in order to obtain information about the actual preconditions and postconditions that are required of each procedure call, and merging this information together. Specifically, we may calculate for each procedure the disjunction of all required preconditions and the conjunction of all required postconditions; this potentially results in a

weaker contract with respect to the annotated contract of the procedure, which can thus be used to slice that procedure.

To implement this idea for a given program Π we consider a preconditions table T_0 that associates to each procedure $\mathbf{p} \in \mathcal{P}(\Pi)$ a precondition, initialized with $T_0[\mathbf{p}] = \texttt{false}$, and a postconditions table T that associates to each procedure $\mathbf{p} \in \mathcal{P}(\Pi)$ a postcondition, initialized with $T[\mathbf{p}] = \texttt{true}$. The algorithm executes the following steps to produce $\mathsf{progslice}_{\mathsf{o}}(\Pi)$.

1. Calculate $\mathsf{Verif}(\Pi)$ and while doing so, for every invocation of the form $\mathsf{wp}(\mathbf{call}\,\mathbf{p}, Q)$ set $T[\mathbf{p}] := T[\mathbf{p}]\ \&\&\ Q$.
2. An alternative set of verification conditions based on *strongest postconditions* can be defined by rules that are fairly symmetric to those given in Figure 1 (omitted here). We calculate this set, and while doing so, for every invocation of the form $\mathsf{sp}(\mathbf{call}\,\mathbf{p}, P)$ set $T_0[\mathbf{p}] := T_0[\mathbf{p}] \parallel P$.
3. For $\mathbf{p} \in \mathcal{P}(\Pi)\backslash\mathcal{S}$ let

$$\begin{aligned}\mathsf{procslice}_{\mathcal{S}}(\mathbf{p}) \doteq\ & \mathbf{pre}\ T_0[\mathbf{p}] \\ & \mathbf{post}\ T[\mathbf{p}] \\ & \mathbf{proc}\ \mathbf{p} = \mathsf{slice}(\mathbf{body}(\mathbf{p}), T_0[\mathbf{p}], T[\mathbf{p}])\end{aligned}$$

4. Then $\mathsf{progslice}_{\mathcal{S}}(\Pi) = \{\ \mathsf{procslice}_{\mathsf{o}}(\mathbf{p}) \mid \mathbf{p} \in \mathcal{S}\ \} \cup \{\mathsf{procslice}_{\mathcal{S}}(\mathbf{p}) \mid \mathbf{p} \in \mathcal{P}(\Pi)\backslash\mathcal{S}\}$

Proposition 3 *For any program Π such that $\models \mathsf{Verif}(\Pi)$, $\mathsf{progslice}_{\mathcal{S}}(\Pi) \lhd_{\mathcal{S}} \Pi$.*

Note that step 1 (or 2) can be skipped, in which case $T[\mathbf{p}]$ (resp. $T_0[\mathbf{p}]$) should be set to $\mathbf{post}(\mathbf{p})$ (resp. $\mathbf{pre}(\mathbf{p})$), and slicing will be less aggressive, based on preconditions or postconditions only. Section 7 illustrates the application of this contract-based slicing algorithm (using only postconditions, i.e. with step 2 of the algorithm skipped) to a program containing a procedure $\mathbf{p}$ that calculates a number of different outputs given an input array; this program is being reused in a context in which no external calls are made to $\mathbf{p}$, and two internal procedures do call $\mathbf{p}$, but some of the information computed by $\mathbf{p}$ is not required by either of the calls. Then maybe some statements of $\mathbf{p}$ can be sliced off.

7 An Illustrative Example

In this section we intend to illustrate the concept of *contract-based slice* through an example. We have implemented a prototype program slicer[1] for a subset of Spec#, which includes many different specification-based slicing algorithms, and which we have used to calculate the slices shown below. Note that in an object-oriented language if we substitute the notions of class, method, and instance attribute for those of program, procedure, and global variables of our previous imperative setting, the ideas introduced earlier essentially apply without modifications.

