Abstraction refinement for the analysis of software product lines

Abstraction Refinement for the Analysis of Software Product Lines*

Ferruccio Damiani[1], Reiner Hähnle[2], and Michael Lienhardt[1]

[1] University of Torino, Torino, Italy
{ferruccio.damiani, michael.lienhardt}@unito.it
[2] University of Darmstadt, Darmstadt, Germany
haehnle@cs.tu-darmstadt.de

Abstract. We generalize the principle of counter example-guided data abstraction refinement (CEGAR) to guided refinement of Software Product Lines (SPL) and of analysis tools. We also add a problem decomposition step. The result is a framework for formal SPL analysis via guided refinement and divide-and-conquer, through sound orchestration of multiple tools.

1 Introduction

A *Software Product Line* (SPL) is a set of similar programs, called *variants*, with a common code base and well documented variability [23]. An SPL can be described by a triple consisting of a feature model, an artifact base, and configuration knowledge. The *feature model* defines the set of variants in terms of *features*: each feature represents an abstract description of functionality and each variant is identified by a set of features, called a *product*. The *artifact base* provides language dependent reusable code artifacts that are used to build the variants. *Configuration knowledge* connects feature model and artifact base by describing how to derive variants from the code artifacts given the products.

Tool-based analysis of software [12] is becoming more and more feasible and, therefore, common. This includes functional verification [1], resource analysis [2], safety verification [15], information flow [36], deadlock detection [30], to name just a few. It is still a challenge, however, to lift such analyses from the level of individual variants to whole SPLs. There are lifting approaches that, by making analyses and tools variability aware (i.e., to operate directly on the code of the SPL, not on the code of the variants) work for type systems [26, 24] or lightweight static analyses [17]. For more complex scenarios, such as formal verification, relatively restrictive assumptions must be made [32] (see also [18, 25]). There is no general theory of lifting software analysis from individual products to SPLs [42].

* This work has been partially supported by: EU Horizon 2020 project HyVar (www.hyvar-project.eu), GA No. 644298; ICT COST Action IC1402 ARVI (www.cost-arvi.eu); Ateneo/CSP D16D15000360005 project RunVar (runvar-project.di.unito.it); project FormbaR (formbar.raillab.de), Innovationsallianz TU Darmstadt–Deutsche Bahn Netz AG.

An alternative to making the analyses and the tools variability aware, is to generate, for a given SPL, a *meta variant* or *variant simulator* (see, e.g., [45]). This is an artifact, expressed in the same language as the variants are written in, that takes as input any product and simulates the behavior of the corresponding variant. A meta variant has the advantage that it can be analyzed with standard tools for the implementation language of its variants. To ensure that this approach is efficient, *variability encoding* (i.e., the process of transforming an SPL into a meta variant) must avoid to duplicate code that is common to different variants. Depending on the given SPL, its meta variant can be significantly more complex than any of the variants, challenging the capabilities of available tools [43]. Indeed, it has has not yet been demonstrated that variability encoding provides a scalable approach to family-based analysis of large SPLs.

In this paper we present a novel and systematic approach that permits to apply software analyses to the meta variant of an SPL. We take our cue from *Counter Example-Guided Abstraction Refinement* (CEGAR) [22], a well-known and highly successful verification strategy to handle programs that are too complex to be verified directly. We generalize the CEGAR principle to guided refinement of SPLs and of analysis tools. We also add a problem decomposition step. The result is a framework for formal SPL analysis via guided refinement and divide-and-conquer, through sound orchestration of multiple tools.

Paper organization. In Sect. 2 we briefly recall the main approches to implement SPLs and introduce the running example of the paper. In Sect. 3 we recall the CEGAR principle and explain our proposal to generalize it to the refinement of SPLs and of tools. In Sect. 4 we recap the workflow of the running example and outline how our framework can be instantiated to other scenarios. In Sect. 5 we discuss related work and in Sect. 6 we conclude.

2 Implementation of Software Product Lines

Currently, there exist three main approaches to implement SPLs [40]: *annotative approaches* expressing negative variability (all variants are represented by a single artifact); *compositional approaches* expressing positive variability (features are associated to artifacts, possibly describing refinements to a base artifact); and *transformational approaches* expressing both positive and negative variability (feature combinations are associated to artifacts describing changes to a base artifact to obtain other system variants).

A prominent example of an annotative approach is based on C preprocessor directives (`#define FEATURE` and `#ifdef FEATURE`). *Delta-Oriented Programming* (DOP, see [13, 38] and [6, Sect. 6.6.1]) is a flexible transformational approach in which the artifact base consists of a *base program* (that might be empty or incomplete) and of a set of *deltas*, which are containers of modifications to a program (e.g., for Java programs, a delta can add, remove or modify classes and interfaces), while configuration knowledge associates to each delta an *activation condition* over the features and specifies an *application ordering* between deltas.

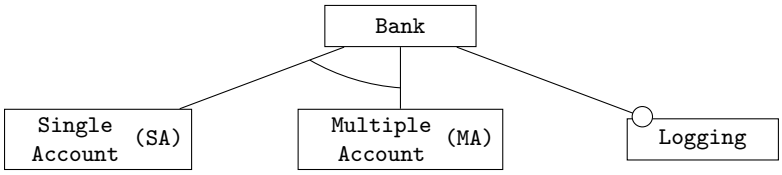

Fig. 1. Visual representation of the feature model of the Bank Account SPL example

```
data Operation = Withdraw(Int) | Deposit(Int);
class Client() implements IClient { }
class Account() implements IAccount {
 Int amount;
 Int getAmount() { return this.amount; }
}

class Bank() implements IBank {
 List<IClient> clients;
 Unit applyOperation(Operation op) { ... }
```

```
 Bool newOperation(Operation op) {
  Bool check_needed = case op {
   Withdraw(i) => i > 10000 ;
   _ => False ;
  };
  Bool apply_accepted = case check_needed {
    True => this.checkAccounts();
    _ => True;
  };
  if(apply_accepted)
    this.applyOperation(op);
  return apply_accepted;
}}
```

Fig. 2. Base Program

DOP supports the automatic generation of variants based on a selection of features: once a user selects a product, the corresponding variant is derived by applying the deltas with a satisfied activation condition to the base program according to the application ordering. DOP can be seen as a generalization of *Feature-Oriented Programming* (FOP) (see [11] and [6, Sect. 6.1]), a compositional approach to SPL implementation, where deltas correspond one-to-one to features and do not contain remove operations [39].

