Combining Analyses for C Program Verification

Loïc Correnson and Julien Signoles*

CEA LIST, Software Safety Lab,
PC 174, 91191 Gif-sur-Yvette, France
firstname.lastname@cea.fr

Abstract. Static analyzers usually return partial results. They can assert that some properties are valid during all possible executions of a program, but generally leave some other properties to be verified by other means. In practice, it is common to combine results from several methods manually to achieve the full verification of a program. In this context, Frama-C is a platform for analyzing C source programs with multiple analyzers. Hence, one analyzer might conclude about properties assumed by another one, in the same environment. We present here the semantical foundations of validity of program properties in such a context. We propose a correct and complete algorithm for combining several partial results into a fully consolidated validity status for each program property. We illustrate how such a framework provides meaningful feedback on partial results.

1 Introduction

Validating a program consists in exhibiting evidence that it will not fail during any of its possible executions. From an engineering point of view, this activity generally consists in manual reviews, testing and formal verifications. Static analyzers can be used to *prove* properties about programs. More precisely, given the source code of a program, an analyzer states a property of *all* of its possible executions. However, analyzers are generally *partial*: they assert some program properties, but leave other ones unverified. Let us illustrate this point of view with some examples of verification techniques.

Abstract Interpretation [1]. This technique computes over-approximations of possible values of each memory location during program execution. When all values in the over-approximation of the memory entail a property, then the property holds during any concrete execution of the program. Otherwise, nothing can be claimed about the property. When such a property is required to hold for the analysis to proceed, the analyzer generally assumes its validity. Hence, the analyzer makes an *assumption* to be verified by other means.

Deductive Verification [2]. This modular technique explicitly proves that a property holds after the execution of a small piece of code, whenever some other property holds before it. We generally say that the *pre*-condition of the verified code entails its *post*-condition. These small theorems can then be chained with each others in order to prove that, whenever some initial pre-condition holds on initial states, the

* This work has been supported by the French ANR project U3CAT and the French FUI9 project Hi-Lite.

desired properties hold on all concrete executions of the program. Generally, not all these elementary steps can be proved, and there remain some properties to be asserted by other means.

Testing. In some sense, testing still falls into the depicted category of analyzers. Executing a test actually asserts that, for all possible executions, if the considered execution corresponds to the test case, then the property defined by the oracle of the test holds. This point of view is of particular interest when we aggregate a collection of tests that covers some criteria. Then, one might claim that the verified properties might only be invalid outside of the covered criteria. Last but not least, testing is also used to exhibit properties that *do not* hold, an activity of major interest during the verification engineering process.

A general practical approach is then to combine several analyzers to increase the coverage of verified properties. Thus there is a need for ensuring the consistency of several partial results. The purpose of this article is to give a semantical foundation to this problem and to provide an algorithm to combine several partial results from different analyzers. A salient feature of our approach is the use of a *blocking semantics*, which is pivotal in ensuring the correctness of the aforementioned algorithm. It allows the correctness to be independent from the hypotheses that the analyzers use to establish their results. These claims remain nevertheless essential for the completeness of the algorithm. The proposed framework is language independent, although it is instantiated in Frama-C [3], a platform dedicated to the verification of critical embedded software written in C, typically in the domain of avionics and energy industries.

Related Work. Combining analysis techniques (in particular static and dynamic ones) is a quite recent but not new idea [4]. However only very few of these works tackle the goal of formally verifying a program by combining these techniques in a consistent way. Heintze *at al.* [5] proposes a framework by equational reasoning to combine an abstract interpreter with a deductive verification tool to enhance verification of user assertions. As in our work, it does not depend on specific analyzers and is correct modulo analyzer's correctness. However, instead of focusing on merging analyzer's results, it implements a new analyzer which operates on the results of the analyzers which it is based on. This analyzer is incomplete in the sense that it not does always provide the more precise result. More recently, the Eve verification environment for Eiffel programs combines a deductive verification tool and a testing tool in order to make software verification practical and usable [6]. Eve reports the separated results obtained from this tool. Since tools which Eve is based upon are not supposed to be correct, Eve computes a so-called *correctness score* for each property. This score is a metrics indicating a level of confidence in its correctness. That is quite different from our approach where we suppose that analyzers are correct but can use other properties as hypotheses to build a proof. Comar *et al.* [7] also aim to integrate a deductive verification tool with testing to verify Spark[1] programs. As in our work, proofs are based on a notion of hypotheses, called *assumptions* in their context. However, to avoid consistency issues, they split the program in several chunks: in each chunk, the same analysis techniques must be applied. In our approach, we allow the user to verify each property by different means.

[1] Spark is a subset of Ada dedicated to the development of critical software.

Outline. The article is structured as follows. First Section 2 presents the problem and the key concepts thanks to a simple C program and a set of properties to verify on it. Section 3 introduces the semantic framework where key concepts are precisely defined. Section 4 presents the algorithm to compute consolidation statuses of properties in a logical way. Section 5 finally focuses on practical usages of the proposed framework: we explain the large variety of user feedbacks that can be obtained after consolidation.

2 Key Concepts

This section introduces all the concepts presented in this article through a running example. It consists of a short program written in C depicted in Figure 1. The program initializes an array with values returned by some external function `f` for which the code is not provided, but only some property `P` on its result is known. We are interested in proving different categories of properties on this short program:

- the program should never produce *runtime errors*, which are situations where the program's behavior is explicitly undefined by the ISO Specification of the C programming language, such as divisions by zero or accesses to uninitialized variables and invalid memory cells;
- once initialized, the values of the array satisfy the property `P` as expected.

