with a ~~Phase 1~~ summary $\chi_f = (\mathcal{B}, AB)$ in a given program S, where $\mathcal{B} = \{B_1, B_2, \ldots, B_n\}$, the analysis considers *every pair* $(\mathtt{x}, \mathtt{y}) \in \Sigma \times \Sigma$ of functions that appear as a *substring* in some of the B_i sequences, i.e., $B_i = w \cdot \mathtt{x} \cdot \mathtt{y} \cdot w'$ for some sequences w, w'. Note that $\mathtt{x}$ could be ε (an empty sequence) if some B_i consists of a single function. All these "atomic pairs" are put into the set $\Omega \in 2^{\Sigma \times \Sigma}$. More formally,

$$\Omega = \{(\mathtt{x}, \mathtt{y}) \in \Sigma \times \Sigma \mid \exists (\mathcal{B}, AB) \in X_S : \exists B \in \mathcal{B} : [|B| = 1 \wedge (\mathtt{x}, \mathtt{y}) = (\varepsilon, B)] \\ \vee [|B| > 1 \wedge \exists w, w' \in \Sigma^* : B = w \cdot \mathtt{x} \cdot \mathtt{y} \cdot w' \wedge (\mathtt{x}, \mathtt{y}) \neq (\varepsilon, \varepsilon)]\}$$

where $X_S \in 2^{2^{\Sigma^*} \times \Sigma^*}$ is a set of all Phase 1 summaries of the program S.

Example 3.2.4. For instance, assume that in Phase 1, ~~there was~~ analysed a function `f`. It produced the summary $\chi_f = (\mathcal{B}, AB)$, where $\mathcal{B} = \{\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c},\ \mathtt{a} \cdot \mathtt{c} \cdot \mathtt{d}\}$, ~~i.e., a set~~ of sequences of functions that should be called atomically. The analysis will then look for the following pairs of functions that are not called atomically: $\Omega = \{\mathtt{a} \cdot \mathtt{b},\ \mathtt{b} \cdot \mathtt{c},\ \mathtt{a} \cdot \mathtt{c},\ \mathtt{c} \cdot \mathtt{d}\}$.

An element ~~of this~~ phase's abstract state is a triple $(\mathtt{x}, \mathtt{y}, \delta) \in \Sigma \times \Sigma \times 2^{\Sigma \times \Sigma \times \mathbb{N}}$ where $(\mathtt{x}, \mathtt{y})$ is a pair of the most recent calls of functions performed on the program path being explored, and δ is a set of so far detected *pairs that violate atomicity* on particular lines of code. Thus, the abstract states are elements of the set $2^{\Sigma \times \Sigma \times 2^{\Sigma \times \Sigma \times \mathbb{N}}}$. Whenever a function `f` is called on some path that led to an abstract state $(\mathtt{x}, \mathtt{y}, \delta)$, a new pair $(\mathtt{x}', \mathtt{y}')$ of the most recent function calls is created from the previous pair $(\mathtt{x}, \mathtt{y})$ such that $(\mathtt{x}', \mathtt{y}') = (\mathtt{y}, \mathtt{f})$. Further, when the current program state is not inside an atomic block, the analysis checks whether the new pair (or just the last call) violates atomicity (i.e., $(\mathtt{x}', \mathtt{y}') \in \Omega \vee (\varepsilon, \mathtt{y}') \in \Omega$). When it does, it is added to the set δ of pairs that violate atomicity.

Formally, the initial abstract state (in this phase) of a function is defined as $s_{init} = \{(\varepsilon, \varepsilon, \emptyset)\}$. To formalise the analysis of a function, let `f` be a called leaf function on a line c. Further, let $s_{\mathtt{g}}$ be the abstract state of a function `g` being analysed before the function `f` is called. After the call of `f`, the abstract state will be changed as follows:

$$s_{\mathtt{g}} = \{(\mathtt{x}', \mathtt{y}', \delta') \in \Sigma \times \Sigma \times 2^{\Sigma \times \Sigma \times \mathbb{N}} \mid \exists (\mathtt{x}, \mathtt{y}, \delta) \in s_{\mathtt{g}} : (\mathtt{x}', \mathtt{y}') = (\mathtt{y}, \mathtt{f}) \\ \wedge [(\neg lock \wedge \delta' = \{(\mathtt{x}'', \mathtt{y}'', c') \in \Sigma \times \Sigma \times \mathbb{N} \mid (\mathtt{x}'', \mathtt{y}'', c') \in \delta \\ \vee [((\mathtt{x}'', \mathtt{y}'') = (\mathtt{x}', \mathtt{y}') \vee (\mathtt{x}'', \mathtt{y}'') = (\varepsilon, \mathtt{y}')) \wedge (\mathtt{x}'', \mathtt{y}'') \in \Omega \wedge c' = c]\}) \vee (lock \wedge \delta' = \delta)]\}$$

A *summary* of the second phase of the analysis — $\chi_f \in 2^{\Sigma \times \Sigma \times \mathbb{N}}$ — of a function `f` is then $\chi_f = \Delta$, where Δ is a set of pairs that violate atomicity within the function `f`. Δ is constructed such that it contains a union of all the δ sets that appear on program paths through `f`. Formally, $\Delta = \bigcup_{\delta' \in \Delta'} \delta'$, where $\Delta' = \{\delta' \in 2^{\Sigma \times \Sigma \times \mathbb{N}} \mid \exists p \in s_f : \exists (\mathtt{x}, \mathtt{y}, \delta) \in p : \delta \neq \emptyset \wedge \delta' = \delta\}$ and s_f is the abstract state at the end of the abstract interpretation of `f`.

The analysis of functions with *nested function calls* and cases where *lock/unlock calls are not paired* within functions are handled analogically as in Phase 1. For a detailed explanation, see [15].

Example 3.2.5. To demonstrate the detection of an atomicity violation, assume the functions `f` and `g` from Listing 3.4. The set of atomic sequences of the function `f` with the first phase's summary $\chi_f = (\mathcal{B}, AB)$ is $\mathcal{B} = \{\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}\}$, thus, $\Omega = \{(\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{c})\}$. In the function `g`, an atomicity violation is detected because the pair of functions `b`, `c` is not called atomically on line 12, i.e., $(\mathtt{b}, \mathtt{c}) \in \Omega$. Consequently, the derived summary χ_g for the function `g` for the second phase of the analysis is $\chi_g = \{(\mathtt{b}, \mathtt{c}, 12)\}$.

```
1  void f()
2  {
3      x();
4      lock(&L); // a . b . c
5      a(); b(); c();
6      unlock(&L);
7      y();
8  }
9  void g()
10 {
11     x();
12     b(); c(); // ATOMICITY_VIOLATION: (b, c)
13     y();
14 }
```

Listing 3.4: An example of an *atomicity violation*

Summary of Phase 2 The sets of pairs that violate atomicity — the Δ sets — from the summaries of all analysed functions are finally reported to the user.

3.3 Atomer's Limitations

The basic version of Atomer has been proposed as it is detailed in Section 3.2. The first version of the analyser has also been implemented in [15], and it works as expected. Moreover, it can be used in practice to analyse various kinds of programs, and it may find *real-world atomicity related bugs*. Nevertheless, there are still several *limitations* and cases where the original version of Atomer would not work correctly, i.e., cases not addressed during the original proposal. Some of these cases are briefly discussed already in [15] and further described in [16].

So far, Atomer does not distinguish *different lock instances* used simultaneously in a program. Only calls of locks/unlocks are identified, and the parameters of these calls — *lock objects* — are not considered at all. Therefore, if there are several lock objects used, the analysis does not work correctly. Although this may happen in *real-life programs*, insomuch as one could have, e.g., another (smaller) atomic section inside a current atomic section (this does not have to be evident at first because the *inner atomic section* could be, e.g., inside a nested function). For example,

```
... lock(L1); ... lock(L2); ... unlock(L2); ... unlock(L1); ...
```

Another possibility is an *alternating sequence of locks*, e.g., two locks are locked at first, and then, they are unlocked in the same order, i.e.,

```
... lock(L1); ... lock(L2); ... unlock(L1); ... unlock(L2); ...
```