Our running example is a simple product line modeling a bank with different features, depicted in Fig. 1. The feature `Single Account` (or `SA`) associates one account with each client of the bank, while feature `Multiple Account` (or `MA`) allows a client to maintain several accounts. Finally, the feature `Logging` adds logging capabilities to the banking operations. Features `SA` and `MA` are *alternative* (i.e., exactly one of them must be selected), while feature `Logging` is *optional*. The code base of our example, presented in Figs. 2–5, is written in the modeling language ABS [35], which realizes DOP.

Fig. 2 contains the *base program* that implements the core functionalities of our example. The data type `Operation` describes the possible banking operations, `Withdraw` and `Deposit`, respectively for withdrawing or depositing a specified amount. The `Client` class is empty, as its content depends on whether feature `SA` or `MA` is selected, while the `Account` class, that implements an account, simply stores the balance of the account.

The `Bank` has a list of clients and declares three methods: `applyOperation` performs a banking operation in the bank, without any check; `newOperation` is a wrapper around `applyOperation` that executes some checks in case the

```
delta dSA {
 modifies class Client {
  adds IAccount account;
  adds IAccount getAccount() { return this.account; }
 }
 modifies class Bank {
  adds Bool checkAccounts() {
   List<IClient> tmp = this.clients;
   Int total_amount = 0;
   while(!isEmpty(tmp)) {
    total_amount = total_amount + head(tmp).getAccount().getAmount();
    tmp = tail(tmp);
   }
   return total_amount > 1000000;
}}}
```

Fig. 3. Delta for the `SA` feature

```
delta dMA {
 modifies class Client {
  adds List<IAccount> accounts;
  adds List<IAccount> getAccounts() { return this.accounts; }
 }
 modifies class Bank {
  adds Bool checkAccounts() {
   List<IClient> tmp1 = this.clients;
   Int total_amount = 0;
   while(!isEmpty(tmp1)) {
    List<Account> tmp2 = head(tmp1).getAccounts();
    while(!isEmpty(tmp2)) {
     total_amount = total_amount + head(tmp2).getAmount();
     tmp2 = tail(tmp2);
    }
    tmp1 = tail(tmp1);
   }
   return total_amount > 1000000;
}}}
```

Fig. 4. Delta for the `MA` feature

operation is a withdrawal of a large amount of money; finally, `checkAccounts` performs the checks and is not part of the base program, as its implementation entirely depends on the selected features.

Fig. 3 presents the delta `dSA` implementing the `SA` feature. Here, the class `Client` is defined, and simply contains an account (with a getter method). The method `checkAccounts` of the class `Bank` is also implemented, and simply iterates over all the accounts of the bank, to ensure that its overall balance is big enough to allow the requested withdrawal.

Fig. 4 presents the delta `dMA` implementing the `MA` feature. Here, the class `Client` contains a list of accounts. The implementation of the `checkAccounts` method still iterates over all the accounts of the bank to check that its overall balance is large enough, but to do so, it now contains an inner loop that iterates over all the accounts of a client.

```
delta dLog {
 modifies class Bank {
  modifies Bool newOperation(Operation op) {
   print("Managing the new operation \"" + op + "\"");
   Bool result = original(op);
   if(result) print("\tOperation successful");
   else print("\tOperation Failed")
   return result;
}}}
```

Fig. 5. Delta for the `Logging` feature

Fig. 5 contains the delta `dLog` that implements the feature `Logging`. This delta redefines the method `newOperation` of the class `Bank`, surrounding the original implementation (modeled with the keyword `original` in place of the method call) with two calls to `print`. These calls simply register which operation was requested and whether it was performed.

Finally, the configuration knowledge required to describe the Bank Account SPL is straightforward and we omit the corresponding ABS declaration—it simply specifies that each delta is activated exactly by the feature that it realizes (since for each product applying the activated deltas in any order yields the same variant, no application ordering needs to be specified).

3 Counter Product-guided Refinement

3.1 Counter Example-guided Abstraction Refinement (CEGAR)

Assume we want to establish that a property P holds for any run of a program m with an analysis tool t, denoted by $m \vdash_t P$. For example, m could be an ABS program, P a safety property saying that certain bad states are unreachable, and t might be a model checker: it can happen that $m \vdash_t P$ cannot be established because t times out or runs out of memory.