Properties in the first category implicitly follow from the language semantics. The second category needs to be expressed explicitly by the developer in order to be verified. The Frama-C platform supports the ACSL language [8] for this purpose. We do not get into details of ACSL here: it is a first-order logical language designed to expressing properties of a C program during its execution.

```
1  /*@ axiomatic A { predicate P(int x); } */
2
3  /*@ ensures P(\result);
4    @ assigns \nothing; */
5  int f(int);
6
7  void main(void) {
8    int i, n = 0, a[100];
9    for(i = 1; i <= 10; i++) n += i;
10   // Have n = Sum {1..10}
11   for(i = 0; i < n; i++) a[i] = f(i);
12   //@ assert \forall integer k; 0 <= k < n ==> P(a[k]);
13 }
```

Fig. 1. Annotated Code Example

We now comment the source code of Figure 1 in more details. ACSL constructs are inserted into `@`-comments. The predicate `P` is abstractly defined in the pure logic world (axiomatic clause). The external function `f` is declared to have no visible side-effect (assigns clause), and to have its results satisfying `P` (ensures clause). The code to be verified lies in function `main`. It consists in two loops: the first one computes the sum of integers from 1 to 10 and stores the result in local variable `n`; the second loop initializes

the n-th first indices of array a with function f. Finally, the ACSL clause assert states the additional property we want to verify for indices less than n. The set of properties to be verified for this simple program is then:

Overflows and Runtime Errors: three potential arithmetic overflows and one potential invalid memory access.
User Property: one user assertion to prove.
External Properties: the specification of function f.

Two static analyzers distributed with Frama-C address those properties:

Value [9] uses a context sensitive forward abstract interpretation to compute an over-approximation of possible values of variables at each program point. This analysis verifies the absence of *any* runtime error and can also handle simple ACSL assertions, like quantifier-free assertions.

Wp [10] implements deductive verification. This modular analysis is able to verify complex logical annotations using external automated or interactive provers, but requires extra code annotations to carry function contracts and loop invariants.

We intend here to use Value for proving the absence of runtime errors, and Wp to prove the assertion, which is not in the scope of Value. The external specification of f will be trusted here. In the rest of this section, we first report on an incremental study for verifying this program. Then, we introduce our key concepts of local and consolidated statuses for properties managed by Frama-C.

It would also be possible to use Wp or other analyzers to prove the absence of runtime errors thanks to the RTE Frama-C's plug-in, which generates standard ACSL assertions for any potential runtime error in a source code. More generally, using RTE promotes runtime errors to standard properties that smoothly integrate with our framework. However, even small C programs reveal many potential runtime errors, and generating all assertions produces a lot of noise compared to user-defined assertions. When Value can be used, it is then much more preferable to rely on it for runtime errors.

2.1 Verifying Properties in Practice

Running Value alone with its default configuration on this program gives poor results: variable i is not tied enough and the over-approximation of n contains overflowing values that become negative. Hence the memory access to a[i] in the second loop may be invalid and an alarm is generated. Running Value a second time with option -slevel 100 makes the analyzer more precise during the first loop[2]. This time, all potential errors are discarded, and we get the following interesting properties on the final memory state: n is equal to 55, a[0..54] takes any int value, and a[55..99] remains uninitialized.

Running Wp to prove the quantified assertion requires additional annotations from the developer, especially on loops. As illustrated in the Wp tutorial [11], a canonical way of proving such a property is to insert the loop invariants of Figure 2. The re-

[2] The option -slevel N of Value makes the analyzer works over N different over-approximations in parallel. On our running example, the maximum of precision is obtained for $N \geq 55$. $N \geq 10$ is sufficient to prove the intended properties.

```
1  /*@ axiomatic A { predicate P(int x); } */
2
3  /*@ ensures P(\result);
4    @ assigns \nothing; */
5  int f(int);
6
7  void main(void) {
8    int i, n = 0, a[100];
9    for(i = 1; i <= 10; i++) n += i;
10   /*@ loop invariant 0 <= i <= n ;
11     @ loop invariant \forall integer k; 0 <= k < i ==> P(a[k]);
12     @ loop assigns i,a[0..n-1]; */
13   for(i = 0; i < n; i++) a[i] = f(i);
14   //@ assert \forall integer k; 0 <= k < n ==> P(a[k]);
15 }
```

Fig. 2. Annotated Code Example for `Wp`

sults of running `Wp` alone are quite encouraging: all annotations are discharged by the Alt-Ergo theorem prover [12], except the first loop invariant `0<=i<=n`. `Wp` proves the preservation of this invariant over loop iterations, but fails to establish it at the very beginning of the loop, because there is no invariant on the first loop establishing that `0<=n`. Of course, it is possible to complete the verification with `Wp` on the first loop, but these range properties over `n` are simple enough to be verified by `Value`. Running both `Value` with option `-slevel 100` and `Wp` on the completely annotated code of Figure 2, we obtain the following results:

Runtime Errors: *all* potential runtime errors are discharged by `Value`.
Loop Annotations: `Wp` proves two of the three, but leaves the first invariant unverified. `Value` proves only this range invariant.
User Property: `Wp` proves it, but under the hypothesis of the range invariant.
External Properties: they are assumed here, but should be verified later against both the definition of `P` and the actual code of `f`.