Another limitation of Atomer's basic version is that it supports only the analysis of programs written in the *C language* that use *PThread* locks to *synchronise concurrent threads*. Of course, in practice, many other *types of locks* for synchronisation of concurrent threads or even synchronisation of *concurrent processes* are used. Although the first version of

X.2

① – Ty prvky by měly být zase rozšířeny o zásobník. Tedy:

$$(x, y, \ell, V) \in \Sigma \times (\Sigma \cup \{\varepsilon\}) \times \{0,1\} \times 2^{\Sigma \times (\Sigma \cup \{\varepsilon\}) \times \mathbb{N}}$$

(jednoprvkové sekvence; ℓ = level)

– Doména by pak byla

$$2^{\Sigma \times (\Sigma \cup \{\varepsilon\}) \times \{0,1\} \times 2^{\Sigma \cup (\Sigma \cup \{\varepsilon\}) \times \mathbb{N}}}$$

nebo je to na první Σ?

② Namísto (x, y, σ) by mělo pak být (x, y, ℓ, V), z čehož vznikne (y, f, ℓ, V'), kde

– $\ell = 0 \wedge ((y, f) \in \Omega \vee (\varepsilon, f) \in \Omega) \Rightarrow$

$$V' = V \cup \{(y, f) \mid (y, f) \in \Omega\} \cup \{(\varepsilon, f) \mid (\varepsilon, f) \in \Omega\}$$

– $\ell = 1$: $V' = V$

Jednodušší je:

$$V' = V \cup \{(y, f) \mid \ell = 0 \wedge (y, f) \in \Omega\} \cup \{(\varepsilon, f) \mid \ell = 0 \wedge (\varepsilon, f) \in \Omega\}.$$

Atomer can analyse C programs with other types of locks, these locks are not recognised as locks. Thus, the analysis would not work as expected. It would definitely be helpful also to analyse other languages than just C. As described in Section 2.3, Facebook Infer is capable of analysing programs written in C, C++, Objective-C, Java, and C#. The analysis algorithm could then be the same for all these languages because Facebook Infer's *intermediate language* is analysed instead of directly analysing the input languages. Again, the first version of Atomer should be able to analyse the above languages, but it has not been tested within [15]. However, most importantly, other languages may use *very different lock types*, which would not be recognised. Examples of some advanced locking mechanisms (not supported by the basic version of Atomer) are *lock guards*, *re-entrant locks*, or *try-locks*.

Regarding *scalability*, the basic version of Atomer can have problems with more *extensive* and *complex* programs, as mentioned in [15] (problems with *memory* as well as problems with the *analysis time*). The problem is working with the sets of (A, B) pairs of *sequences* in the abstract states of Phase 1 and working with *sequences* of calls in the summaries of this phase. It may be necessary to store many of these sequences, and they can be very long (due to all different paths through the CFG of an analysed program). It may lead to the *state space explosion problem* [35].

One of the main reasons that Atomer's first version reports *false alarms* is that in *critical sections*, in real-world programs, there are sometimes called *generic functions* that do not influence atomicity violations (such as functions for printing to the standard output, functions for recasting variables to different types, functions related to iterators, and whatever other "safe" functions for particular program types). Often, to find some atomicity violations, it is sufficient to focus only on certain "critical" functions. In practice, another issue is that in an analysed program, there can be "large" critical sections or critical sections in which appear function calls with a *deep hierarchy of nested function calls*. All the above cases may cause massive and "imprecise" atomic sequences that are the source of false alarms. However, regardless of the above issues, Atomer can still report quite some false alarms. It is due to the assumption that *sequences called atomically once* should *always be called atomically*, but this does not always have to hold. None of the above reasons that can generate false alarms is resolved in the first version of Atomer.

The next source of false alarms is something that the author of this work calls *local atomicity violations*. Imagine a function `f` that contains non-atomic calls of functions `a`, `b`, and these functions should always be invoked atomically. Obviously, this is an atomicity violation. However, suppose that `f` is called exclusively from atomic sections of other functions higher in the call hierarchy. In this case, in effect, that is not a real atomicity violation (it can be considered as a local atomicity violation within a single function, but globally, it is not). As a consequence, a false alarm would be reported by the basic version of Atomer. In real-life programs, this situation may be fairly common due to *complicated call graphs*.

