Abstract Program Slicing: From Theory towards an Implementation

Isabella Mastroeni and Đurica Nikolić

Dipartimento di Informatica, Università di Verona, Italy
isabella.mastroeni@gmail.com, durica.nikolic@univr.it

Abstract. In this paper we extend the formal framework proposed by Binkley et al. for representing and comparing forms of program slicing. This framework describes many well-known forms of slicing in a unique formal structure based on (abstract) projections of state trajectories. We use this formal framework for defining a new technique of slicing, called abstract slicing, which aims to slice programs with respect to properties of variables. In this way we are able to extend the original work with three forms of abstract slicing, static, dynamic and conditioned, we show that all existing forms are instantiations of their corresponding abstract forms and we enrich the existing slicing technique hierarchy by inserting these abstract forms of slicing. Furthermore, we provide an algorithmic approach for extracting abstract slices. The algorithm is split into two modules: the simple approach, used for abstract static slicing, and the extended approach, composed of several applications of the simple one, which is used for abstract conditioned slicing.

Keywords: Program Slicing, Semantics, Program Analysis, Abstract Interpretation.

1 Introduction

It is well-known that as the size of programs increases, it becomes impractical to maintain them as monolithic structures. Indeed, splitting programs in smaller pieces allows to construct, understand and maintain large programs much more easily. *Program slicing* [3,7,14,15], is a program manipulation technique that extracts from programs statements which are relevant to a particular computation. In particular, a *program slice* is an executable program whose behavior must be identical to a specific subset of the original program's behavior. The specification of this behavior subset is called *slicing criterion* and can be expressed as the value of some sets of variables at some set of statements and/or program points [15]. Slicing[1] can be used in debugging [15], software maintenance [9], comprehension [4,8], re-engineering [5], etc.

Since the seminal paper introducing slicing [15], there have been many works proposing several notions of slicing, and different algorithms to compute slices (see [7,14] for good surveys about the existing slicing techniques). Program slicing is a transformation technique that reduces the size of programs to analyze.

[1] We use *slicing* (*slice*) and *program slicing* (*program slice*) as interchangeable terms.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 452–467, 2010.

Nevertheless, the reduction obtained by means of standard slicing techniques may be not sufficient for simplifying program analyses since it may keep more statements than those strictly necessary for the desired analysis. Suppose we are analyzing a program, and suppose we want a variable x to have a particular property φ. If we realize that, at a fixed program point, x does not have that property, we may want to understand which statements affect the computation of property φ of x, in order to find out more easily where the computation was wrong. In this case we are not interested in the exact value of x, hence we may not need *all* the statements that a standard slicing algorithm would return. In this situation we would need a technique that returns the minimal amount of statements that actually affect the computation of a desired property of x.

In this paper we introduce a novel notion of slicing, called *abstract slicing*, that looks for the statements affecting a fixed property (modelled in the context of abstract interpretation [6]) of variables of interest.

Example 1. Consider the program P in Fig. 1. If we are interested in exact values of variable d at the end of execution, program Q can be a *slice* of P with respect to that criterion. But, if we are interested in the parity of d at that point, the situation is a little bit different. The parity of d in $d = 2 * c + b + a - a$ depends on variable b only. Therefore, program R may be an *abstract slice* of P w.r.t. the specified criterion. Even in this simple case, the *abstract slice* gives us more precise information about the statements affecting the property of interest.

1 a := 1;	1 a := 1;	1
2 b := b + 1;	2 b := b + 1;	2 b := b + 1;
3 c := c + 2;	3 c := c + 2;	3
4 e := a + b + c;	4	4
5 d := 2∗c + b + a − a;	5 d := 2∗c + b + a − a;	5 d := 2∗c + b + a − a;
Program P	Program Q	Program R

Fig. 1. Q and R are *slice* and *abstract slice* of P

Moreover, we have taken the first steps towards an implementation of this new form of slicing and we propose two approaches that, under certain hypotheses, extract *abstract slices*. We provide an example illustrating the application of the simplest approach and which highlights some of the differences between them.
This is not the first attempt to define weakened forms of slicing, often called abstract, even if it has never been formally described. Several authors focused on the concept of abstract dependency and tried to define it formally. Mastroeni and Zanardini [13] defined abstract slicing in terms of abstract dependency, a notion of dependency parametric on properties of interest, and obtained as negation of the notion of (abstract) non-interference [10]. The difference lies on the fact that while we provide a general formal definition, which allows several forms of abstract slicing simply by instantiating parameters, in [13] the authors consider only standard static slicing, defining abstract slicing by means of abstract dependencies. The notion of abstract slicing introduced by Hong, Lee and Sokolsky [11]

represents an implementation of a technique of conditioned slicing [4] based on abstract interpretation and model checking. Therefore, instead of weakening the observation of all executions, the authors only consider a subset of all possible executions.

2 Program Slicing

In this section we introduce different notions of program slicing and the Binkley et al. framework [1,2] in a slightly revised way by *constructing* a unified notation for slicing criteria.

Let us introduce some basic notions. We use $\mathbb{L}$ to denote the set of line numbers and $\mathbb{V}$ to denote a set of values. *Var* denotes a set of memory locations. A memory state is a function $\rho \in \mathbb{M}$, where $\mathbb{M} \stackrel{\text{def}}{=} Var \to \mathbb{V}$. When a program is in a state $\sigma_i = \langle n_i, k_i, \rho_i \rangle \in \mathbb{S} \stackrel{\text{def}}{=} \mathbb{L} \times \mathbb{N} \times \mathbb{M}$, it means that it is in a memory state ρ_i, the next statement to be executed is at a line number n_i and k_i is the number of occurrences of n_i until that point. A state trajectory $\sigma = (n_1, k_1, \rho_1) \dots (n_l, k_l, \rho_l) \in \Sigma \stackrel{\text{def}}{=} \mathbb{S}^*$ is a sequence of states $\sigma_i \in \mathbb{S}$, $i \in [1..l]$ a program goes through during the execution. The state trajectory obtained by executing program P from input state ρ is denoted T_P^ρ.

The very first definition of slicing was given by Weiser [15]. Nowadays, that technique is known as static slicing. It states that, if we execute both the original program and its static slice from the same input, any time the point of interest is reached in the original program, it is reached in the slice as well, and the values of all variables of interest in both programs are equal.

A key notion in program slicing is the *slicing criterion*, let us define a general characterization able to model all the forms of slicing we are going to introduce. As far as static slicing is concerned, the slicing criterion simply specifies the set $V \subseteq Var$ of variables of interest, and the program point $n \in \mathbb{L}$ of interest, namely the criterion is $\mathcal{C} = (V, n)$. Korel and Laski proposed a new techinque of slicing, called *dynamic slicing* [12]. It considers only one particular input, and the dynamic slice preserves the meaning of the original program for that input only. Let's consider two programs P and Q. I_P and I_Q denote sequences of line numbers reached during the executions of P and Q from ρ. The occurrence of interest is the n-th element of I_P. Q is a dynamic slice of P if there is an execution position n' in I_Q such that: 1) If we consider only first n elements of I_P, and if we eliminate all of them not appearing in Q, we obtain first n' elements of I_Q; 2) For all variables $v \in V$ the value of v in P before executing statement $I_P(n)$ is equal to the value of v in Q before executing statement $I_Q(n')$ and 3) Statements $I_P(n)$ and $I_Q(n')$ are equal. In order to characterize the slicing criterion also for dynamic slicing we have to enrich the notion and to consider a set of initial memories $\mathcal{I} \subseteq \mathbb{M}$. Hence the criterion is now $\mathcal{C} = (\mathcal{I}, V, n)$, where $\mathcal{I} = \mathbb{M}$ for static slicing, while $\mathcal{I} = \{\rho\}$, with $\rho \in \mathbb{M}$ for dynamic slicing.