Intuitively, the verification task is now complete: everything has been discharged by at least one analyzer. But formal practitioners would notice that it is not clear whether such a verification is conclusive. Indeed, complex dependencies between properties might interfere with each others.

2.2 Soundly Merging Results

A presentation of the results obtained during our verification process can be represented by a graph. With a node for each property, we can represent assumptions by edges from the proved property towards its hypotheses. We also represent analyzers as nodes, with edges towards the properties they established. To increase readability, it is convenient to merge isomorphic nodes into a single one. On our running example, the associated *final* graph is represented in Figure 3.

Such a report is actually accessible through the report plug-in and from the graphical user interface of Frama-C. Let us now present how Frama-C is able to perform this consolidation and report about this verification process. In the example presented above, we have collected different results at different times by using two analyzers with various parameters. Hence, it is not possible to build efficiently and incrementally the desired

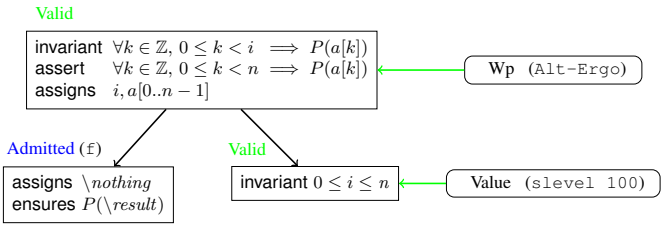

Fig. 3. Consolidated Graph of Properties Validity

graph of Figure 3. Instead, it is easy to register each verification experiment with their parameters in a database and to build the consolidation on demand.

This is the key idea behind *local* versus *consolidated* statuses of properties. We introduce the concept of *emitter* to identify an *analyzer* with all of its *parameters*. The partial results provided by an analyzer are registered in a Frama-C database. Each entry of the database precisely consists of:

- an emitter made of an *analyzer* with concrete *parameters' values*;
- a target *property*;
- a *local status*, ranging over *True*, *False* or *Dont_know*;
- a list of *properties* notably used by the analyzer to claim this local status.

The entries obtained after many verification rounds can be very complex to represent. The graph in Figure 4 shows an extract from the full data collected during the verification of Example 2. Two kinds of nodes distinctly represent properties and emitters. Edges are added when analyzers emit local statuses. For instance, three edges are added when `Wp` (using Alt-Ergo as prover) emits *True* for the user assertion A: one from `Wp` to A labeled by the status, and two from A to the loop invariants representing the hypotheses under which this status holds.

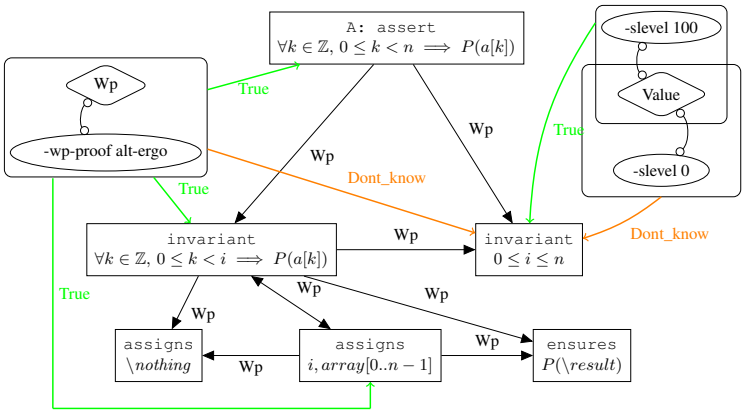

Fig. 4. Graph of Local Status (extract)

Let us illustrate how we obtain the *consolidated* status for this user assertion. Consider all the dependencies that were emitted in conjunction with a local status *True*.

All these paths either end at the range invariant, which is locally *True* with no more assumptions, or at the function contract of `f`. The internal cycle between loop invariants represents `Wp`'s internal inductive scheme. Hence the consolidation algorithm of Frama-C concludes that everything is proven, except the admitted properties of `f`. As we will see, consolidation can be quite challenging on more involved programs. For instance, a property may be false but only over unreachable traces. The general algorithm is complex enough for a semantical approach to be necessary.

3 Semantics

This section formalizes the semantics of annotated programs and property statuses. Our formalisation is independent from both the programming language and the formal specification language: we only suppose that the programming language is imperative, based on a set of *instructions*, and admits a specification language based on a set of *predicates*.

Property. A *property* $\pi = \phi \blacktriangleright \iota$ is a predicate ϕ attached to the program point just before the instruction ι. A predicate which does not depend on a program point (*e.g.* a mathematical lemma required to prove the program) is supposed to be attached to an arbitrary instruction ι_0 without any effect and put just before the first instruction of the program. We note Φ_P the finite set of properties of a program P.

Evaluation. The programming language being imperative, we suppose that there is a notion of *state* in which instructions are evaluated consistently with the operational semantics of the programming language. This notion of evaluation can be extended to predicates, as presented for instance in Herms' works [13]: a state ς validates ϕ, denoted by $\varsigma \models \phi$, if and only if the predicate ϕ is valid in the state ς.