To render verification feasible, the CEGAR verification strategy (illustrated in Fig. 6) executes t not with m, but with an *abstraction* of m, written $A(m)$: for example, all datatypes are initially abstracted to booleans which greatly reduces the number of reachable states. Note that the chosen abstraction must be sound in the sense that $A(m)$ preserves all possible behaviors of m. Now we can assume that the—simplified—problem $A(m) \vdash_t P$ terminates. If $A(m) \vdash_t P$ holds, then also $m \vdash_t P$ holds (because the abstraction is sound) and we are done. If $A(m) \vdash_t P$ doesn't hold, then we extract a counter example, i.e., an input c of m such that $A(m)(c)$ violates P. If $m(c)$ violates P as well, then the counter example exhibits a real bug of m and we are done (i.e., we can try to fix the bug and restart the process). If $m(c)$ does not violate P, then we use c to refine A to a more precise abstraction A' so that $A'(m)(c)$ does not violate P, and we

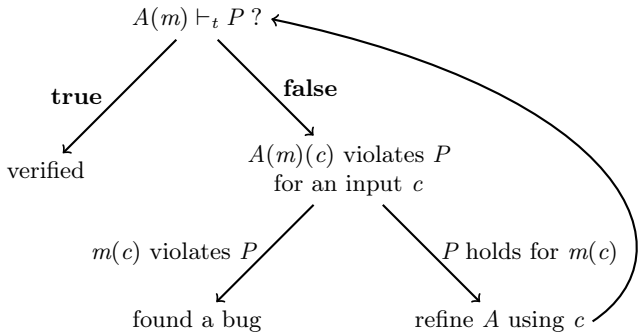

Fig. 6. Work flow of CEGAR

re-enter the CEGAR loop with the refined abstraction.[3] A concrete example of a CEGAR-style refinement is presented below in Sect.3.3.

3.2 Counter Example-guided Product Line Refinement (CEGPLR)

In the context of Software Product Lines, another kind of refinement can be considered: CEGAR looks at one program at a time and performs refinement on that program's data abstraction, however, Software Product Lines add the dimension of having to analyze different program variants at the same time. We observe that the meta variant of an SPL is compatible with the CEGAR approach in the following sense: A meta variant of an SPL by definition encompasses the behavior of each of its variants. Hence, a meta variant constitutes a behavioral abstraction of each variant or set of variants of a given SPL. Consequently, a meta variant might be refined to the behavior embodied in any subset of its variants.

For instance, the SPL presented in Sect. 2 defines four different variants identified by the four following products: {`Bank`, `SA`}, {`Bank`, `SA`, `Logging`}, {`Bank`, `MA`} and {`Bank`, `MA`, `Logging`}. In this context, one can apply a CEGAR-like iteration to the SPL: first one runs an analysis tool t on an abstraction that comprises all variants. If t succeeds then, as with CEGAR, we are done. Otherwise, a counter example consisting of an input c and a *subset of the variants* exhibiting the error for c can be extracted. This triggers a *decomposition step* that consists of splitting the input SPL into two parts: one that has c as a possible counter example, and one that has not. Both parts can then be analyzed independently, as they don't exhibit the same behavior. If the part where c is no counter example has no other counter example, then that part of the SPL is verified.

[3] This abstract description of CEGAR leaves many issues open: how to make sure that the refinement loop terminates? How to select a counter example? How to compute the refinement? On each of these questions a considerable literature exists, but this is not the focus of this paper.

```
data Product = Product(Bool fBank, fBool SA, Bool fMA, Bool fLogging);
def Bool isValid(Product p) = fBank(p) && (fSA(p) || fMA(p)) && !(fSA(p) && fMA(p));
def Bool dLogging(Product p) = fLogging(p);
...

Bool newOperation(Operation op) {
 Bool result = False;
 if(dLogging(p)) {
  print("Managing the new operation \"" + op + "\"");
  result = this.newOperationCore(op);
  if(result) print("\tOperation successful");
  else print("\tOperation Failed");
 } else {
  result = this.newOperationCore(op);
 }
 return result;
}

Bool newOperationCore(Operation op) { ... }
```

Fig. 7. Excerpt of meta variant for the Bank SPL

To illustrate this approach to Product Line Refinement with a concrete example, let us consider the Bank SPL presented in Sect. 2, simply called L from now: assume we want to ensure the property P stating that the execution time of the `newOperation()` method is at most linear in the number of accounts in the bank. The analysis tool we consider is SACO [2], which is a state-of-art cost analysis tool that abstracts every non-boolean datatype by its size.

For the abstraction of the variants of an SPL, we use its meta variant, i.e., a program that contains each behavior in each variant of L (cf. Sect. 1). There are different techniques to obtain it, and here we use the 150% test model of [29, 31] which is an instance of a sound *variability encoding* [45]. An excerpt of our meta variant is depicted in Fig. 7. The first two lines encode product selection and what a valid product is. The third line relates the code delta `dLog` to the feature Logging. This has to be completed for the remaining features and is not necessarily one-to-one like here. The meta variant selection mechanism can be seen in the method `newOperation()`. When the logging delta is requested, then the main `if` condition executes the code from Fig. 5, otherwise the core product version of the method is executed (that version is stored in a new method with a new name, to disambiguate the calls).

Running SACO on our meta variant yields an interesting result: it validates the property P when the feature `MA` is not selected, but fails to prove it when `MA` is selected. An analysis of the obtained counter example shows that during its abstraction step, SACO replaced lists by integers corresponding to their size, thus ignoring essential information about the accounts when the feature `MA` is activated, as these are stored inside a list of lists. In the following decomposition step the meta variant is split in two parts. The first of these contains all variants that do not have the behavior required by feature `MA`. We write this as $L[\{\texttt{SA}\}, \{\texttt{SA}, \texttt{Logging}\}]$ and call it a *partial meta variant*. SACO guarantees that its two variants validate P. The second partial meta variant, where `MA` is

activated (written $L[\{\texttt{MA}\}, \{\texttt{MA}, \texttt{Logging}\}]$), does not have this guarantee. Of it we know that to prove P, we must not abstract away the list of lists structure.

The general form of a partial meta variant is $L[F_1, \ldots, F_n]$ where L is the SPL from which the meta variant is generated and $F_1, \ldots, F_n$ are the products of L available in this meta variant.