Atomer considers only the *basic contracts for concurrency*, defined in Section 2.4.1. It is pretty limited in some circumstances, and therefore, Atomer can report *false alarms*. The basic contracts do not take into consideration the *data flow* within function calls. However, a better idea is to work with the assumption that a sequence of function calls must be atomic only if it *handles the same data*. Assume that functions `f`, `g` are manipulating with the same container `C` as follows: `f(C); g(C);`. These are called atomically. Somewhere else — where `f`, `g` are not called atomically — it does not necessarily cause an atomicity violation because they can be invoked with different arguments, which could be valid.

To deal with such cases one would need the

~~This behaviour corresponds to the~~ *extended contracts with parameters* (see Section 2.4.2). Another (more complex) limitation is that basic contracts do not consider any *contextual information*. It would be more precise to consider as atomicity violations such sequences that could be violated only by particular ("dangerous") function calls, not by any calls. For example, suppose that there is the following sequence of functions called atomically: `f(); g();`. While somewhere else, these functions are not called atomically, it does not necessarily cause that it is an atomicity violation because, in this particular context, none of the "dangerous" functions can be executed by any concurrent thread. The *extended contracts with spoilers* formally describe these scenarios in Section 2.4.3.

A remarkable problem (though it is not directly a problem of Atomer) is identifying whether a reported atomicity violation is a *real bug* or whether it is just a false alarm. It could be really challenging, especially in *extensive real-life* programs.

Solutions for some of the above issues and limitations are proposed in Chapter 4 and further implemented in a new version of Atomer, detailed in Chapter 5.

Chapter 4

Proposal of Enhancements for Atomer

In this chapter, the author proposes solutions for some of the Atomer's limitations stated in Section 3.3. The solutions enhance the analysis's *precision* and *scalability*. In order to formally define these enhancements, the notions and symbols introduced in Section 3.2 are used. Some of the enhancements were briefly discussed already in [16].

Section 4.1 proposes an optimisation of Atomer's scalability. The following Sections 4.2, 4.3, 4.4 cover precision improvements, i.e., extensions of Atomer with additional features that improve its ability to cope with cases that were not supported in the first version of Atomer, and that can be seen in *real-life code*. Furthermore, Chapter 5 provides an overview of the implementation of all the below improvements in a new version of Atomer.

In the following sections, to give an intuition, there are used listings with pieces of C programs (assuming `lock/unlock` functions for *mutual exclusion* to *critical sections*). In addition, there are used C++ and Java programs to illustrate the *locking mechanisms* in these languages.

4.1 Approximation of Sequences by Sets

Regarding *scalability*, the basic version of Atomer can have problems with more *extensive* and *complex* programs, which can manifest both in its *time* and *memory* consumption. The problems arise primarily due to working with the sets of (A, B) pairs of **sequences** of function calls in *abstract states* (during Phase 1). It may be necessary to store many of these sequences, and they could be very long (due to all different paths through the CFG of an analysed program). The author's idea is to *approximate* these sets by working with sets of (A, B) pairs of **sets** of function calls. Apart from representing the abstract states of the first phase of the analysis, elements of these pairs do also appear in the first phase's *summaries*, and they are then used during Phase 2 as well. Thus, it is needed to make a certain approximation in the summaries and their subsequent usage too. The approximated phases of the analysis and their collaboration are illustrated in Figure 4.1 (one can compare that with the illustration of the first version of Atomer in Figure 3.1).