Trace. We now consider that the underlying programming language comes with a trace semantics [14, 15] keeping all intermediate instructions and states during execution. Thus a trace $\sigma = (\varsigma_i \rhd \iota_i)_i$ is a (potentially infinite) sequence of instructions, each of them coming with the state in which it is evaluated. Traces begin at the early program entry point $\varsigma_0 \rhd \iota_0$ and are consistent with the small step operational semantics of the program: at each step k, the transition $\varsigma_k \xrightarrow{\iota_k} \varsigma_{k+1}$ holds in the operational semantics of the program. A finite trace σ does not contain the final state ς of a finite execution. But it is still possible to extend it with $(\varsigma \rhd \texttt{skip})$ where `skip` is the identity instruction which does not modify ς. We note $\sigma_1 \prec \sigma_2$ if and only if σ_1 is a strict trace prefix of σ_2. Also, we say that the trace σ ends at instruction ι in state ς, and we note $\sigma \hookrightarrow \varsigma \rhd \iota$, if and only if σ is a finite trace of length n such that $\varsigma_n = \varsigma$ and $\iota_n = \iota$. By extension, for a property π attached at instruction ι, we note $\sigma \hookrightarrow \pi$ if and only if $\sigma \hookrightarrow \varsigma \rhd \iota$ for some state ς.

Trace Validity. We also extend the notation $\models$ for predicates to traces and properties. With $\pi = \phi \blacktriangleright \iota$, we say that σ validates π, and we note $\sigma \models \pi$, the fact: if σ ends at $\varsigma \rhd \iota$ then $\varsigma \models \phi$.

Trace Invalidity. The converse notation, $\sigma \not\models \pi$, is used for the logical negation of $\sigma \models \pi$. Remark it is *not* equivalent to $\sigma \models \neg\pi$, however, we still have $\sigma \not\models \pi \Rightarrow \sigma \models \neg\pi$.

Blocking Semantics. The correctness of our algorithm (see Theorem 1) requires a *blocking semantics* which is usual in semantics of annotated programs (see for instance [13, 16]). In our theoretical framework, it can be expressed as follows.

Assumption 1 (Blocking Semantics) *If a trace leads to an invalid property, then the program stops and does not evaluate the following properties in the execution flow. More formally:*

$$\forall \textit{ traces } \sigma \textit{ and } \sigma', \forall \textit{ properties } \pi \textit{ and } \pi',\ \textit{if } \sigma' \prec \sigma \textit{ and } \sigma' \not\models \pi',\ \textit{then } \sigma \not\hookrightarrow \pi.$$

If all properties are valid, the blocking semantics coincides with the non-blocking one.

Reachability. An associated concept is the reachability of some instruction of the program. More precisely, we are interested in the reachability of instructions to which given properties are attached. In our framework, this concept is represented by global predefined (meta) properties of program properties, attached to the initial state ι_0 of the program:

$$\mathrm{reach}(\pi) \triangleq (\exists \sigma, \sigma \hookrightarrow \pi) \blacktriangleright \iota_0.$$

Local Validity. We say that a property π is *locally valid* under a finite set of hypotheses ξ, and we note $\xi \models \pi$, if and only if:

$$\forall \text{ trace } \sigma,\ \text{if } (\forall \pi_i \in \xi, \forall \text{ trace } \sigma_i,\ \text{if } \sigma_i \prec \sigma,\ \text{then } \sigma_i \models \pi_i),\ \text{then } \sigma \models \pi.$$

Informally, a property is locally valid if it is validated by each trace σ ending at it, assuming that each hypothesis π_i is itself validated by all subtraces of σ ending at π_i.

Local Invalidity. A property π is *locally invalid* under a finite set of hypotheses ξ, and we note $\xi \not\models \pi$, if and only if:

$$\text{if } (\forall \pi_i \in \xi, \forall \text{ trace } \sigma_i,\ \text{if } \sigma_i \prec \sigma,\ \text{then } \sigma_i \models \pi_i),\ \text{then } \exists \text{ trace } \sigma, \sigma \not\models \pi.$$

A property is locally invalid if there is a trace σ ending at it but does *not* validate it, but still assuming each hypothesis π_i is valid on any subtrace of σ ending at π_i. These notions of local validity and local invalidity correspond to statuses emitted by Frama-C analyzers as we will see in assumptions 2 and 3 in the next section. Note that being locally invalid is not equivalent to not being locally valid: $\xi \not\models \pi \iff \neg(\xi \models \pi)$. Moreover, none of these predicates is equivalent to $\xi \models \neg\pi$.

Cycles. Statements $\{\pi\} \models \pi$ and $\xi \models \pi_i$ with $\pi_i \in \xi$ are not tautologies in loops. Instead they exactly correspond to proofs by induction, as committed by the strict prefix relation on traces. These statements are actually valid if and only if we can prove $\sigma \models \pi$ (resp. π_i), for any trace σ, under the hypotheses that $\sigma_j \models \pi$ for any *strict* subtrace σ_j of σ (resp. $\sigma_j \models \pi_j$ for any $\pi_j \in \xi$).

Global Validity. Last but not least, a property π is *valid*, and we note $\models \pi$, if and only if $\sigma \models \pi$ for each trace σ. We say that π is invalid, and we note $\not\models \pi$, if π is not valid, that is $\neg(\models \pi)$. Once again, $\not\models \pi \iff \models \neg\pi$. These notions of validity and invalidity correspond to the consolidated statuses computed by our algorithm from all the local validity statuses emitted by the analyzers.

4 Consolidation Algorithm

This section presents a high-level view of the so-called consolidation algorithm implemented in Frama-C. From the *local statuses* of a property π computed by each emitter under hypotheses, each of them corresponding to the local validity or invalidity of π, this algorithm computes its *consolidated status* corresponding to $\models \pi$.