Finally, Canfora et al. proposed a new techinque of slicing called *conditioned slicing* [4], which requires that a conditioned slice preserves the meaning of the original program for a set of inputs satisfying one particular condition π. Let us

denote $\mathbb{M}_{in}$ the set of input states satisfying π [1], then the slicing criterion is still $\mathcal{C} = (\mathcal{I}, V, n)$ where $\mathcal{I} = \mathbb{M}_{in}$ characterizes conditioned slicing.

Each type of slicing can have four forms: *standard* form - it considers one ore more points in a program with respect to a set of variables; *Korel and Laski* form (KL) - a stronger form where the program and the slice must follow identical paths; *iteration count* form (IC) - requires that a program and its slice need only agree at a particular iteration of a program point; and *Korel and Laski iteration count* form (KLi) - requires that a program and its slice must follow identical paths and need only agree at a particular iteration of a point. As far as the slicing criterion is concerned, we have to make some more changes in the definition. In particular, for the KL form of slicing we simply add a boolean parameter specifying whether we are considering a KL form or not, i.e., $\mathcal{C} = (\mathcal{I}, V, n, \mathcal{L})$, where $\mathcal{L} = true$ if we are considering a KL form of slicing, it is *false* otherwise. While, in order to model the IC form we have to change the third parameter of the criterion, in particular let $k \in \mathbb{N}$ be the iteration of the program point $n \in \mathbb{L}$ we are interested in, then instead of n in the criterion we should have $\langle n, k \rangle$. Hence, $\mathcal{C} = (\mathcal{I}, V, \mathcal{O}, \mathcal{L})$, where $\mathcal{O} \in \mathbb{L} \times \wp(\mathbb{N})$. Note that $\{n\} \times \mathbb{N}$ represents the fact that we are interested in all occurrences of n.

There are also some *simultaneous* (sim.) types of slicing that consider more points of interest, and at each of them the slice has to preserve the values of all variables of interest. In order to deal also with sim. forms of slicing we have simply to extend the definition of slicing criterion by considering $\mathcal{O}$ a set instead of a singleton, namely $\mathcal{O} \in \wp(\mathbb{L} \times \wp(\mathbb{N}))$.

The formal framework proposed in [1,2] represents different forms of slicing by means of a pair: a *syntactic preorder*, a *function from slicing criteria to semantic equivalences*. The *preorder* fixes a syntactic relation between the program and its slices. In standard slicing, this relation represents the fact that slices are obtained from the original program by removing zero or more statements. This preorder is called *traditional syntactic ordering*, denoted $\sqsubseteq$, and is defined as follows: $Q \sqsubseteq P \Leftrightarrow \mathcal{F}(Q) \subseteq \mathcal{F}(P)$, where $\mathcal{F}(P)$ maps l to c if and only if program P contains the statement c at line number l.

The *function* fixes the semantic constraints that a subprogram has to respect in order to be a slice of the original program. It is worth noting that the equivalence relation returned by the function is uniquely determined by the form of slicing and by the chosen slicing criterion. In this way Binkley et al. are able to characterize eight forms of non-sim. slicing, and twelve forms of sim. slicing.

Finally, this framework is used to formally compare notions of slicing. Given two semantic equivalence relations $\approx_A$ and $\approx_B$, we say that $\approx_A$ subsumes $\approx_B$ if and only if for every two programs P and Q, $P \approx_B Q \Rightarrow P \approx_A Q$.

Given a binary relation on slicing criteria $\rightarrow$, we say that a form of slicing $(\sqsubseteq, \mathcal{E}_A)$ is *weaker than* $(\sqsubseteq, \mathcal{E}_B)$ w.r.t. $\rightarrow$ iff $\forall \mathcal{C}_A, \mathcal{C}_B$, slicing criteria such that $\mathcal{C}_A \rightarrow \mathcal{C}_B$, and $\forall P, Q$, if Q is a slice of P w.r.t. $(\sqsubseteq, \mathcal{E}_B(\mathcal{C}_B))$, then Q is a slice of P w.r.t. $(\sqsubseteq, \mathcal{E}_A(\mathcal{C}_A))$ as well, formally $(\sqsubseteq, \mathcal{E}_A) \overrightarrow{\subseteq} (\sqsubseteq, \mathcal{E}_B)$. Following this definition Binkley et al. show that all forms of slicing introduced in [1] are comparable in the way shown in Fig. 2. The symbols $\mathcal{S}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{SS}$, $\mathcal{SC}$ and $\mathcal{SD}$ represent static,

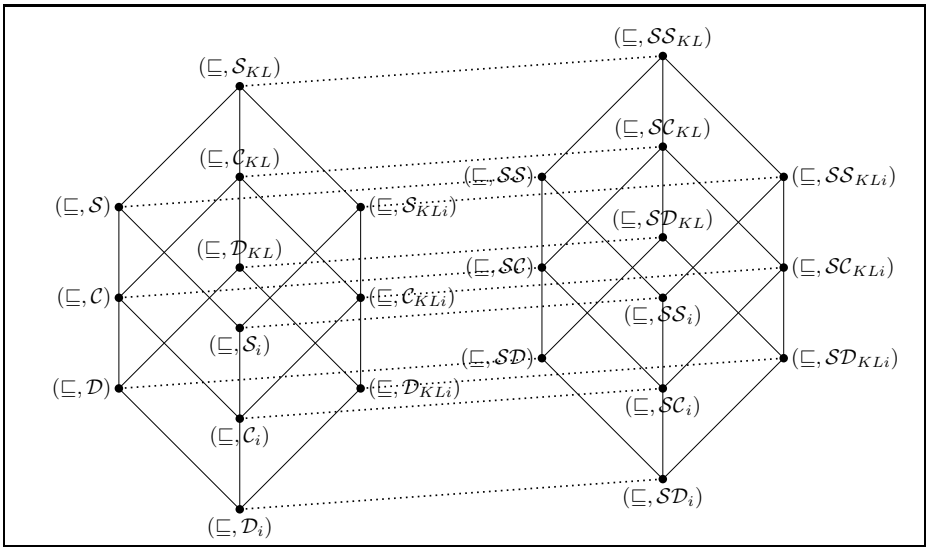

Fig. 2. Given two forms A and B, both (non-)sim., A is weaker than B if A is connected to B by a solid line and it is below B. If A is non-sim. and B is sim., then A is weaker than B if A is connected to B by a dotted line and it is to the left of B.

conditioned, dynamic, static sim., conditioned sim. and dynamic sim. types of slicing respectively. The subscripts i, KL and KLi represent IC, KL and KLi forms of slicing respectively, no subscript denotes standard forms of slicing. Following their structure, we enrich one of the slicing criterion comparison relations and consequently we insert the four non-sim. forms of conditioned slicing in the hierarchy constructed in [1], as shown in Fig. 2.