[1] Available through a web-based interface from
`http://gamaepl.di.uminho.pt/gamaslicer`

```
1   int summ, sumEven, productum, maximum, minimum;
2   public int[]! a = new int[100];
3   public int length = 100;
4
5   public void OpersArrays()
6     ensures summ == sum{int i in (0: length); a[i]};
7     ensures sumEven ==
8         sum{int i in (0: length); (((a[i] % 2)== 0)? a[i]:0)};
9     ensures productum == product{int i in (0:length); a[i]};
10    ensures maximum == max{int i in (0:length); a[i]};
11    ensures minimum == min{int i in (0:length); a[i]};
12  {
13    summ = 0; sumEven = 0; productum = 0;
14    maximum = Int32.MinValue; minimum = Int32.MaxValue;
15    for (int n = 0; n < length; n++)
16      invariant n <= length;
17      invariant summ == sum{int i in (0: n); a[i]};
18      invariant sumEven ==
19          sum{int i in (0: n); (((a[i] % 2)== 0)? a[i]:0)};
20      invariant productum == product{int i in (0: n); a[i]};
21      invariant maximum == max{int i in (0: n); a[i]};
22      invariant minimum == min{int i in (0: n); a[i]};
23    {
24      summ += a[n];
25      productum *= a[n];
26      if ((a[n] % 2) == 0) { sumEven += a[n]; }
27      if(a[n] > maximum) { maximum = a[n]; }
28      if(a[n] < minimum) { minimum = a[n]; }
29    }
30  }
```

Listing 1.1. Annotated method `OpersArrays`

Listings 1.1 and 1.2 contain extracts from a class Π containing an annotated method, called `OpersArrays`, which computes several array operations: the sum of all elements, the sum of all the even elements, the iterated product, and the maximum and minimum. Π contains two other methods `Average` and `Multiply`; the former computes the average of the elements belonging to the array, and the latter multiplies the minimum of the array by a parameter `y`. The code contains appropriate contracts for all methods, as well as loop invariants. All 3 methods are correct with respect to their contracts – we assume this has been established beforehand.

Moreover suppose that `OpersArrays` is a private method, i.e. in a given context it is known that it will not be invoked externally. As can be observed in Listing 1.3, the method `Average` calls the method `OpersArrays` and then uses the calculated value of the variable `summ`, and `Multiply` calls `OpersArrays` and then uses the value of variable `minimum`. Then it makes sense to calculate

```
1  public double Average()
2    ensures result == (sum{int i in (0: length);a[i]})/length;
3   {
4      double average;
5      OpersArrays();
6      average = summ / length;
7      return average;
8   }
9
10 public int Multiply(int y)
11   requires y >= 0;
12   ensures result == (min{int i in (0: length); a[i]})*y;
13  {
14     OpersArrays();
15     int x = minimum, q = 0, i =0;
16     while(i <y )
17         invariant 0 <= i <= y;
18         invariant q == (i * x);
19     {
20         q = q + x;
21         i= i+1;
22     }
23     return q;
24  }
```

Listing 1.2. Annotated methods `Average` and `Multiply`

a contract-based slice of this class with $\mathcal{S} = \{\texttt{Average}, \texttt{Multiply}\}$, which will result in the method `OpersArrays` being stripped of the irrelevant code.

In order to calculate $\mathsf{progslice}_{\mathcal{S}}(\Pi)$, T is first initialized as

$$T[\texttt{OpersArrays}] := \texttt{true}$$

After performing the first step of the algorithm we set

$$T[\texttt{OpersArrays}] := \texttt{true} \;\&\&\; Q_1 \;\&\&\; Q_2$$

where (calculations omitted)

$$Q_1 = \frac{summ}{length} == \frac{sum\{int\ i\ in(0 : length); a[i]\}}{length}$$

$$Q_2 = 0 \leq i \leq y \;\&\&\; q == i \times minimum$$

So, as the component `OpersArrays` is called twice in the context of the two previous components, the result is the weaker postcondition $\texttt{true} \;\&\&\; Q_1 \;\&\&\; Q_2$. The final step in the calculation of the slice using table T gives us