4.1 Local statuses

As already mentioned, an emitter can emit three different local statuses, namely *True*, *False* and *Dont_know*. The third one indicates that it is not able to conclude. Let Λ be the set of these local statuses. Local statuses emitted by analyzers are collected into a database, and we denote $\mathcal{L}_P$ the lookup function that returns them for each property:

$$\mathcal{L}_P : \Phi_P \to \mathscr{P}(\Lambda \times \mathscr{P}(\Phi_P))$$

If an emitter put a local status λ to the property π with hypotheses ξ, then $(\lambda, \xi) \in \mathcal{L}_P(\pi)$. We expect that analyzers are correct and emit local statuses consistently with the underlying annotated program semantics, in particular local validities and local invalidities of annotations. Furthermore, when emitting *False* for a property π, our algorithm also requires that the only possible hypothesis is $\mathrm{reach}(\pi)$. The following assumption formalizes this restriction.

Assumption 2 (Strong Correctness of Analyzers) *We assume that each analyzer is strongly correct: it emits the local status True (resp. False) only for locally valid (resp. invalid) properties under the hypotheses really used (and limited to reachability in case of invalidity). More formally, for each property π of a program P:*

$$\forall (\lambda, \xi) \in \mathcal{L}_P(\pi), \begin{cases} \text{if } \lambda = \textit{True}, \text{ then } \xi \models \pi; \\ \text{if } \lambda = \textit{False}, \text{ then } \xi \not\models \pi \text{ and } \forall \pi_i, \pi_i = \mathrm{reach}(\pi). \end{cases}$$

However, in practice, it may be complicated or inefficient to compute the exact set of hypotheses which is used to compute a local status. Actually, in presence of a blocking semantics, the correctness of the consolidation algorithm does not rely on these hypotheses, as explained by Theorem 1 (correctness of the algorithm). They are useful for Theorem 2 (completeness of the algorithm) and to compute more precise informations for the end-user in the unconclusive cases. Thus, for correctness, the following weaker assumption is enough.

Assumption 3 (Weak Correctness of Analyzers) *Analyzers are assumed to be weakly correct. They emit the local status True (resp. False) only for locally valid (resp. invalid) properties under some unknown hypotheses (resp. reachability). More formally, for each property π of a program P:*

$$\forall (\lambda, \xi) \in \mathcal{L}_P(\pi), \begin{cases} \text{if } \lambda = \textit{True}, \text{ then } \exists \xi' \subseteq \Phi_P, \xi' \models \pi; \\ \text{if } \lambda = \textit{False}, \text{ then } \xi \not\models \pi \text{ and } \forall \pi_i, \pi_i = \mathrm{reach}(\pi). \end{cases}$$

Ensuring correctness of analyzers in practice is out of the scope of this paper[3] and is strongly related to the qualification of verification tools for an operational use in a certified industrial process, for instance with respect to norms like aeronautic's DO-178C. Although none of the Frama-C analyzers is qualified at this time, efforts have already been made in this direction [13, 17].

4.2 Invalidity and Reachability

Unfortunately there is a practical issue with the previous local status *False*: proving local invalidity of a property π requires to prove two different properties: (1) it exists a trace σ which ends at π and (2) this trace does not validates π. Let us examine what tools are able to assess.

- Deductive methods based on weakest precondition calculus are usually not able to prove invalidity. They are only able to prove validity: in practice, they never emit *False*.
- Testing tools usually prove together properties (1) and (2) by exhibiting a test case which invalidates the property: all is fine.
- Abstract interpreters only reason with an over-approximation of all possible traces of the program. Thus, when a property is invalidated for all these over-approximated traces, it means the property is invalid *if* the program point is reachable. But, abstract analyzers are usually not able to prove reachability.

For solving this issue, Frama-C allows emitters to emit either the local status *False_and_reachable* or the local status *False_if_reachable*. So, instead of working with $\mathcal{L}_P$, our algorithm uses the function $\mathcal{L}^\star_P : \Phi_P \to \mathscr{P}(\Lambda^\star \times \mathscr{P}(\Phi_P))$, where $\Lambda^\star$ is the set of emittable statuses defined by:

$$\Lambda^\star \triangleq \{\textit{True}, \textit{Dont\_know}, \textit{False\_if\_reachable}, \textit{False\_and\_reachable}\}.$$

For any program P, thanks to the reach operator, we can automatically compute $\mathcal{L}_P$ from $\mathcal{L}^\star_P$ as follows:

$$\begin{aligned}\forall \pi \in \Phi_P, \mathcal{L}_P(\pi) \triangleq\ & \{(\lambda,\xi) \mid \lambda \in \{\textit{True}, \textit{Dont\_know}\} \text{ and } (\lambda,\xi) \in \mathcal{L}^\star_P(\pi)\} \\ & \cup\ \{(\textit{False},\xi) \mid (\textit{False\_and\_reachable},\xi) \in \mathcal{L}^\star_P(\pi)\} \\ & \cup\ \{(\textit{False},\xi \cup \{\mathrm{reach}(\pi)\}) \mid (\textit{False\_if\_reachable},\xi) \in \mathcal{L}^\star_P(\pi)\}\end{aligned}$$

Emitting *False_and_reachable* is changed into emitting *False*, and emitting *False_if_reachable* is modified into emitting *False* under the additional hypothesis $\mathrm{reach}(\pi)$. Emitting *True* and *Dont_know* is left unchanged. This definition of $\mathcal{L}_P$ preserves both the strong and the weak correctness of analyzers (assumptions 2 and 3).