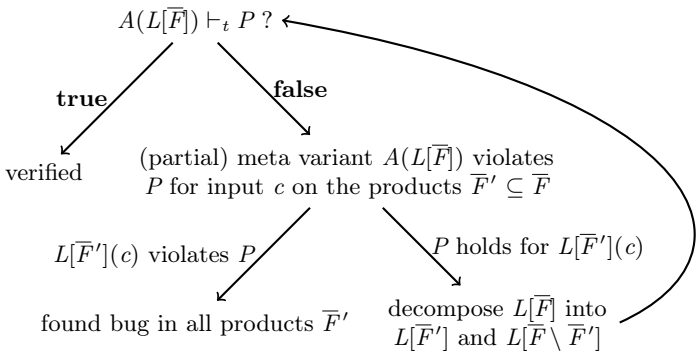

Fig. 8. Work flow of CEGPLR

We can now define (illustrated in Fig. 8) a CEGAR-like loop for refining and decomposing a Software Product Line. The loop is started with the (full) meta variant of the input SPL, i.e. initially $\overline{F} = F_1, \ldots, F_n$ are all the products of the SPL. Note that we work with an abstraction $A(L[\overline{F}])$ of the meta variant, implying that standard CEGAR and SPL refinement can be interleaved.

Like before, if we manage to verify the property, then we are done. If not, then the counter example does not only consist of a concrete input c, but also of a set of products $\overline{F}'$ exhibiting this counter example. Like in CEGAR one checks now whether the counter example is real: we test it against the partial meta variant $L[\overline{F}']$. If $L[\overline{F}'](c)$ violates P, we found a bug. Otherwise, we attempt to refine the current meta variant $L[\overline{F}]$ into $L[\overline{F} \setminus \overline{F}']$, i.e., we assume that the selected features were critical for the counter example to manifest itself, and, therefore exclude them.[4] If we manage to verify at some point $A(L[\overline{F} \setminus \overline{F}']) \vdash_t P$ for some $\overline{F}' \subseteq \overline{F}$, then we have refined the original verification problem to $L[\overline{F}']$. We call this process *counter example-guided product line refinement* (CEGPLR).

In fact, CEGPLR goes beyond CEGAR, because it provides not only a problem refinement, but also a *problem decomposition* (into $L[\overline{F}']$ and $L[\overline{F} \setminus \overline{F}']$). Therefore, it is a combined abstraction refinement and *divide-and-conquer* approach.

[4] This is a coarse-grained refinement step. Alternatively, one could branch into $|\overline{F}'|$ many refinements of the form $L[\overline{F}'']$ with $\overline{F}'' \subseteq \overline{F}'$.

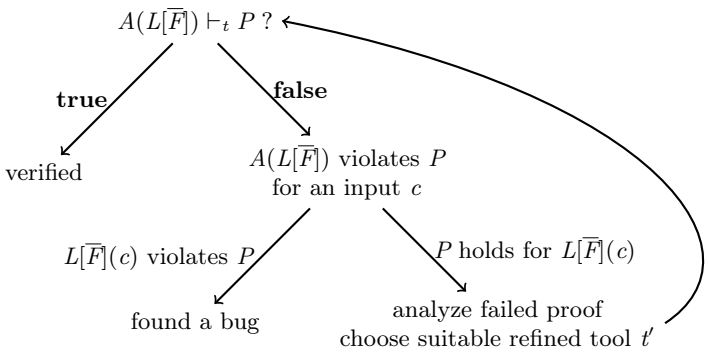

Fig. 9. Work flow of CEGTR

3.3 Tool Refinement

Existing CEGAR-like approaches work with a single verification or analysis tool, for example, a model checker or symbolic execution, but this constitutes no principal limitation. In fact, there is growing evidence that huge efficiency gains can be obtained from systematic combination of different analysis tools [5, 14, 20]. One can even hypothesize that *only* the systematic combination of different tools and methods will make it feasible to attack complex problems [12]. Hence, in addition to abstraction and product line refinement, we suggest *tool refinement*. This term is justified, as long as the refined tool analyzes at least as many behaviors as the old one.

In Fig. 9 we present yet another variant of the CEGAR loop (Fig. 6), this time based on tool refinement. The difference lies in the analysis of the failed proof. Instead of looking for ways to refine the abstraction A or the partial meta variant $L[\overline{F}]$, we now look for a verification tool t' that refines the analysis performed by t in a manner such that $A(L[\overline{F}]) \vdash_{t'} P$ (or a refinement thereof) becomes provable. Obviously, this is in general a step that requires deliberation by an expert, in contrast to CEGAR, where abstractions are computed automatically. Nevertheless, it is beneficial: (i) one obtains guidance in choosing an appropriate tool, (ii) behavioral refinement of the tools preserves overall soundness, and (iii) the input and instrumentation of tool t' can be obtained from A and $L[\overline{F}]$. The third point is, in principle, automatable.

We illustrate tool refinement with our running example. In the previous section, SACO failed to analyze the partial meta variant $L[\{\texttt{MA}\}, \{\texttt{MA}, \texttt{Logging}\}]$: SACO abstracted away lists into integers, and was unable to find a bound for the nested loop in `dMA` (Fig. 4). SACO can, in principle, deal with nested loops, but it has limited support for reference types (like lists) which are abstracted by their size. For this reason, the tool doesn't know enough about the structure of type `List<IAccount>` to perform the analysis. The tool also cannot express separation conditions (e.g., that the `Account` objects in a list are unaliased).