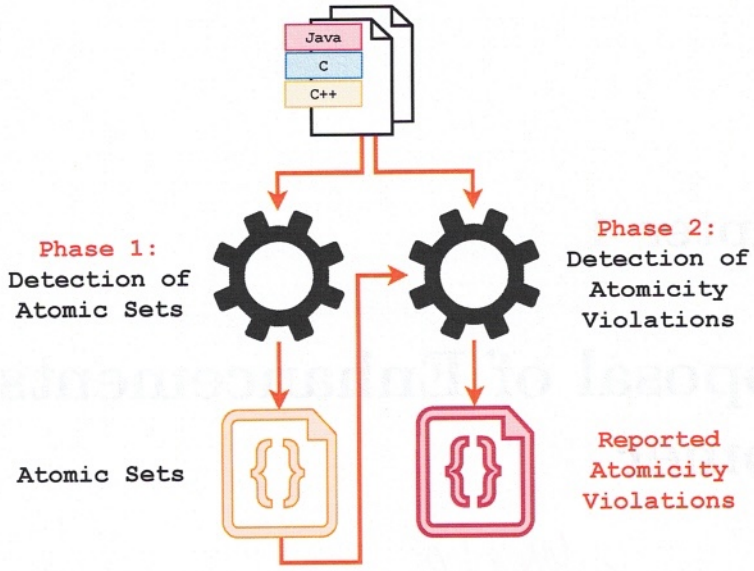

Figure 4.1: An illustration of the *phases* of the Atomer's analysis and the *high-level analysis process* with an *approximation* of working with *sequences* by working with *sets* (moreover, note that a new version of Atomer accepts programs also written in C++ and Java languages, which is described in Section 5.3)

In particular, the proposed solution is more scalable because the ordering of function calls that appear in the pairs is not relevant anymore. Therefore, less memory is required because different sequences of function calls can map the same set. The analysis is also faster since there are stored fewer sets of function calls to work with. On the other hand, the analysis is less accurate because the new approach causes some loss of information. In practice, this loss of information could eventually lead to *false alarms*. However, the number of such false alarms is typically not that high as this thesis's experimental evidence shows. Moreover, later, there are discussed some techniques that may rid of these false alarms.

4.1.1 Approximation with Sets in Phase 1

The *detection of sequences of calls to be executed atomically* now generates all (A, B) pairs of **sets** of function calls for each path instead of pairs of **sequences**, i.e., $(A, B) \in 2^{\Sigma} \times 2^{\Sigma}$. Here, A, B are not *reduced sequences* (the notion of a reduced sequence is not needed anymore) but sets. The purpose of the pairs is preserved. Hence, the abstract states are elements of the set $2^{2^{\Sigma} \times 2^{\Sigma}}$. In all the implemented algorithms and definitions, it is sufficient to work with:

- *sets* 2^{Σ} of functions, instead of *sequences* Σ^* of functions;
- the *empty sets* $\emptyset$, instead of the *empty sequences* ε; and
- *unions* $\cup$ of sets, instead of the *concatenation* $\cdot$ of sequences.

The above implies that the *initial abstract state* of a function is changed to $s_{init} = \{\{(\emptyset, \emptyset)\}\}$. During the analysis of a function `g` with an abstract state $s_{\mathtt{g}}$, when a leaf function `f` is called,

the abstract state's transformation is changed as follows:

$$s_{\mathtt{g}} = \{p' \in 2^{2^{\Sigma} \times 2^{\Sigma}} \mid \exists p \in s_{\mathtt{g}} : p' = \{(A', B') \in 2^{\Sigma} \times 2^{\Sigma} \mid \exists (A, B) \in p : \\ [\neg actual(p, (A, B)) \wedge (A', B') = (A, B)] \vee [actual(p, (A, B)) \\ \wedge [(lock \wedge (A', B') = (A, B \cup \{\mathtt{f}\})) \vee (\neg lock \wedge (A', B') = (A \cup \{\mathtt{f}\}, B))]]\}\}$$

Further, when an unlock is called, a new (A, B) pair is created as follows:

$$s'_{\mathtt{g}} = \{p' \in 2^{2^{\Sigma} \times 2^{\Sigma}} \mid \exists p \in s_{\mathtt{g}} : p' = \{(A, B) \in 2^{\Sigma} \times 2^{\Sigma} \mid \\ [(A, B) = (\emptyset, \emptyset) \wedge setActual(p, (A, B))] \vee (A, B) \in p\}\}$$

Other algorithms (e.g., calling an already analysed *nested* function) are modified analogically.