To avoid an inconsistency of our algorithm in a corner case leading to uncorrectness, we also introduce the following assumption for any $\mathrm{reach}(\cdot)$ property.

Assumption 4 (Do not prove unreachability with reachability) *We assume that no analyzer tries to prove unreachability of a property π by using its reachability. More formally, for a given program P:*

$$\forall \pi \in \Phi_P, \forall (\lambda,\xi) \in \mathcal{L}_P(\mathrm{reach}(\pi)), \textit{if } \lambda = \textit{False}, \textit{ then } \mathrm{reach}(\mathrm{reach}(\pi)) \notin \xi.$$

[3] See the small discussion about the status *Inconsistent* latter in this section however.

4.3 Algorithm

We now introduce the consolidation algorithm itself. Applied on a given program P, it may be seen as a function $\mathcal{S}_P : \Phi_P \to \Sigma$ in which the set Σ is defined by:

$$\Sigma \triangleq \{\mathit{Valid}, \mathit{Invalid}, \mathit{Unknown}, \mathit{Inconsistent}\}.$$

The third status is returned by the algorithm when it is not able to conclude, while the last one is returned when there is both a proof of validity and a proof of invalidity: in such a case, we can conclude that one emitter is not (strongly) correct[4]. Before the formal definition of $\mathcal{S}_P$, we present an informal sketch of the algorithm:

1. abort if assumption 4 is violated;
2. compute the most precise local status λ;
3. for each emitter which emits λ, compute the conjunction of the consolidated statuses of its hypotheses;
4. compute the most precise conjunction γ computed above;
5. compute the status of $\gamma \implies \lambda$;
6. check for inconsistencies.

Step 1 of the algorithm is a simple structural check.

Computing the most precise local status in Step 2 relies on the operator $\bigvee^L$ based on $\vee^L$ which ensures local validity and is defined below. It is mosly equivalent to a logical disjunction in a tri-valued boolean logic. But, in the case where an emitter emits *True* and another one emits *False*, we do not choose yet a status, even if it would be correct to choose *True*: in order to be complete, we wait Step 5 of the algorithm to select the one which is possible to fully consolidate.

$\vee^L$	*True*	*Dont_know*	*False*
True	{ *True* }	{ *True* }	{ *True*, *False* }
Dont_know	{ *True* }	{ *Dont_know* }	{ *False* }
False	{ *True*, *False* }	{ *False* }	{ *False* }

$$\bigvee^L \{\lambda_n\}_n = \begin{cases} \lambda_1 \vee^L \ldots \vee^L \lambda_n & \text{if } n > 0 \\ \mathit{Dont\_know} & \text{otherwise} \end{cases}$$

Computing the conjunction of the statuses of the hypotheses in Step 3 of the algorithm is done by the operator $\bigwedge^H$ based on $\wedge^H$ and defined below. This operator exactly is the standard conjunction of a tri-valued boolean logic. We omit the case *Inconsistent* which is treated as *Unknown* here, and still returns *Unknown*.

$\wedge^H$	*Valid*	*Unknown*	*Invalid*
Valid	*Valid*	*Unknown*	*Invalid*
Unknown	*Unknown*	*Unknown*	*Invalid*
Invalid	*Invalid*	*Invalid*	*Invalid*

$$\bigwedge^H \{\lambda_n\}_n = \begin{cases} \lambda_1 \wedge^H \ldots \wedge^H \lambda_n & \text{if } n > 0 \\ \mathit{Valid} & \text{otherwise} \end{cases}$$

Computing the most precise consolidated status as performed in Step 4 of the algorithm is done by the operator $\bigvee^H$ based on $\vee^H$ and defined below. It exactly corresponds to the standard disjunction of a tri-valued boolean logic. We omit the case *Inconsistent* which never occurs here, since $\wedge^H$ returns *Unknown* instead.

[4] In practice, other origins are possible like inconsistent user-defined ACSL axiomatics.

$\vee^H$	*Valid*	*Unknown*	*Invalid*
Valid	*Valid*	*Valid*	*Valid*
Unknown	*Valid*	*Unknown*	*Invalid*
Invalid	*Valid*	*Invalid*	*Invalid*

$$\bigvee^H \{\lambda_n\}_n = \begin{cases} \lambda_1 \vee^H \ldots \vee^H \lambda_n & \text{if } n > 0 \\ \textit{Unknown} & \text{otherwise} \end{cases}$$

The implication operator $\overset{\text{HL}}{\Longrightarrow}$ involved in Step 5 of the algorithm is defined below (left part in row, right part in column). Like for $\vee^H$, *Inconsistent* is omitted and treated as *Unknown*.

$\overset{\text{HL}}{\Longrightarrow}$	*True*	*Dont_know*	*False*
Valid	*Valid*	*Unknown*	*Invalid*
Unknown	*Unknown*	*Unknown*	*Unknown*
Invalid	*Unknown*	*Unknown*	*Valid*

This operator corresponds to the standard implication of a tri-valued boolean logic, but most cases remain unknown: remember that $\not\models \pi$ means that π is incorrect *for some trace* σ. Thus it would be wrong to assume than an hypothesis π_i being incorrect for some trace σ_i leads to a correct goal *for any trace*: it is still possible to have another trace σ independent of σ_i ($\sigma_i \not\prec \sigma$) which ends at π and invalidates it. It is possible to conclude *Valid* in the case *Invalid* $\overset{\text{HL}}{\Longrightarrow}$ *False* since the only possible hypothesis is $\text{reach}(\pi)$ (assumption 4): if it is invalid, π is unreachable, hence valid.