The abstraction of SACO cannot be further refined and we did the product line refinement already, so the only possibility now is to refine the tool. In the paper [3] a formal link between resource analysis tools and formal verification tools is described. This makes it possible to use a formal verification tool such as KeY [1] to reason about resource properties. Of course, KeY is an interactive tool and might require input from a verification expert. But thanks to product line refinement, we managed to reduce the problem already. In addition, all the invariants derived by SACO are automatically imported into KeY, such that only the *additional* annotations to prove the correctness of the meta variant $L[\{\texttt{MA}\}, \{\texttt{MA}, \texttt{Logging}\}]$ need to be supplied. A further reason to use the KeY tool in the experiment is that it can be instrumented with user-defined data type abstractions [46].

We first attempt to prove $A(L[\{\texttt{MA}\}, \{\texttt{MA}, \texttt{Logging}\}]) \vdash_{\text{KeY}} P$, where A is the abstraction embodied in SACO. As A still abstracts the inner `Account` lists away, this fails in KeY as well, but now we can again enter the CEGAR loop and refine the abstraction, based on the analysis above: we now model lists precisely, but we can still abstract completely away from `Account`. With this new abstraction, denoted A', the statement $A'(L[\{\texttt{MA}\}, \{\texttt{MA}, \texttt{Logging}\}]) \vdash_{\text{KeY}} P$ was successfully proven. Note that A' simplifies the verification problem considerably compared to the normal KeY verification workflow, because, in contrast to CEGAR, KeY is usually started with no abstraction at all.[5] The integration of KeY into a CEGAR framework allows KeY to profit from a previously computed abstraction.

3.4 Other Abstractions

Behavioral Abstraction. CEGAR is based on datatype refinement, but with SPL and tool refinement we introduced behavioral refinement already. Therefore, it is natural to look at further possibilities for the *behavioral abstraction* of a given program. For example, if we are interested in deadlocks (i.e., we are out to prove deadlock-freedom), it might be useful to abstract a program away to merely its call and synchronization points and completely ignore datatypes.[6] Even more drastic abstractions, e.g., occurring in type-based analyses [30], abstract completely away from object creation. This fits perfectly well into our framework. We simply extend the meaning of A to include behavioral abstractions as well.

Property Abstraction. It is also possible to abstract or refine the *property* to be proven. Please note that both directions can be useful. If we have proven P, by abstraction soundness, we have also proven $A(P)$. In this case, it might be worth trying to prove a *stronger* property. An example of a situation, where this is useful is given in Sect. 4.3 under *Formal Verification*.

Vice versa, if we do not manage to prove P, a possible strategy is to prove a weaker property $A(P)$. For example, in Sect. 3.3 we proved with KeY a linear

[5] In the standard workflow of KeY abstractions are computed on demand and are mainly used for loop invariant generation and state merging.

[6] Another way to view this is to abstract all data to a single value.

bound for $L[\{\texttt{MA}\}, \{\texttt{MA}, \texttt{Logging}\}]$. However, this requires a suitable modification of the loop invariant. If we weaken P to prove just termination with no concrete bound, then it is sufficient to provide *termination witnesses* for both loops which are completely straightforward: `length(clients)-length(tmp1)` and `length(accounts)-length(tmp2)`, respectively.

4 Abstraction Refinement for Software Product Lines

4.1 An Abstraction Layer in the Analysis of SPLs

In the previous section we proposed two new CEGAR-like loops in the context of static analysis of SPLs: CEGPLR (Fig. 8) realizes SPL refinement and decomposition, based on the observation that the meta variant of a product line constitutes a behavioral abstraction of each partial meta variant and, in particular, of each single product variant; CEGTR (Fig. 9) realizes refinement of the underlying analysis tool with a tool that can distinguish more behavior. In addition, it can also be useful to abstract or refine the properties to be proven and to work with behavioral abstractions (not mere data abstractions) of the system under verification.

The central role that is played by abstraction and refinement, both data-level and behavioral, both of the target system and the target property, suggests to maintain an explicit configuration and abstraction layer when analyzing SPLs to achieve a clean and flexible separation between the problem space and the solution space, see Fig. 10. To work out the details and to formalize such an abstraction layer is the topic of future work.

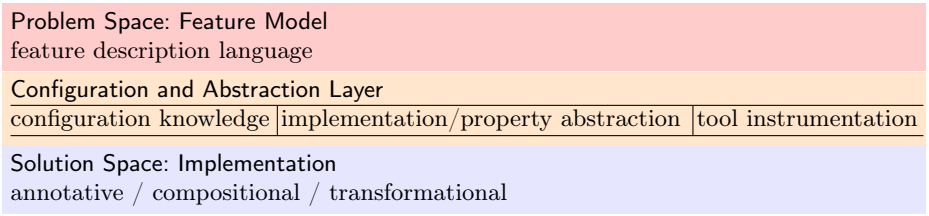

Fig. 10. SPL implementation with explicit abstraction layer

4.2 Workflow in Abstraction Refinement for SPL

It is worth to recap the workflow of our example in Sects. 3.2 & 3.3: from a failed attempt to prove a linear worst-case runtime bound with the tool SACO we decomposed via SPL refinement the problem into two partial meta variants and showed the desired property for all products in one of them (Sect. 3.2). No further abstraction refinement in SACO is possible, so the only option (except to

weaken the targeted property) was to refine the tool. The verification tool KeY offers more precision than SACO. It was instrumented with the data abstraction of SACO and the invariants computed by it. After a standard CEGAR step, KeY managed to prove the desired property (Sect. 3.3).

It is worth to note that (i) after the first refinement step, the exhaustion of other options drove the choice to perform tool refinement and (ii) that the output of the analysis in the first step provided the instrumentation of the next tool in the chain. This suggests that our framework is suitable to orchestrate the combination of static analysis and verification tools that work at different levels of precision.