$$\mathsf{progslice}_{\mathsf{c}}(\Pi) \doteq \{\mathsf{procslice}_{\mathsf{o}}(\texttt{Average}), \mathsf{procslice}_{\mathsf{o}}(\texttt{Multiply}), \mathsf{procslice}_{\mathcal{S}}(\texttt{OpersArrays})\}$$

```
1   public void OpersArrays()
2     ensures summ == sum{int i in (0: length); a[i]};
3     ensures minimum == min{int i in (0:length); a[i]};
4   {
5     summ = 0;
6     minimum = Int32.MaxValue;
7     for (int n = 0; n < length; n++)
8       invariant n <= length;
9       invariant summ == sum{int i in (0: n); a[i]};
10      invariant minimum == min{int i in (0: n); a[i]};
11    {
12      summ += a[n];
13      if(a[n] < minimum) { minimum = a[n]; }
14    }
15  }
```

Listing 1.3. Sliced method `OpersArrays`

Calculating $\mathsf{slice}(\mathbf{body}(\texttt{OpersArrays}), \mathbf{pre}(\texttt{OpersArrays}), T[\texttt{OpersArrays}])$ results in cutting the statements present in lines 14, 27, 28 and 29 (after removing from the invariant the predicates in lines 20–23, which contain only occurrences of variables that do not occur in $T[\texttt{OpersArrays}]$). The final sliced method is shown below. The other two methods remain unchanged.

8 Conclusion

We introduced notions of slicing for programs developed according to design-by-contract principles. The motivation was to bring to the inter-procedural level the power of specification-based slicing, which we believe has great application potential and will surely become more popular in coming years, profiting from advances in verification and automated proof technology. We believe that the ideas proposed can be useful in the traditional fields of application of slicing, such as source code analysis (to help in debugging or program comprehension); program maintenance (when more precision is required); certification of programs constructed by reusing annotated components; and the specialization of programs composed of fully annotated and certified procedures.

This work sets up a theoretical framework for slicing multi-procedure, contract-annotated programs, and is part of a bigger effort that includes a fundamental investigation of command-level (intra-procedural) specification-based slicing algorithms [3], as well as the development of an online laboratory for assertion-based slicing [8] that interacts with external automatic theorem provers for discharging slicing conditions. Our approach can be applied to program comprehension (as it makes easier to understand the process of slicing a program with contracts) and program reuse (it can be seen as a lever to integrate annotated components with other programs in an easier manner).

As future work we intend to continue with the development of GamaSlicer in order to perform tests with more examples as well as to confirm that our approach scales up.

Acknowledgment. This work was partially supported by projects RESCUE (FCT contract PTDC / EIA / 65862 / 2006) and CROSS (FCT contract PTDC / EIA-CCO / 108995 / 2008).

References

1. John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, first edition, March 2003.
2. Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS : construction and analysis of safe, secure, and interoperable smart devices*, volume 3362, pages 49–69. Springer, Berlin, ALLEMAGNE, March 2004.
3. J. Barros, D. da Cruz, P. R. Henriques, and J. S. Pinto. Assertion-based Slicing and Slice Graphs. In J. L. Fiadeiro and S. Gnesi, editors, *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM'10)*, 2010.
4. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of jml tools and applications, 2003.
5. Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–608, November 1998. Special issue on program slicing.
6. I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 605–609, New York, NY, USA, 2001. ACM.
7. Joseph J. Comuzzi and Johnson M. Hart. Program slicing using weakest preconditions. In *FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, pages 557–575, London, UK, 1996. Springer-Verlag.
8. Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Gamaslicer: an Online Laboratory for Program Verification and Analysis. In *proceedings of the 10th. Workshop on Language Descriptions Tools and Applications (LDTA'10)*, 2010.
9. Chris Fox, Sebastian Danicic, Mark Harman, and Robert M. Hierons. Backward conditioning: A new program specialisation technique and its application to program comprehension. In *IWPC*, pages 89–97. IEEE Computer Society, 2001.
10. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
11. Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.
12. Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
13. Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
14. Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.