Step 6 detects inconsistency when it is possible to consolidate a property to both *Valid* and *Invalid*, thanks to the operator $\bigvee^I$ which is equivalent to $\bigvee^H$ except that:

$$\textit{Valid} \vee^I \textit{Invalid} = \textit{Invalid} \vee^I \textit{Valid} = \textit{Inconsistent}.$$

With all the operators now introduced, we can formally define our algorithm as the function $\mathcal{S}_P$ in the following way:

$$\begin{aligned}
\mathcal{S}_P(\pi) &\triangleq \mathcal{S}_P^{\emptyset}(\pi) \\
\text{with } \mathcal{S}_P^{\Psi}(\pi) &\triangleq \bigvee_{\lambda_\pi \in \Lambda_\pi}^{I} \left(\left(\bigvee_{\xi \in \Xi_{\lambda_\pi}}^{H} \bigwedge_{\pi_\xi \in \xi \setminus \Psi}^{H} \mathcal{S}_P^{\Psi \cup \{\pi\}}(\pi_\xi) \right) \overset{\text{HL}}{\Longrightarrow} \lambda_\pi \right) \\
\text{and } \Lambda_\pi &= \bigvee^{L} \{\, \lambda \mid (\lambda, _) \in \mathcal{L}_P(\pi) \,\} \\
\text{and } \Xi_{\lambda_\pi} &= \{\, \xi \mid \mathcal{L}_P(\pi) = (\lambda_\pi, \xi) \,\}.
\end{aligned}$$

In this definition, the set Ψ used in $\mathcal{S}_P^{\Psi}$ stores the properties already visited in order to handle cycles in a well-founded way. This algorithm is correct with respect to the trace semantics of Section 3, as stated by the following theorem[5].

Theorem 1 (Correctness). *Under assumptions 1 (blocking semantics) and 3 (weak correctness of analyzers), if the consolidation algorithm returns* *Valid* *(resp.* *Invalid*)

[5] Proofs are provided in appendix.

for a property π, then π is valid (resp. invalid). If it returns Inconsistent*, then both $\models \pi$ and $\not\models \pi$ hold* (i.e. *logical inconsistency). More formally, for a given program P:*

$$\forall \pi \in \Phi_P, \begin{cases} \text{if } \mathcal{S}_P(\pi) = \textit{Valid}, & \text{then } \models \pi; \\ \text{if } \mathcal{S}_P(\pi) = \textit{Invalid}, & \text{then } \not\models \pi; \\ \text{if } \mathcal{S}_P(\pi) = \textit{Inconsistent} & \text{then } \models \pi \text{ and } \not\models \pi. \end{cases}$$

The algorithm is also *complete* when the analyzers are strongly correct, in the following sense: if a property is assigned a local status of validity (resp. invalidity), and if recursively, all its dependencies are *globally valid*, then our algorithm computes a valid (resp. invalid) consolidated status. The notion of recursively valid hypotheses for property π is captured the following definition:

$$\mathcal{D}(\pi) \triangleq \exists \lambda \neq \textit{Dont\_know}, (\lambda, \xi) \in \mathcal{L}_P(\pi) \text{ and } \forall \pi_i, \mathcal{D}(\pi_i) \text{ and } \models \pi_i;$$

Theorem 2 (Completeness). *Under assumptions 1 (blocking semantics) and 2 (strong correctness of analyzers), if a property is valid (resp. invalid) and an emitter emits a local status different from Dont_know under recursively valid hypotheses, then the consolidation algorithm returns Valid (resp. Invalid). More formally, for a given program P:*

$$\forall \pi \in \Phi_P, \begin{cases} \text{if } \mathcal{D}(\pi) \text{ and } \models \pi, \text{ then } \mathcal{S}_P(\pi) = \textit{Valid}; \\ \text{if } \mathcal{D}(\pi) \text{ and } \not\models \pi, \text{ then } \mathcal{S}_P(\pi) = \textit{Invalid}. \end{cases}$$

5 Consolidated Partial Statuses

The previous formalization provides correctness and completeness results when every property is consolidated to valid or invalid. While this is perfect for the success of a verification campaign under strong qualification requirements, there is no way to know the origin of partial results, in particular in case of *Unknown* statuses.

In Frama-C, there is actually a variety of 11 consolidated statuses that can be synthesized for a property. These statuses provide feedback to the engineer from three complementary points of view: validity of the property, completeness with respect to its recursively valid hypotheses, and reachablity. A color is assigned to each point of view, and each of the 11 statuses of Frama-C has one or two colors for a fully detailed feedback on any property status. This variety of statuses can be simply understood as *refinements* for the four basic consolidated statuses presented in Section 4.

Refinements of Valid. As illustrated in the running example with external functions, it is sometimes impossible to complete a verification process inside the verification tool. It is then useful to consolidate admitted results like valid ones, while tagging those admitted results for manual reviews outside the tool. As seen in the previous section, another important case of validity is a locally invalid but unreachable property. To avoid confusing the user by presenting a *Valid* status on a locally invalid status *False*, we use a special "Invalid but dead" consolidated status in this situation. There is no interesting refinement for *Inconsistent* and *Invalid* statuses.