3 Abstract Program Slicing

Program slicing is used for reducing the size of programs to analyze. Nevertheless, sometimes this reduction is not sufficient for really improving the analyses. Suppose that some variables at some point of execution do not have a desired property. In order to understand where the error occurred it would be useful to find the statements affecting the property of these variables. Standard slicing may return too many statements, making it hard for the programmer to realize which ones caused the error. Consider the following example.

Example 2. Let us consider a program P in Fig. 3, that reverses a linked list. Lists are defined recursively with selectors *data*, storing the information, and *next*, pointing to the following element. Suppose a property of *well-formedness* is defined over lists. A list is well-formed if it has $data = [0]$ in the last element. A well-formed empty list is presented as $\langle[0]\rangle$, where square brackets indicate that 0 is not a proper element. Program P reverses a list l passed to it as argument, so if $l = \langle 1, 2, 3, 4, [0]\rangle$, the correct implementation of P should return $\langle 4, 3, 2, 1, [0]\rangle$.

```
1 list rev(list l) {
2   list *last;
3   list *tmp;
4   while (l->next != null){
5     tmp = l->next;
6     l->next = last;
7     last = l;
8     l = tmp;
9   }
10  return last;
11}
```

Fig. 3.

After running the program, we can realize that at line 9, list *last* is not well-defined. Standard static slicing w.r.t. criterion $(\{last\}, 9)$ would return the whole program as slice, since *last* is affected, even if not directly, by all statements. But if we use the property of *well-formedness*, and if we want to understand if *last* is well-formed, a slicing based on that abstract criterion should return the following slice: *list* $*last$; **return** *last*;, since the *well-formedness* property of *last* is not affected by the *while*. In this way we are able to characterize that the error occurred at line 2.

Let us introduce a new technique, called *abstract slicing*, which, regarding the syntax, can only delete statements from programs, while, regarding the semantics, compares the program and its abstract slices by considering *properties* instead of exact values of some variables. These properties are represented as abstract domains of the variable domain in the context of abstract interpretation [6]. The *abstract slicing* should help us finding all the statements affecting some particular *properties* of variables of interest.

First of all, we introduce the notion of abstract slicing criterion where we specify also the *property* of interest. For the sake of simplicity we define the abstract slicing criterion (Def. 1) only for non-sim. forms, i.e., $\mathcal{O}$ is a singleton and not a set of occurrences ($\mathcal{O} = \mathbb{L} \times \wp(\mathbb{N})$). In order to be as general as possible, we consider relational properties of variables and, for this reason, properties are associated with tuples and not with single variables.

Definition 1. *Let $\mathcal{I} \subseteq \mathbb{M}$ be a set of input memories, $V \subseteq Var$ be a set of variables of interest, $\mathcal{O} \in \mathbb{L} \times \wp(\mathbb{N})$ be a set of occurrences of interest, $\mathcal{L}$ a boolean value determining KL forms. Let $V_1, \ldots, V_k$ be a partition of V and for each $i \in [1..k]$ let φ_i be a property of interest (modelled as an uco [6]) for V_i, then the abstract slicing criterion is $\mathcal{C}_\mathcal{A} = (\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A})$, where $\mathcal{V} \stackrel{def}{=} \langle V_1, \ldots, V_k \rangle$ and $\mathcal{A} \stackrel{def}{=} \langle \varphi_1, \ldots, \varphi_k \rangle$.*

The extension of this definition to sim. forms is possible by considering $\mathcal{A}$ as a partial function that takes the set of points/occurrences of interest, the corresponding tuples of variables of interest $\mathcal{V}$, and return an abstract property to observe on $\mathcal{V}$ in the specified point/occurrence. For this reason in the following we consider any form of slicing. Note that, when dealing with non-abstract notions of slicing we have $\mathcal{A} = \langle \varphi_{\text{ID}}, \ldots, \varphi_{\text{ID}} \rangle \stackrel{\text{def}}{=} \mathcal{ID}$, where $\varphi_{\text{ID}} \stackrel{\text{def}}{=} \lambda x.x$.

It is worth underlying that, exactly as it happens for the non-abstract forms, we have that if $\mathcal{I}=\mathbb{M}$ we have *abstract static slicing*, if $|\mathcal{I}|=1$ we have *abstract dynamic slicing*, otherwise we have *abstract conditioned slicing* .

Definition 2. *Let P and Q be executable programs such that Q can be obtained from P by removing zero or more statements and let $\mathcal{C}_{\mathcal{A}}=(\mathcal{I},\mathcal{V},\mathcal{O},\mathcal{L},\mathcal{A})$. Q is an* abstract slice *of P w.r.t. $\mathcal{C}_{\mathcal{A}}$ if for each $\rho\in\mathcal{I}$, when the execution of P from input ρ reaches the point (or ocurrence) $\mathcal{O}$, the execution of Q from ρ reaches $\mathcal{O}$ as well, and for each $i\in[1..k]$, V_i has the same property φ_i both in P and in Q. Moreover, if $\mathcal{L}=$ true then P and Q have to follow identical paths [12]. The extraction of* abstract slices *is called* abstract slicing.

3.1 Abstract Formal Framework

In this section we define the notion of abstract formal framework in which all forms of slicing can be formally represented. It is an extension of a mathematical structure introduced by Binkley et al. [1,2], that is used for representing and comparing different forms of slicing.

Following the framework of Binkley et al., described in Sect. 2, we represent an abstract form of slicing by a pair $(\sqsubseteq,\mathcal{E}_{\mathcal{A}})$, where $\sqsubseteq$ is the traditional syntactic ordering and $\mathcal{E}_{\mathcal{A}}$ is a function that maps abstract slicing criteria to semantic equivalence relations on programs. Given two programs P and Q, and an abstract slicing criterion $\mathcal{C}_{\mathcal{A}}$ we say that Q is $(\sqsubseteq,\mathcal{E}_{\mathcal{A}})$-*(abstract) slice* of P with respect to $\mathcal{C}_{\mathcal{A}}$ iff $Q\sqsubseteq P$ and $Q\ \mathcal{E}_{\mathcal{A}}(\mathcal{C}_{\mathcal{A}})\ P$. At this point we have to define the function $\mathcal{E}_{\mathcal{A}}$ in the context of *abstract slicing*. In order to derive this equivalence we have to define some preliminary useful notions.

Let us first define the *abstract memory state* which restricts the domain of a memory state to variables of interest only, and assigns to each tuple an abstract value determined by its abstract property of interest.

Definition 3. *Let $\rho\in\mathbb{M}$ be a memory state, $\mathcal{V}=\langle V_1,\ldots,V_k\rangle$ a tuple of variables and $\mathcal{A}=\langle\varphi_1,\ldots,\varphi_k\rangle$ the corresponding tuple of properties of interest. The* abstract memory state *is $\rho\restriction^{\alpha}\mathcal{V}\stackrel{def}{=}\mathcal{A}\circ\rho(\mathcal{V})\stackrel{def}{=}\langle\varphi_1\circ\rho(V_1),\ldots,\varphi_k\circ\rho(V_k)\rangle$.*

Consider the following example.