Another approximation was made in the summaries. The first component of the summary has to be changed to a set of sets of function calls because it is constructed from the B items from the abstract states, which are now sets. The second component of the summary can be changed to a set of function calls because even before, it was a reduced sequence of all the (A, B) pairs. Therefore, the ordering of function calls was significantly approximated even so. Moreover, it is used to analyse functions higher in the *call hierarchy* where it is appended to A or B, which are now sets. Thus, it would make no sense to store it in summaries as a sequence. Formally, the form of summaries χ changes from $2^{\Sigma^*} \times \Sigma^*$ to $2^{2^{\Sigma}} \times 2^{\Sigma}$. In particular, a summary $\chi_{\mathtt{f}} \in 2^{2^{\Sigma}} \times 2^{\Sigma}$ of a function $\mathtt{f}$ is redefined as $\chi_{\mathtt{f}} = (\mathcal{B}, AB)$, where:

- $\mathcal{B} = \{B' \in 2^{\Sigma} \mid \exists p \in s_{\mathtt{f}} : \exists (A, B) \in p : B \neq \emptyset \wedge B' = B\}$, where $s_{\mathtt{f}}$ is the abstract state at the end of the abstract interpretation of $\mathtt{f}$.
- $AB = \bigcup_{ab \in AB'} ab$, where $AB' = \{ab \in 2^{\Sigma} \mid \exists p \in s_{\mathtt{f}} : \exists (A, B) \in p : ab = A \cup B\}$.

Example 4.1.1. For demonstrating the approximation of the analysis to sets, assume functions `f` and `g` from Listing 4.1. Further, assume that `a`, `b`, `x`, `y` are leaf nodes of the call graph. Before the approximation, when the analysis was working with sequences of function calls, Phase 1 of the analysis produced the following abstract states and summaries while analysing the functions:

- $s_{\mathtt{f}} = \{\{(\mathtt{x} \cdot \mathtt{y},\ \mathtt{a} \cdot \mathtt{b}), (\mathtt{y} \cdot \mathtt{x},\ \mathtt{b} \cdot \mathtt{a}), (\varepsilon,\ \varepsilon)\}\}$, $\chi_{\mathtt{f}} = (\{\mathtt{a} \cdot \mathtt{b},\ \mathtt{b} \cdot \mathtt{a}\},\ \mathtt{x} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b})$;
- $s_{\mathtt{g}} = \{\{(\mathtt{y} \cdot \mathtt{x},\ \mathtt{b} \cdot \mathtt{a}), (\varepsilon,\ \varepsilon)\}\}$, $\chi_{\mathtt{g}} = (\{\mathtt{b} \cdot \mathtt{a}\},\ \mathtt{y} \cdot \mathtt{x} \cdot \mathtt{b} \cdot \mathtt{a})$.

Whereas, after the approximation, the produced abstract states and summaries are as follows: $s_{\mathtt{f}} = s_{\mathtt{g}} = \{\{(\{\mathtt{x}, \mathtt{y}\}, \{\mathtt{a}, \mathtt{b}\}), (\emptyset, \emptyset)\}\}$, $\chi_{\mathtt{f}} = \chi_{\mathtt{g}} = (\{\{\mathtt{a}, \mathtt{b}\}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{x}, \mathtt{y}\})$. They are the same for both functions because there are the same locked/unlocked function calls; only the order of calls differs.

4.1.2 Approximation with Sets in Phase 2

The *detection of atomicity violations* in Phase 2 then works almost the same way as before the approximation. However, there is one difference. Before, the analysis implemented in

```
1  void f()
2  {
3      x(); y();
4      lock(&L); // a . b -> {a, b}
5      a(); b();
6      unlock(&L);
7      y(); x();
8      lock(&L); // b . a -> {a, b}
9      b(); a();
10     unlock(&L);
11 }
12 void g()
13 {
14     y(); x();
15     lock(&L); // b . a -> {a, b}
16     b(); a();
17     unlock(&L);
18 }
```

Listing 4.1: A code snippet used to illustrate the proposed *approximation* of the first phase of the analysis in a new version of Atomer using *sets of function calls*

the second phase looked for violations of atomic sequences obtained from Phase 1. Now, **atomic sets** are obtained from Phase 1; hence, the detection of atomicity violations needs to work with sets too. Again, the analysis looks for *pairs of functions that should be called atomically*, while this is not the case on some path through the CFG. This algorithm is identical to the algorithm before the approximation.