For our example, only one refinement loop of each kind was necessary, but this is not true in general. For example, with a larger product line, after refinement abstraction in KeY, probably another round of product line refinement makes sense. It would be premature to speculate about concrete meta refinement loops while a robust implementation of our framework is lacking, so we refrain from it. Having said that, it seems a good idea to always attempt to refine and decompose the product line as much as possible.

4.3 More Usage Scenarios

In the previous section we illustrated our framework with a usage scenario about resource analysis. In fact, our approach is applicable to a wide range of analyses and we want it to be understood as a general framework for the sound and systematic combination and orchestration of software analysis tools. To substantiate this claim, we instantiate our framework with three more scenarios. In each case we assume that we have an SPL over ABS programs specified with DOP following [21]. While this is not necessary in general, it makes it possible in what follows to provide concrete examples of analysis methods and tools.

Feature Interference. With feature interference we mean feature interaction within an SPL that has undesired effects. It is a practically important and intensely studied problem [34]. Denote with $f \lightning f'$ that features f and f' from a given valid product F interfere with each other, for example, they both have write access to the same memory location. To analyze a given SPL for feature interference, one may start with an obvious, but coarse abstraction: Assume that for any method m required to implement $f \in F$ and m' required for $f' \in F$, such that both m and m' share a critical resource r, it is the case that m can never be executed in parallel to m' (where both $m = m'$ and $f = f'$ is possible). This is a typically sufficient criterion to exclude feature interference.

As a first verification tool we choose a may-happen-in-parallel (MHP) analysis: the predicate $MHP(m, m')$ holds for a given ABS program if it contains methods m, m' that can possibly be executed in parallel. An efficient overapproximation of MHP is available for ABS [4]. Now we enter the product line refinement loop of Fig. 8, where P is the absence of feature interference, t is MHP, and A the not-in-parallel abstraction of the meta variant $L[\overline{F}]$. As most features tend not to interact, we can assume that the CEGPLR loop refines and

decomposes the problem into a much smaller partial meta variant $L[\overline{F}']$, where absence of feature interference was proven for $L[\overline{F} \setminus \overline{F}']$.

An analysis of the failed proof for $L[\overline{F}']$ now might show that certain methods m, m' actually *do* interfere, but not in a safety-critical manner. This is not provable with MHP, but one may use deductive verification with KeY instead. To this end, one refines P so as to express that for any m, m' such that $\mathrm{MHP}(m, m')$ holds, their common resources r satisfy a safety invariant. It is possible to encode this property in a program logic with the help of self composition [27] and use KeY to prove it. However, one might abstract away from most datatypes in that proof, because they are likely to be irrelevant for feature interaction. Hence, we would instrument KeY to implement a CEGAR loop over symbolic execution with abstraction [16, 46].

Formal Verification. Deductive verification tools (e.g., KeY [1]) as well as safety verification tools (e.g., CPAchecker [15]) have impressive, yet complementary strengths. KeY was used for functional verification of SPL's using variability encoding [43], but it quickly becomes very expensive in terms of runtime and user interaction [19]. This indicates that variability encoding is not a scalable strategy for formal verification of SPLs.

Instead, one could start formal verification of a property P for an SPL L with a CEGAR-based safety verification tool [16], where P is abstracted to a weaker property $A(P)$ that is expressible in it and and the initial program is of the form $\mathit{Boolean}(L[\overline{F}])$ (where *Boolean* abstracts all data to booleans). A combination of CEGAR and CEGPLR decomposes and reduces the problem to a partial meta variant $L[\overline{F}']$ and computes a refinement $A'(L[\overline{F}'])$ from where no further progress seems possible. Only then one uses a deductive verification tool such as KeY, instrumented with A'. Once $A(P)$ has been shown for some $A''(L[\overline{F}'])$, one can perform property refinement from $A(P)$ to P, followed by further product line and abstraction refinement loops. This scenario shows that it can make perfect sense to (i) work with different abstractions for programs and properties and (ii) not just abstract from a property, but also refine it.

Information Flow. Information flow control is the problem to analyze whether a program allows an attacker to deduce information about secret values by manipulating its public interface. There is a wide variety of analysis tools and methods for this problem with complementary strengths: type-based approaches [37] and lightweight static analyses [33] scale well, even to SPL [17], but yield many false positives and can only express limited security policies. Deductive approaches [27, 41] are expensive and often require manual annotation. As a consequence, information flow is a natural usage scenario for our framework and it can be developed in a similar manner as the previous scenarios.

5 Related Work

There are a number of verification approaches that decompose or transform a complex analysis problem such that different tools can be used in combination

to solve it. CPAchecker [15] is a flexibly configurable tool framework for fully automatic verification of safety properties that allows to integrate other tools in a sound manner. Specifically, Beyer & Lemberger [16] applied CEGAR in the context of symbolic execution within CPAchecker. However, it is not designed to express complex functional properties. Ahrendt et al. [20] use the result of a partial verification attempt of a given program to generate an optimized runtime assertion checker that only monitors those properties that could not be proven. Küsters et al. [36] combine static analysis and deductive verification for information flow proofs: they transform the given program and prove with KeY preservation of behavior, then use the static analyzer on the simplified program. This corresponds to manual computation of a suitable program abstraction, whereas we propagate abstraction refinement. None of these papers is concerned with the analysis of SPLs.

Clafer [9] is a modeling language that is designed as an extension of Alloy and has a unified representation of features as well as OO models. It has been used to model and analyse Software Product Lines [8], however, it is not directly connected to executable code.