Example 3. Let $Var=\{x_1,x_2,x_3,x_4\}$ be a set of variables and suppose the properties of interest are the sign of $x_1\times x_2$ and the parity of x_3. Let us consider *ucos* $\varphi_{\textsc{Sign}}=\{(\varnothing,\varnothing),(\mathbb{Z}^+,\mathbb{Z}^-)\cup(\mathbb{Z}^-,\mathbb{Z}^+),(\mathbb{Z}^+,\mathbb{Z}^+)\cup(\mathbb{Z}^-,\mathbb{Z}^-),(\mathbb{Z},\mathbb{Z})\}$ and $\varphi_{\textsc{Par}}=\{\varnothing,2\mathbb{Z},2\mathbb{Z}+1,\mathbb{Z}\}$. $\varphi_{\textsc{Par}}$ is defined on single integer variables v and represents their parity, i.e., if the value of v is even (odd), then it is mapped to all even (odd) numbers, $2\mathbb{Z}$ $(2\mathbb{Z}+1)$. $\varphi_{\textsc{Sign}}$ is defined on pairs of integer variables $\langle v,t\rangle$ and represents the sign of their product, i.e., if v and t are both positive or both negative, the pair is mapped to $(\mathbb{Z}^+,\mathbb{Z}^+)\cup(\mathbb{Z}^-,\mathbb{Z}^-)$ meaning that $v\times t$ is positive, otherwise it is mapped to $(\mathbb{Z}^+,\mathbb{Z}^-)\cup(\mathbb{Z}^-,\mathbb{Z}^+)$ meaning that $v\times t$ is negative.

Therefore, we consider $\mathcal{V}=\langle\{x_1,x_2\},\{x_3\}\rangle$ and $\mathcal{A}=\langle\varphi_{\textsc{Sign}},\varphi_{\textsc{Par}}\rangle$. Let $\rho=\langle1,2,3,4\rangle$, then we obtain $\rho\restriction^{\alpha}\mathcal{V}=\mathcal{A}\circ\rho(\mathcal{V})=\langle(\mathbb{Z}^+,\mathbb{Z}^+)\cup(\mathbb{Z}^-,\mathbb{Z}^-),2\mathbb{Z}+1\rangle$.

The *abstract projection* function modifies the state trajectory by removing all the states that do not contain occurrences or additional points of interest. If there is a state that contains an occurrence of interest, its memory state component is restricted to variables of interest, and for each tuple of interest only a property of interest is considered. We define an auxiliary function $Proj'^{\mathcal{A}}$ and the *abstract projection function*, $Proj^{\mathcal{A}}$ formally.

Definition 4. *Given $n \in \mathbb{L}$, $k \in \mathbb{N}$, $\rho \in \mathbb{M}$, and parameters $\mathcal{V}, \mathcal{O}, L, \mathcal{A}$, where $L \subseteq \mathbb{L}$ is a set of line numbers, we define a function $Proj'^{\mathcal{A}}$ as:*

$$Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},L,\mathcal{A})}(n,k,\rho) \stackrel{def}{=} \begin{cases} (n, \rho \restriction^{\alpha} \mathcal{V}) & \text{if } (n,k) \in \mathcal{O}, \\ (n, \rho \restriction^{\alpha} \varnothing) & \text{if } (n,k) \notin \mathcal{O} \wedge n \in L \\ \lambda & \text{otherwise.} \end{cases}$$

$Proj'^{\alpha}$ takes a state of a state trajectory and returns a pair or an empty string, λ. If (n,k) is an occurrence of interest, i.e., if $(n,k) \in \mathcal{O}$, it returns a pair $(n, \rho \restriction^{\alpha} \mathcal{V})$. It means that at n we consider properties of interest for each tuple of interest, and not their exact values. If (n,k) is not an occurrence of interest, but n is an additional point of interest due to a *KL* form, i.e., if $n \in L$, $Proj'^{\alpha}$ returns a pair $(n, \rho \restriction^{\alpha} \varnothing)$. It means that we require the presence of n, but we are not interested in any property of any tuple, and therefore the domain of $\rho \restriction^{\alpha}$ is $\varnothing$.

Definition 5 (Abstract Projection). *Let* $T = (n_1, k_1, \rho_1) \ldots (n_l, k_l, \rho_l) \stackrel{def}{=} \bigoplus_{i=1}^{l} (n_i, k_i, \rho_i)$ *be a state trajectory, we define the notion of abstract projection as* $Proj^{\alpha}_{(\mathcal{V},\mathcal{O},L,\mathcal{A})}(T) \stackrel{def}{=} \bigoplus_{i=1}^{l} Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},L,\mathcal{A})}(n_i, k_i, \rho_i)$.

The *abstract projection* function permits us to define all the semantic equivalence relations we need for representing the abstract forms of slicing: *abstract static backward equivalence*, *abstract dynamic backward equivalence* and *abstract conditioned backward equivalence*. If we have two programs P and Q, we can say that Q is an *abstract (static, dynamic or conditioned) slice* of P if $Q \sqsubseteq P$ and if Q is *abstract backward equivalent* to P w.r.t. the corresponding semantic equivalence relation.

3.2 Abstract Unified Equivalence

The only missing thing for completing the formal definitions of the three forms of abstract slicing is the characterization of the functions mapping slicing criteria to semantic equivalence relations. The *abstract unified equivalence*, $\mathcal{U}^{\mathcal{A}}$, is a function that takes parameters $\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A}$ characterizing the slicing criterion and, therefore, the form of slicing, and returns a corresponding abstract semantic equivalence relation, which has to hold between a program and each one of its (abstract) slices. It can be used as a unified model for representing abstract equivalence relations for all possible forms of slicing.

Definition 6 ($\mathcal{U}^{\mathcal{A}}$). *Let P and Q be executable programs, $\mathcal{C}_{\mathcal{A}} = (\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A})$ be an abstract criterion, and let L be such that $\mathsf{L}_{\mathcal{L}}(P,Q) = I_P \cap I_Q$, where $I_P \subseteq \wp(\mathbb{L})$*

denotes the set of all line numbers of P, if $\mathcal{L} = true$, $\mathsf{L}_{\mathcal{L}}(P, Q) = \varnothing$ otherwise[2]. *Then P is* abstract backward equivalent *to Q, denoted $P\ \mathcal{U}^{\mathcal{A}}(\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathsf{L}_{\mathcal{L}}, \mathcal{A})\ Q$, iff* $\forall \rho \in \mathcal{I}. Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\mathsf{L}_{\mathcal{L}}(P,Q),\mathcal{A})}(T^{\rho}_{P}) = Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\mathsf{L}_{\mathcal{L}}(P,Q),\mathcal{A})}(T^{\rho}_{Q})$.

Now we are able to define the functions mapping each criterion $\mathcal{C}_{\mathcal{A}}$ to the corresponding semantic equivalence relation.

$$\mathcal{E}_{\mathcal{A}} \stackrel{\text{def}}{=} \lambda(\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A}).\, \mathcal{U}^{\mathcal{A}}(\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathsf{L}_{\mathcal{L}}, \mathcal{A})$$

Hence a generic form of slicing can be represented as $(\sqsubseteq, \mathcal{E}_{\mathcal{A}})$. These functions can be used for the formal definitions of both abstract and non-abstract forms of slicing in the abstract formal framework. We can therefore state that the abstract formal framework is a generalization of the original formal framework. The following examples show how it is possible to use these definitions in order to check whether a program is an *abstract slice* of another one.