Nevertheless, it is needed to propose a new algorithm that derives the pairs of function calls (from the atomic sets) to be checked for atomicity (i.e., the set $\Omega \in 2^{\Sigma \times \Sigma}$). Intuitively, the second phase of the analysis now looks for non-atomic execution of any pair of functions `f`, `g` such that $\{\mathtt{f}, \mathtt{g}\}$ is a *subset* of some set of functions that were found to be executed atomically. In order to obtain the pairs, all possible pairs of functions are taken from atomic sets from Phase 1, i.e., all *2-element variations*. Formally, let S be an analysed program, and let $X_S \in 2^{2^{2^\Sigma} \times 2^\Sigma}$ be a set of all summaries of the program S. Then, all the atomic pairs (the first item of a pair may be empty if an atomic set consists of a single function) are obtained as follows:

$$\Omega = \{(\mathtt{x}, \mathtt{y}) \in \Sigma \times \Sigma \mid \exists (\mathcal{B}, AB) \in X_S : \exists B \in \mathcal{B} : [|B| = 1 \wedge (\mathtt{x}, \mathtt{y}) \in \{\varepsilon\} \times B] \\ \vee [|B| > 1 \wedge (\mathtt{x}, \mathtt{y}) \in B \times B \wedge \mathtt{x} \neq \mathtt{y}]\}$$

Example 4.1.2. For example, assume that Phase 1 analysed a function `f`, which produced the summary $\chi_{\mathtt{f}} = (\mathcal{B}, AB)$. Assume that before the approximation, a set of sequences of functions that should be called atomically was as follows: $\mathcal{B} = \{\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}\}$. Then, the analysis looked for the following pairs of functions that are not called atomically: $\Omega = \{(\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{c})\}$. Since the result of the first component of the summary was changed to the set $\mathcal{B}$ of sets of functions that should be called atomically ~~as follows~~: $\mathcal{B} = \{\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}\}$, the analysis now looks for the following pairs of functions that are not called atomically (all 2-element variations): $\Omega = \{(\mathtt{a}, \mathtt{b}), (\mathtt{a}, \mathtt{c}), (\mathtt{b}, \mathtt{a}), (\mathtt{b}, \mathtt{c}), (\mathtt{c}, \mathtt{a}), (\mathtt{c}, \mathtt{b})\}$.

X.3.

- Doména by asi měla být

$$2^{2^{\Sigma} \times 2^{\Sigma} \times \{0,1\} \times \{0,1\}}$$

(↑ active, ↑ locked)

- $Sinit = \{\{(\emptyset, \emptyset, 0, 0)\}\}$

↑ vždy se očekává unlocked?

Co když je lock výše? Rozbrání se později?

- Volání leaf f-ce f:

$$S'_g = \bigcup_{p \in S_g} \{ (p \cap 2^{\Sigma} \times 2^{\Sigma} \times \{0\} \times \{0,1\}) \cup \{ (A, B \cup \{f\}, 1, 1) \mid (A, B, 1, 1) \in p \} \cup \{ (A \cup \{f\}, B, 1, 0) \mid (A, B, 1, 0) \in p \} \}$$

- Pro unlock:

$$S'_g = \bigcup_{p \in S_g} \{ \{ (A, B, 0, \ell) \mid \exists c \in \{0,1\} . (A, B, c, \ell) \in p \} \cup (\emptyset, \emptyset, 1, 0) \}$$

- Implikaci $\{B\}$ zjednodušit a upravit na:

$$\mathcal{B} = \{ B \in 2^{\Sigma} \setminus \emptyset \mid \exists p \in S_f \; \exists A, \exists c, \exists \ell : p = \{(A, B, c, \ell)\} \}$$

- C by mělo jít zjednodušit až na

$$C = \bigcup_{(A,B,c,\ell) \in S_f^{A \cup B}} \quad \text{(tedy 'AB')}$$