Batory [10] developed a theory of modular composition and decomposition of software that has been used also in the context of SPLs and that has been extended to verification proofs. It is also based on refinement, but requires a theoretical framework that makes it not straightforward to apply to existing languages and tools. To the best of our knowledge, it does not contain a CEGAR-like strategy. Proof composition [44] relies on creating partial correctness proofs for certain features that are then combined into proofs for a desired product. However, this approach becomes problematic when properties of feature implementations depend on each other.

The 150% model technique [29, 31] originates from model-based testing and was then also employed in software analysis, e.g., [7, 43]. All of these approaches are an instance of *variability encoding*, as classified and formalized in [45].

Bodden et al. [17] lift static analysis of control flow properties from product variants to software product lines, essentially by a form of variability encoding into a somewhat more expressive static analysis framework. This approach, however, does not work for more complex properties.

Independently of our work, Dimovski & Wąsowski [28] recently implemented what seems to be the first product line refinement approach for LTL model checking. It follows the same pattern as ours, employing a notion of *partial meta variant* containing all nodes and transitions of the included variants. Like in our approach, the meta variant is a standard product, in their case an LTS, that allows to use the SPIN model checker. As we do, upon finding a spurious counter example, they split the meta variant, with the help of Craig interpolation.

6 Conclusion and Future Work

In this paper we drafted an SPL analysis framework based on the principle to perform as much work as possible with lightweight, efficient, and automatic

methods: this means to start analyzing product lines at a high level of abstraction, possibly with an abstract version of the targeted property. Then we apply the main lesson behind the CEGAR principle: don't throw away failed proof attempts, but carefully analyze the information contained in them to improve the analysis.

Based on the insight that a meta variant is a behavioral abstraction of each subset of its variants, we designed a CEGAR-like loop to perform *product line refinement* and, made possible through an extensional, feature-based representation of products, extended it to a *divide-and-conquer* approach that provides product line decomposition. Crucially, even when neither CEGAR nor CEGPLR are successful, this is not the end of the line: one refines the analysis tool and uses a more precise, but also more heavyweight method, but benefits from the refinement and decomposition made in the previous steps. Indeed, all four usage scenarios we discussed—resource analysis, feature interference, formal verification, information flow—offer a variety of analysis tools working at differing levels of abstraction. The concept of *tool refinement* soundly integrates these tools, where a CEGAR-style refinement analysis guides the *selection* of the chosen tool and helps to *instrument* it.

Overall, our framework implements a version of the *subsidiarity principle* in the realm of software analysis: a subtask should be solved at the highest possible level of abstraction, with the least expensive method.

CEGAR loops are normally part of a single, fully automated tool, but this is an unnecessary limitation. Our work shows that manual abstraction refinement for guiding the selection of a new tool makes perfect sense. Another important lesson that can be drawn is that it is extremely useful to have tools that can be flexibly instrumented with data abstraction. This is the case already, for example, for CPAchecker [15] and KeY [1].

The next step is to provide a robust implementation of our framework, including a suitable abstraction layer (see Fig. 10) and to conduct larger case studies.

Acknowledgment.

The authors gratefully acknowledge the help of Antonio Flores Montoya who ran a number of experiments with SACO for us and helped with their analysis.

References

1. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification—The KeY Book: From Theory to Practice*, volume 10001 of *LNCS*. Springer-Verlag, 2016.
2. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: static analyzer for concurrent objects. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 20th Intl. Conf., Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Grenoble, France*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.

3. E. Albert, R. Bubel, S. Genaim, R. Hähnle, and G. R. Díez. A formal verification framework for static analysis—as well as its instantiation to the resource analyzer COSTA and formal verification tool KeY. *Software & Systems Modeling*, 15(4):987–1012, 2016.
4. E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. May-happen-in-parallel analysis for actor-based concurrency. *ACM Trans. Comput. Log.*, 17(2):11:1–11:39, 2016.
5. E. Albert, M. Gómez-Zamalloa, and M. Isabel. Combining static analysis and testing for deadlock detection. In E. Ábrahám and M. Huisman, editors, *Integrated Formal Methods, 12th Intl. Conf. IFM, Reykjavik, Iceland*, volume 9681 of *LNCS*, pages 409–424. Springer, 2016.
6. S. Apel, D. S. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.
7. S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In P. Alexander, C. S. Pasareanu, and J. G. Hosking, editors, *26th IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE), Lawrence, KS, USA*, pages 372–375. IEEE Computer Society, 2011.
8. K. Bak. *Modeling and Analysis of Software Product Line Variability in Clafer*. PhD thesis, University of Waterloo, 2013.
9. K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wasowski. Clafer: unifying class and feature modeling. *Software and System Modeling*, 15(3):811–845, 2016.
10. D. S. Batory. A theory of modularity for automated software development. In R. B. France, S. Ghosh, and G. T. Leavens, editors, *Companion Proc. 14th Intl. Conf. on Modularity, Fort Collins, CO, USA*, pages 1–10. ACM, 2015.
11. D. S. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6), 2004.
12. B. Beckert and R. Hähnle. Reasoning and verification. *IEEE Intelligent Systems*, 29(1):20–29, Jan.–Feb. 2014.
13. L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013.
14. D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Correctness witnesses: exchanging verification results between verifiers. In T. Zimmermann, J. Cleland-Huang, and Z. Su, editors, *Proc. 24th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering, FSE, Seattle, WA, USA*, pages 326–337. ACM, 2016.
15. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification, 23rd Intl. Conf. CAV, Snowbird, UT, USA*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.
16. D. Beyer and T. Lemberger. Symbolic execution with CEGAR. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 7th International Symposium (ISoLA), Part I, Corfu, Greece*, volume 9952 of *LNCS*, pages 195–211. Springer, Oct. 2016.
17. E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. Spl^{lift}: statically analyzing software product lines in minutes instead of years. In H. Boehm and C. Flanagan, editors, *ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI, Seattle, WA, USA*, pages 355–364. ACM, 2013.
18. R. Bubel, F. Damiani, R. Hähnle, E. B. Johnsen, O. Owe, I. Schaefer, and I. C. Yu. Proof repositories for compositional verification of evolving software systems. In *Foundations for Mastering Change (FoMaC) I*, volume 9960 of *LNCS*, pages 130–156. Springer-Verlag, 2016.