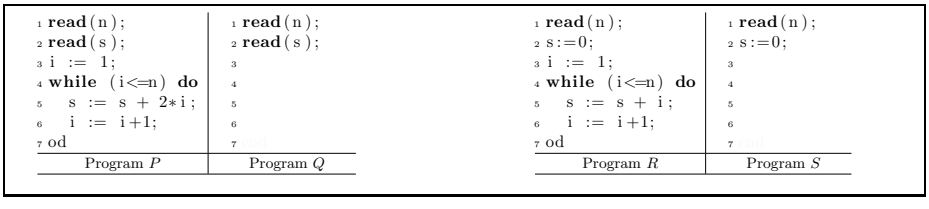

Fig. 4. **Fig. 5.**

Example 4. Let us consider programs P and Q given in Fig. 4, and let $\mathcal{C}_{\mathcal{A}} = (\mathbb{M}, \langle s \rangle, \{7\} \times \mathbb{N}, \mathit{false}, \langle \varphi_{\text{PAR}} \rangle)$, i.e., we are interested in the parity ($\mathcal{A} = \langle \varphi_{\text{PAR}} \rangle$) of s ($\mathcal{V} = \langle s \rangle$) at the end of execution ($\mathcal{O} = \{7\} \times \mathbb{N}$) for all possible inputs ($\mathcal{I} = \mathbb{M}$). Since $Q \sqsubseteq P$, in order to show that Q is an *abstract static slice* of P w.r.t. $C_{\mathcal{A}}$, we have to show that $P\ \mathcal{E}_{\mathcal{A}}(\mathcal{C}_{\mathcal{A}})\ Q$ holds. Let $\rho = \{n \leftarrow a, s \leftarrow b\} \in \mathcal{I}$ be an initial memory. Execution of P from ρ determines:

$$\begin{array}{c} T^{\rho}_{P} = \ldots (5^1, \langle a, b, 1 \rangle)(6^1, \langle a, b+2, 1 \rangle)(4^2, \langle a, b+2, 2 \rangle) \ldots \\ (5^a, \langle a, b+2+\ldots+2(a-1), a \rangle)(6^a, \langle a, b+2+\ldots+2a, a+1 \rangle) \\ (4^{a+1}, \langle a, b+2+\ldots+2a, a+1 \rangle)(7^1, \langle a, b+a(a+1), a+1 \rangle), \end{array}$$

where a^b is the b-th occurrence of point a. Application of $Proj^{\alpha}$ to T^{ρ}_{P} gives:

$$\begin{array}{c} Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{P}) = Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(7^1, \langle a, b+a(a+1), a+1 \rangle) \\ = (7, \langle a, b+a(a+1), a+1 \rangle \restriction^{\alpha} \{\langle s \rangle\}) = (7, \varphi_{\text{PAR}}(\langle b+a(a+1) \rangle)) \\ = (7, \varphi_{\text{PAR}}(b)). \end{array}$$

Let us give some clarifications: the only state of interest is the state containing the occurrence of the point 7, $(7^1, \langle a, b+a(a+1), a+1 \rangle)$, so we apply the function

[2] When dealing with *KL* forms, i.e., $\mathcal{L} = true$ it takes the intersection of the line numbers, otherwise it is the empty set.

$Proj'^{\alpha}$ to that state. Since we have $7^1 = (7,1) \in \mathcal{O}$, $Proj'^{\alpha}$ returns $(7, \rho \restriction^{\alpha} \mathcal{V}) = (7, \langle a, b{+}a(a{+}1), a{+}1\rangle \restriction^{\alpha} \{\langle s\rangle\})$. The *abstract memory state* restricts the domain of ρ to variables of interest only, so we consider only the part of ρ regarding s, i.e., $b{+}a(a{+}1)$. Hence, the result of the last application is $(7, \mathcal{A} \circ \rho(\{\langle s\rangle\}) = (7, \varphi_{\text{PAR}}(\langle b{+}a(a{+}1)\rangle))$. Since the parity of $b{+}a(a{+}1)$ depends on the parity of b only, the final result of the application of $Proj^{\alpha}$ is $(7, \varphi_{\text{PAR}}(b))$. Now, the execution of Q from ρ gives the following state trajectory: $T^{\rho}_{Q} = (1^1, \langle a\rangle)(2^1, \langle a, b\rangle)(7^1, \langle a, b\rangle)$. Application of $Proj^{\alpha}$ to T^{ρ}_{Q} gives: $Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{Q}) = Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(7^1, \langle a, b\rangle) = (7, \langle a, b\rangle \restriction^{\alpha} \{\langle s\rangle\}) = (7, \varphi_{\text{PAR}}(b))$, and therefore $Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{P}) = Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{Q})$. As ρ is an arbitrary input, we realize that this equation holds for each $\rho \in \mathcal{I}$, hence $P\ \mathcal{E}_{\mathcal{A}}(\mathcal{C}_{\mathcal{A}})\ Q$, and this implies that Q is an *abstract static slice* of P w.r.t. $\mathcal{C}_{\mathcal{A}}$.

Example 5. Consider R and S in Fig. 5, and let $\mathcal{C}_{\mathcal{A}} = (\mathcal{I}, \langle s\rangle, \{7\} \times \mathbb{N}, \mathit{false}, \langle \varphi_{\text{PAR}}\rangle)$, where $\mathcal{I} = \{\rho \mid \rho(n) \in 4\mathbb{Z}\}$, i.e., we are interested in the parity of s at the end of execution for all inputs that assign to n a multiple of 4. Since $S \sqsubseteq R$, in order to show that S is an *abstract conditioned slice* of R w.r.t. $\mathcal{C}_{\mathcal{A}}$, we have to show that $R\ \mathcal{E}_{\mathcal{A}}(\mathcal{C}_{\mathcal{A}})\ S$ holds. Let $\rho \in \mathcal{I}$ be an initial memory, and suppose $\rho(n) = a = 4b$. Execution of R from ρ determines:

$$\begin{gathered} T^{\rho}_{R} = \ldots (5^1, \langle a, 0, 1\rangle)(6^1, \langle a, 1, 1\rangle)(4^2, \langle a, 1, 2\rangle) \ldots \\ (5^a, \langle a, 1{+}2{+}\ldots{+}(a{-}1), a\rangle)(6^a, \langle a, 1{+}2{+}\ldots{+}a, a{+}1\rangle) \\ (4^{a+1}, \langle a, \tfrac{1}{2}a(a{+}1), a{+}1\rangle)(7^1, \langle a, \tfrac{1}{2}a(a{+}1), a{+}1\rangle), \end{gathered}$$

Application of $Proj^{\alpha}$ to T^{ρ}_{R} gives:

$$\begin{gathered} Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{R}) = Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(7^1, \langle a, \tfrac{1}{2}a(a{+}1), a{+}1\rangle) \\ = (7, \langle a, \tfrac{1}{2}a(a{+}1), a{+}1\rangle \restriction^{\alpha} \{\langle s\rangle\}) = (7, \varphi_{\text{PAR}}(\langle \tfrac{1}{2}a(a{+}1)\rangle)) \\ = (7, \varphi_{\text{PAR}}(2b(4b{+}1))) = (7, 2\mathbb{Z}). \end{gathered}$$

Execution of S from ρ gives the state trajectory $T^{\rho}_{S} = (1^1, \langle a\rangle)(2^1, \langle a, 0\rangle)(7^1, \langle a, 0\rangle)$. Application of $Proj^{\alpha}$ to T^{ρ}_{S} gives: $Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{S}) = Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(7^1, \langle a, 0\rangle) = (7, \langle a, 0\rangle \restriction^{\alpha} \{\langle s\rangle\}) = (7, \varphi_{\text{PAR}}(0))) = (7, 2\mathbb{Z})$, and therefore we have $Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{R}) = Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{S})$. As ρ is an arbitrary input from $\mathcal{I}$, we realize that this equation holds for each $\rho \in \mathcal{I}$, hence $R\mathcal{E}_{\mathcal{A}}(\mathcal{C}_{\mathcal{A}})S$, and this implies that S is an *abstract conditioned slice* of R w.r.t. $\mathcal{C}_{\mathcal{A}}$.