19. R. Bubel, C. Din, and R. Hähnle. Verification of variable software: an experience report. In B. Beckert and C. Marché, editors, *Pre-Proc. International Conference on Formal Verification of Object-Oriented Software (FoVeOOS), Paris, France*, 2010.
20. J. M. Chimento, W. Ahrendt, G. J. Pace, and G. Schneider. StaRVOOrS: A tool for combined static and runtime verification of Java. In E. Bartocci and R. Majumdar, editors, *Runtime Verification — 6th Intl. Conf., Vienna, Austria*, volume 9333 of *LNCS*, pages 297–305. Springer, 2015.
21. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schafer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 417–457. Springer-Verlag, 2011.
22. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, Chicago/IL, USA*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.
23. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.
24. F. Damiani and M. Lienhardt. On type checking delta-oriented product lines. In E. Ábrahám and M. Huisman, editors, *Integrated Formal Methods: 12th Intl. Conf., iFM, Reykjavik, Iceland*, volume 9681 of *LNCS*, pages 47–62. Springer, 2016.
25. F. Damiani, O. Owe, J. Dovland, I. Schaefer, E. B. Johnsen, and I. C. Yu. A transformational proof system for delta-oriented programming. In *SPLC (2)*, pages 53–60, 2012.
26. F. Damiani and I. Schaefer. Family-based analysis of type safety for delta-oriented software product lines. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change — 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece*, volume 7609 of *Lecture Notes in Computer Science*, pages 193–207. Springer, Oct. 2012.
27. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer-Verlag, 2005.
28. A. S. Dimovski and A. Wąsowski. Variability-specific abstraction refinement for family-based model checking. In M. Huisman and J. Rubin, editors, *Fundamental Approaches to Software Engineering: 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 406–423, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
29. C. Dziobek and J. Weiland. Variantenmodellierung und -konfiguration eingebetteter automotive Software mit Simulink. In H. Giese, M. Huhn, U. Nickel, and B. Schätz, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme V, Schloss Dagstuhl, Germany*, volume 2009-01 of *Informatik-Bericht*, pages 36–45. TU Braunschweig, Institut für Software Systems Engineering, 2009.
30. E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in `core abs`. *Software & Systems Modeling*, 15(4):1013–1048, 2016.

31. H. Grönniger, J. Hartmann, H. Krahn, S. Kriebel, L. Rothhardt, and B. Rumpe. View-centric modeling of automotive logical architectures. In H. Giese, M. Huhn, U. Nickel, and B. Schätz, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV, Schloss Dagstuhl, Germany*, volume 2008-2 of *Informatik-Bericht*, pages 3–12. TU Braunschweig, Institut für Software Systems Engineering, 2008.
32. R. Hähnle and I. Schaefer. A Liskov principle for delta-oriented programming. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change — 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece*, volume 7609 of *Lecture Notes in Computer Science*, pages 32–46. Springer, Oct. 2012.
33. C. Hammer, J. Krinke, and G. Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE Intl. Symp. on Secure Software Engineering (ISSSE)*, pages 87–96. IEEE, March 2006.
34. M. Jackson and P. Zave. Distributed Feature Composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.
35. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: a core language for abstract behavioral specification. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects: 9th Intl. Symp., FMCO, Graz, Austria. Revised Papers*, pages 142–164. Springer, 2012.
36. R. Küsters, T. Truderung, B. Beckert, D. Bruns, M. Kirsten, and M. Mohr. A hybrid approach for proving noninterference of Java programs. In C. Fournet, M. W. Hicks, and L. Viganò, editors, *IEEE 28th Computer Security Foundations Symp., CSF, Verona, Italy*, pages 305–319. IEEE Computer Society, 2015.
37. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
38. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 77–91, Berlin, Heidelberg, 2010. Springer-Verlag.
39. I. Schaefer and F. Damiani. Pure Delta-oriented Programming. In S. Apel, D. Batory, K. Czarnecki, F. Heidenreich, C. Kästner, and O. Nierstrasz, editors, *Proc. 2nd International Workshop on Feature-Oriented Software Development (FOSD'10) Eindhoven, The Netherlands*, pages 49–56. ACM Press, 2010.
40. I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012.
41. C. Scheben and S. Greiner. Information flow analysis. In Ahrendt et al. [1], chapter 13, pages 453–472.
42. T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.
43. T. Thüm, I. Schaefer, M. Hentschel, and S. Apel. Family-based deductive verification of software product lines. In K. Ostermann and W. Binder, editors, *Generative Programming and Component Engineering, GPCE'12, Dresden, Germany*, pages 11–20. ACM, 2012.
44. T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *Proc. Int'l Workshop Variability-intensive*

Systems Testing, Validation and Verification (VAST), pages 270–277. IEEE Computer Society, 2011.

45. A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, and S. Apel. Variability encoding: From compile-time to load-time variability. *J. Log. Algebr. Meth. Program.*, 85(1):125–145, 2016.
46. N. Wasser, R. Bubel, and R. Hähnle. Abstract interpretation. In Ahrendt et al. [1], chapter 6, pages 167–189.