4 Comparing Forms of Abstract Slicing

In this section we provide a formal theory which allows to compare the abstract forms of slicing among themselves. Moreover, in the same theory we can compare the abstract forms of slicing with the non-abstract ones. First of all, we show under which conditions an abstract semantic equivalence relation subsumes another one, and analogously we show when the form of (*abstract*) slicing corresponding to the former equivalence relation subsumes the form of (*abstract*) slicing corresponding to the latter one. These results are necessary for obtaining a precise characterization of the *weaker then* relation (Sect. 2) involving also the new forms of slicing.

The *abstract unified equivalence lemma* ($\mathcal{U}^{\mathcal{A}}$-lemma), shows under which conditions on the parameters determined by the slicing criteria, two semantic equivalence relations $\approx_A$ and $\approx_B$ are such that $\approx_A$ subsumes $\approx_B$.

Lemma 1 ($\mathcal{U}^{\mathcal{A}}$-lemma). *Given two forms of (abstract) slicing and their corresponding criteria $\mathcal{C}^i_{\mathcal{A}} = (\mathcal{I}^i, \mathcal{V}^i, \mathcal{O}^i, \mathcal{L}^i, \mathcal{A}^i)$, with $i \in \{1, 2\}$.*

If $\mathcal{I}^1 \subseteq \mathcal{I}^2, \mathcal{O}^1 \subseteq \mathcal{O}^2, \mathcal{V}^1 \subseteq \mathcal{V}^2, \forall P, Q. \mathsf{L}_{\mathcal{L}^1}(P, Q) \subseteq \mathsf{L}_{\mathcal{L}^2}(P, Q)$ and $\mathcal{A}^2 \overline{\sqsubseteq} \mathcal{A}^1$, where $\overline{\sqsubseteq}$ denotes the approximation relation between tuples of abstractions[3], then we have that $P\ \mathcal{U}^{\mathcal{A}}(\mathcal{I}^2, \mathcal{V}^2, \mathcal{O}^2, \mathsf{L}_{\mathcal{L}^2}, \mathcal{A}^2)\ Q \Rightarrow P\ \mathcal{U}^{\mathcal{A}}(\mathcal{I}^1, \mathcal{V}^1, \mathcal{O}^1, \mathsf{L}_{\mathcal{L}^1}, \mathcal{A}^1)\ Q$.

This lemma tells us how it is possible to find the relationship (in the sense of subsume) between two semantic equivalence relations determined by two forms of (*abstract*) slicing. In particular, if we denote abstract notions of slicing by putting an $\mathcal{A}$, e.g., $\mathcal{AS}$ denotes static abstract slicing and $\mathcal{AD}$ denotes dynamic abstract slicing, by using this lemma we can show that, given a slicing criterion $\mathcal{C}_{\mathcal{A}}$, all the abstract equivalence relations introduced in Sect. 3.2, subsume the corresponding non-abstract equivalence relations $\mathcal{S}(\mathcal{C}_{\mathcal{A}})$, $\mathcal{D}(\mathcal{C}_{\mathcal{A}})$ and $\mathcal{C}(\mathcal{C}_{\mathcal{A}})$. Furthermore, by using this lemma we can show that $\mathcal{AD}(\mathcal{C}_{\mathcal{A}})$ subsumes $\mathcal{AC}(\mathcal{C}_{\mathcal{A}})$ which subsumes $\mathcal{AS}(\mathcal{C}_{\mathcal{A}})$. Hence, let us first recall an important result that we then apply in the context of abstract slicing, showing the relationship between the different forms of slicing.

Theorem 1. *[2] Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be semantic equivalence relations such that $\mathcal{R}_2$ subsumes $\mathcal{R}_1$, then $\forall P, Q$ we have $P\ (\sqsubseteq, \mathcal{R}_1)\ Q\ \Rightarrow\ P\ (\sqsubseteq, \mathcal{R}_2)\ Q$.*

Hence, for instance, given two slicing criteria $\mathcal{C}'_{\mathcal{A}}$ and $\mathcal{C}''_{\mathcal{A}}$ satisfying hypotheses of Lemma 1, for each pair of programs P and Q:

$P\ (\sqsubseteq, \mathcal{AS}(\mathcal{C}''_{\mathcal{A}}))\ Q \Rightarrow P\ (\sqsubseteq, \mathcal{AC}(\mathcal{C}'_{\mathcal{A}}))\ Q$
$P\ (\sqsubseteq, \mathcal{S}(\mathcal{C}''_{\mathcal{A}}))\ Q \Rightarrow P\ (\sqsubseteq, \mathcal{AS}(\mathcal{C}'_{\mathcal{A}}))\ Q$
$P\ (\sqsubseteq, \mathcal{C}(\mathcal{C}''_{\mathcal{A}}))\ Q \Rightarrow P\ (\sqsubseteq, \mathcal{AC}(\mathcal{C}'_{\mathcal{A}}))\ Q$
$P\ (\sqsubseteq, \mathcal{AC}(\mathcal{C}''_{\mathcal{A}}))\ Q \Rightarrow P\ (\sqsubseteq, \mathcal{AD}(\mathcal{C}'_{\mathcal{A}}))\ Q$
$P\ (\sqsubseteq, \mathcal{D}(\mathcal{C}''_{\mathcal{A}}))\ Q \Rightarrow P\ (\sqsubseteq, \mathcal{AD}(\mathcal{C}'_{\mathcal{A}}))\ Q$

Finally, in order to formally prove the *weaker than* relation among the considered forms of slicing we have to define the slicing criteria comparison relation as introduced in Sect. 2. We define two slicing criterion comparison relations $\overset{\alpha_1}{\rightarrow}$ and $\overset{\alpha_2}{\rightarrow}$. $\overset{\alpha_1}{\rightarrow}$ permits to compare criteria of abstract forms of slicing among themselves, while $\overset{\alpha_2}{\rightarrow}$ permits to compare criteria of abstract forms of slicing with criteria of non-abstract forms of slicing.

Definition 7. *The slicing criterion comparison relation $\overset{\alpha_1}{\rightarrow}$ is defined by the following rule: $\forall \mathcal{I}^1, \mathcal{I}^2 \subseteq \mathbb{M}$ such that $\mathcal{I}^1 \subseteq \mathcal{I}^2$, $\forall \mathcal{V} = \langle V_1, \ldots, V_k \rangle, \forall \mathcal{A} = \langle \varphi_1, \ldots, \varphi_k \rangle, \forall \mathcal{O} \in \mathbb{L} \times \wp(\mathbb{N}), \mathcal{L} \in \{true, false\}$*

$$(\mathcal{I}^1, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A}) \overset{\alpha_1}{\rightarrow} (\mathcal{I}^2, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A}).$$

For instance, this rule permits us to compare $(\sqsubseteq, \mathcal{AC})$ to $(\sqsubseteq, \mathcal{AS})$ being $\mathcal{I} \subseteq \mathbb{M}$, and $(\sqsubseteq, \mathcal{AD})$ to $(\sqsubseteq, \mathcal{AC})$ whenever $\rho \in \mathcal{I}$.

[3] We denote $\overline{\sqsubseteq}$ the relation "more concrete than" in the lattice of abstract interpretations between tuples of abstractions [6], i.e., $\mathcal{A}^1 \overline{\sqsubseteq} \mathcal{A}^2$ iff $\forall i \leq k.\ \varphi^1_i \sqsubseteq \varphi^2_i$ and $\varphi' \sqsubseteq \varphi''$ iff $\forall x.\ \varphi'(x) \leq \varphi''(x)$.

Definition 8. *The slicing criteria comparison relation* $\overset{\alpha_2}{\rightarrow}$ *is defined by the following rule:* $\forall \mathcal{I} \subseteq \mathbb{M}$, $\forall \mathcal{V} = \langle V_1, \ldots, V_k \rangle, \forall \mathcal{A} = \langle \varphi_1, \ldots, \varphi_k \rangle$ *where* $V = \bigcup_{i \in [1,k]} V_i$, $\forall \mathcal{O} \in \mathbb{L} \times \wp(\mathbb{N}), \mathcal{L} \in \{true, false\}$,

$$(\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A}) \overset{\alpha_2}{\rightarrow} (\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{ID}) = (\mathcal{I}, V, \mathcal{O}, \mathcal{L}).$$

This rules allows, for instance, to compare $(\sqsubseteq, \mathcal{AS})$ to $(\sqsubseteq, \mathcal{S})$, $(\sqsubseteq, \mathcal{AC})$ to $(\sqsubseteq, \mathcal{C})$ and $(\sqsubseteq, \mathcal{AD})$ to $(\sqsubseteq, \mathcal{D})$.

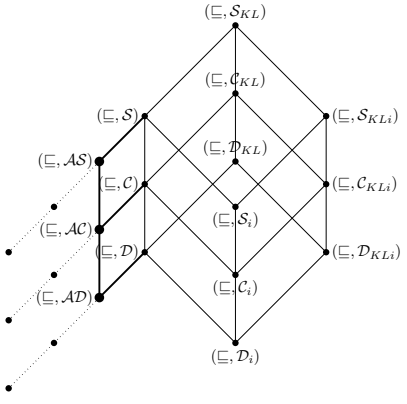

Fig. 6.

In Fig. 6 we show the non-sim. hierarchy obtained by enriching the hierarchy in Fig. 2 with standard forms of *abstract static slicing*, *abstract dynamic slicing* and *abstract conditioned slicing*. In general we can enrich this hierarchy with any abstract form of slicing simply by using the comparison notions defined above. Non-abstract forms are particular cases of abstract forms of slicing, as they can be instantiated by choosing the identity property, φ_{ID}, for each variable of interest. Hence, non-abstract forms are the "strongest" forms, since for each property φ, we have $\varphi_{\text{ID}} \sqsubseteq \varphi$. Moreover, if we fix the parameters $\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}$ of an abstract form, by abstracting the parameter $\mathcal{A}$, i.e., by reducing the information represented by the property, the abstract slicing forms we obtain become weaker. The dotted lines in Fig. 6 represent this fact.

5 Towards an Implementation

In this section, we describe an algorithmic approach for obtaining *abstract slices*. The idea is to define a notion of abstract state which observes variables of interest by means of abstract properties. These states are used for analyzing the *evolution* of the *properties* of variables of interest instead of their values. In order to perform the *evolution* analysis, we construct the *abstract* state graph (ASG), whose vertices are abstract states and which models program executions at some level of abstraction. At this point, we propose a technique for *pruning* the ASG in order to remove *all* the statements not relevant for the properties of interest.

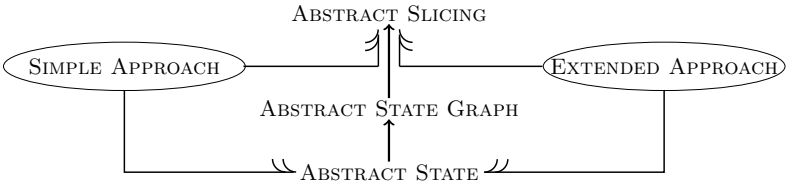

Fig. 7.

In the following we consider two different approaches: *simple* (*SA*) and *extended* (*EA*). The difference between them lies in the *abstract state* definition, where *EA* adds information about the relationship between input variables and properties of variables of interest. Moreover, the two approaches differ also in the pruning process. In particular, *EA*'s pruning consists of more than one application of *SA*'s pruning. See Fig. 7 for a graphical representation of our approaches. Note that *SA* can be used for extraction of *abstract static slices* only. Unfortunately, since it does not consider the relationship between variables of interest and input variables, it is not precise enough and cannot be applied to the problem of *abstract conditioned slicing*.

The *simple approach pruning* is illustrated by the method **pruningSA** given in Fig. 8, and we illustrate its application by the following example. Instead, the *extended approach pruning* is characterized by the method **pruningEA**, given in Fig. 10, and it will be only informally and intuitively described.

pruningSA(G)	**removeBlock**(G, B)	**findBlocks**(G)

Input: $G=(V,E)$ - ASG_S
Output: pruned G
$list=$**findBlocks**(G);
foreach $B \in list$
  if $(absv(in(B))=absv(out(B)))$
  then
  $G=$**removeBlock**(G, B);
  $E=E\cup\{(in(B),out(B))\}$;
  $G=(V,E)$;
  fi
endf
return G;
	Input: $G=(V,E)$ - ASG_S,
	$B \subseteq V$ - block
	Output: G' - ASG_S
	foreach $a \in B$
	  foreach $(a,b) \in E$
	  $E=E\setminus\{(a,b)\}$;
	  endf
	  foreach $(b,a) \in E$
	  $E=E\setminus\{(b,a)\}$;
	  endf
	  $V=V\setminus\{a\}$;
	endf
	return (V,E);
		Input: $G=(V,E)$ - ASG_S
		Output: list of blocks to be removed
		$list=\emptyset$;
		$G^{SCC}=(V^{SCC},E^{SCC})=$**tarjan**($G$);
		foreach component $V_i \in V^{SCC}$
		  $i=0$; $o=0$;
		  foreach $(V_i,V_j) \in E^{SCC}$ $o=o+1$; **endf**
		  foreach $(V_j,V_i) \in E^{SCC}$ $i=i+1$; **endf**
		  if $i=1 \wedge o=1$ **then**
		  $list=list\cup\{C_i\}$
		  fi
		endf
		return $list$;

Fig. 8.

Example 6. Consider P in Fig. 4, and let $\mathcal{C}_{\mathcal{A}}=(\mathbb{M},\langle s\rangle,\{7\}\times\mathbb{N},\mathit{false},\langle\varphi_{\mathrm{PAR}}\rangle)$. In the *abstract states* induced by α, s can have two abstract values: `E`, if its concrete value is even, or `O`, if its concrete value is odd. In Fig. 9 we give the ASG of P for $\mathcal{C}_{\mathcal{A}}$, which vertices are identified by an overlined number given in the top left corner. Consider only edges represented by a solid line. Let us consider a block B composed of vertices $\overline{7}$, $\overline{9}$ and $\overline{11}$. The only input and output edges of B are edges from $\overline{5}$ and towards $\overline{13}$ respectively. The vertices $\overline{5}$ and $\overline{13}$ assign the same abstract value, `E`, to s. It means that we can remove the block B and add an

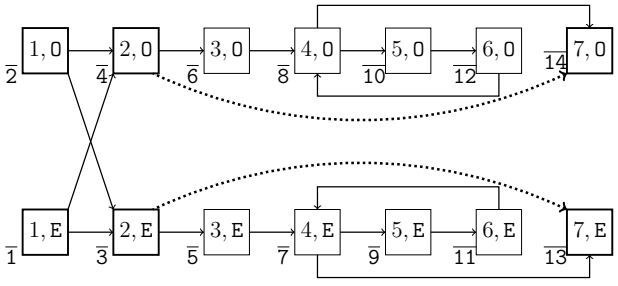

Fig. 9.

pruningEA(G)
Input: $G=[G_1,\ldots,G_w]$ - ASG_E
Output: pruned G
foreach $i \in [1..w]$
  G'_i=**pruningSA**(G_i);
endf
return $G=[G'_1,\ldots,G'_w]$;

Fig. 10.

edge from $\overline{5}$ to $\overline{13}$. Analogously, it is possible to remove a block C, composed of vertices $\overline{8}$, $\overline{10}$ and $\overline{12}$, and to add an edge from $\overline{6}$ to $\overline{14}$. In a similar way we can remove vertices $\overline{5}$ and $\overline{6}$ and add the edges from $\overline{3}$ to $\overline{13}$ and from $\overline{4}$ to $\overline{14}$. These edges are represented by dotted lines in Fig. 9. The remaining vertices, $\overline{1}$, $\overline{2}$, $\overline{3}$ and $\overline{4}$[4], represented in bold in Fig 9, correspond to the statements 1 and 2, which form the *abstract slice*, Q, given in Fig. 4.

Abstract slices obtained this way can be larger than they are expected to be. In the worst case, we do not remove any instruction from the starting program P.

In order to extract *abstract conditioned slices*[5] we refine *abstract states*. This refinement permits us to construct ASG_E containing more information and which captures the relationship between input and properties of variables of interest. At this point the *extended approach pruning* (Fig. 10), takes as input G, and applies the method **pruningSA** (Fig. 8) to all subgraphs G_i, $i \in [1..w]$ of G. Each of these applications returns a pruned subgraph, G'_i, which permits us to construct an *abstract slice*, called *partial abstract slice*, containing only statements of P which vertices appear in G'_i. From the complexity point of view we can note that, once we have the graph, the complexity of both the algorithms is linear w.r.t. the dimension of the graph. Indeed, the most difficult part of our approaches is the construction of ASG. Note that the ASGs used in our examples are constructed manually. In order to completely automate our approaches, we should use some automatic tool for the constructions of our ASGs.

[4] Vertices $\overline{13}$ and $\overline{14}$ represent the end of execution.
[5] Recall that abstract dynamic slicing is a particular case of abstract conditioned slicing.

Yorsh et al. [16] presented a method for static program analysis that leverages tests and concrete program executions. They introduce *state abstractions* which generalize the set of program states obtained from concrete executions and define a notion of *abstract graphs*, similar to ours. Furthermore, they use a theorem prover to check that the generalized set of concrete states covers all potential executions and satisfies additional safety properties, and they use these results to construct an approximation of their *abstract graphs*. The relation between [16] and our work should be further analyzed, in order to use their method for an automatic construction of our graphs (ASG).

6 Conclusion and Future Work

This paper introduces a notion of *abstract slicing*. Abstract slicing is very useful when we want to determine which statements of original program affect some particular properties of variables of interest. We have defined the *Abstract Formal Framework*, that is an extension of a structure used for representation of all existing forms of slicing and for their comparison [1,2]. Within our framework it is possible to represent even the abstract forms of slicing. Furthermore, we formally proved that the three abstract forms are *weaker* than the corresponding standard forms.

Moreover, we have made the first steps towards implementation, and we introduced two novel approaches (*simple* and *extended*) for the extraction of *abstract slices*. We illustrated their application with an example showing the first approach, and informally explaining how this approach is used for the extended one. The automation of these approaches deserves further research. The main challenge is an automatic construction of ASG, and it might be done by using a theorem prover controlling whether an edge between two vertices exists or not. Unfortunately, this construction may introduce some edges that do not correspond to real cases, so the *abstract slices* may not be precise enough. We will try to solve the problem of automatic construction of ASG by using some sophisticated method, such as [16].

We think that the abstract slicing can be seen as a particular technique of *code obfuscation*. Suppose we have a program P and we want the obfuscated program $\mathcal{O}(P)$ to preserve properties of interest for some particular variables of P, but to hide everything else. For instance, if $x = 10$ and $y = 0$ in P, $\mathcal{O}(P)$ could tell us that $x \leq 20$ but it should not public neither its real value nor any information regarding y. In this case abstract slices of the original program w.r.t. a specified criterion can be very useful. Abstract slicing would be even more applicable to code obfuscation if we defined a notion of *abstract amorphous slicing*, that would permit us even to modify statements, and not to eliminate them only. The relationship between abstract slicing (standard or amorphous) and code obfuscation deserves further research. Moreover, we think there is a strong connection between *non-interference* and program slicing, and therefore between *abstract non-interference* [10] and abstract slicing. A formalization of this relationships may be one more object of the future work, and may lead to an implementation of abstract non-interference certifications for program security.

References

1. Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, Á., Korel, B.: A formalisation of the relationship between forms of program slicing. Sci. Comput. Program 62(3), 228–252 (2006)
2. Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, Á., Korel, B.: Theoretical foundations of dynamic program slicing. Theor. Comput. Sci. 360(1), 23–41 (2006)
3. Binkley, D.W., Gallagher, K.B.: Program slicing. Advances in Computers 43 (1996)
4. Canfora, G., Cinitile, A., De Lucia, A.: Conditioned program slicing. Information and Software Tech. 40, 11–12 (1998)
5. Cimitile, A., De Lucia, A., Munro, M.: A specification driven slicing process for identifying reusable functions. Journal of Software Maintenance 8(3), 145–178 (1996)
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of Conf. Record of the 4th ACM Symp. on Principles of Programming Languages (POPL 1977), pp. 238–252. ACM Press, New York (1977)
7. De Lucia, A.: Program slicing: Methods and applications. In: IEEE International Workshop on Source Code Analysis and Manipulation (2001)
8. Field, J., Ramalingam, G., Tip, F.: Parametric program slicing. In: POPL 1995: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 379–392. ACM, New York (1995)
9. Gallagher, K.B., Lyle, J.R.: Using program slicing in software maintenance. IEEE Trans. on Software Engineering 17(8), 751–761 (1991)
10. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: Parameterizing non-interference by abstract interpretation. In: Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2004), pp. 186–197. ACM-Press, New York (2004)
11. Hong, H.S., Lee, I., Sokolsky, O.: Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In: Proc. of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005), pp. 25–34. IEEE Comp. Soc. Press, Los Alamitos (2005)
12. Korel, B., Laski, J.: Dynamic program slicing. Information Processing Letters 29(3), 155–183 (1988)
13. Mastroeni, I., Zanardini, D.: Data dependencies and program slicing: From syntax to abstract semantics. In: Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2008), pp. 125–134 (2008)
14. Tip, F.: A survey of program slicing techniques. J. of Programming Languages 3, 121–189 (1995)
15. Weiser, M.: Program slicing. IEEE Trans. on Software Engineering 10(4), 352–357 (1984)
16. Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: better together! In: ISSTA 2006: Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp. 145–156. ACM, New York (2006)