Valid	Admitted
	Invalid but dead

Inconsistent
Invalid

Refinements of Unknown. The most versatile situations are related to the status *Unknown*. We distinguish several cases by taking into account whether the local status is *True* or *False*, or whether some hypothesis is surely *Invalid*. In the first category of cases, we want to retain the local status feedback although nothing can be claimed since assumptions are missing. In the second category of cases, we want to mark the property as irrelevant since there is an *Invalid* property previously in the control flow graph which may impact the status of this property.

 No local status  Unknown	No analyzer tried. No analyzer succeeded.
 Locally Valid  Locally Invalid	Hypotheses are not yet consolidated (*Unknown*).
 Valid but irrelevant  Unknown but irrelevant	One hypothesis is surely *Invalid*.

Extension of the Algorithm. Extending the consolidation algorithm of Section 4 with this full variety of statuses is quite straightforward. Roughly, an extended status is treated like the status it refines. For instance, *Locally Valid* is treated as *Unknown*. This extension may be synthetized in the modified table of the $\overset{\text{HL}}{\Longrightarrow}$ operator below, which is responsible for consolidating the best local status with respect to the consolidated statuses of its hypotheses. The refined statuses are marked with a star (*).

$\overset{\text{HL}}{\Longrightarrow}$	*True*	*Dont_know*	*False*
Valid	*Valid*	*Unknown*	*Invalid*
Unknown	*Locally Valid* *	*Unknown*	*Locally Invalid* *
Invalid	*Valid but irrelevant* *	*Unknown but irrelevant* *	*Invalid but unreachable* *

The extended table for this operator is sound: a status λ is only replaced by a *refinement* of λ. Hence, we still benefit from correctness and completeness theorems.

6 Conclusion

We have presented a consolidation algorithm for verifying program properties by combining results from several program analyzers. This algorithm is proved to be correct and complete with respect to a generic blocking semantics of annotated programs, as long as analyzers are correct. Its correctness does not rely on hypotheses emitted by the analyzers: these hypotheses are only required for completness. We have also presented how to refine results to provide more informative feedback to the end-user.

This algorithm is fully implemented in Frama-C, a platform gathering several static analysis techniques in a single collaborative framework. It has successfully been used on a confidential 50-kloc case study which is representative of real-life software of systems important to safety in nuclear power plants. Here `Value` is primarily used, while `Wp` helps it to prove assertions on which `Value` is unconclusive. Other collaborations between different set of analyses are currently under way, in particular between test generation tools and static verifiers [18].

Acknowledgement We would like to thank the anonymous referees for their meaningful comments. We also thank Patrick Baudin, Pascal Cuoq, Florent Kirchner, Benjamin Monate, Dillon Pariente, Virgile Prevosto and Boris Yakobowski for the intensive discussions about property statuses, their semantics and their implementation in Frama-C.

References

1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977
2. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8) (1975)
3. Correnson, L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C User Manual. (October 2011) `http://frama-c.com`.
4. Elberzhager, F., Münch, J., Nha, V.T.N.: A systematic mapping study on the combination of static and dynamic quality assurance techniques. Information & Software Technology **54**(1) (2012) 1–15
5. Heintze, N., Jaffar, J., Voicu, R.: A framework for combining analysis and verification. In: POPL 2000
6. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Usable verification of object-oriented programs by combining static and dynamic techniques. In: SEFM 2011
7. Comar, C., Kanig, J., Moy, Y.: Integrating formal program verification with testing. In: ERTSS 2012
8. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. (February 2011) `http://frama-c.cea.fr/acsl.html`.
9. Canet, G., Cuoq, P., Monate, B.: A Value Analysis for C Programs. In: SCAM 2009
10. Correnson, L., Dargaye, Z.: WP Plug-in Manual, version 0.5. (January 2012)
11. Baudin, P., Correnson, L., Hermann, P.: WP Tutorial, version 0.5. (January 2012)
12. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S., Mebsout, A.: The Alt-Ergo Automated Theorem Prover `http://alt-ergo.lri.fr`.
13. Herms, P., Marché, C., Monate, B.: A certified multi-prover verification condition generator. In: VSTTE 2012
14. Grall, H.: Deux critères de sécurité pour l'exécution de code mobile. PhD thesis, École Nationale des Ponts et Chaussées (December 2003) In French.
15. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Information and Computation **207**(2) (2009)
16. Giorgetti, A., Groslambert, J., Julliand, J., Kouchnarenko, O.: Verification of class liveness properties with Java Modeling Language. IET Software **2**(6) (2008)
17. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., Yang, X.: Testing static analyzers with randomly generated programs. In: NFM 2012
18. Delahaye, M., Kosmatov, N., Signoles, J.: Towards a common specification language for static and dynamic analyses of C programs. Submitted.

A Proofs of Theorems

This appendix contains the proofs of the correctness and the completeness theorems of the paper. First we introduce a lemma which links local validity to validity.

Lemma 1 (Local validity implies validity). *Under assumption 1 (blocking semantics), if a property is locally valid (resp. invalid), then it is valid (resp. invalid). More formally, for a given program P:*

$$\forall \pi \in \Phi_P, \forall \Pi \subseteq \Phi_P, \textit{ if } \Pi \models \pi, \textit{ then } \models \pi.$$

Proof. Let π be a property, Π be a finite set of properties such that $\Pi \models \pi$ and σ be a trace. We have to show that $\sigma \models \pi$.

Case $\forall \pi_i \in \Pi, \forall$ **trace** $\sigma_i,$ **if** $\sigma_i \prec \sigma,$ **then** $\sigma_i \models \pi_i$**.** By definition of local validity, $\sigma \models \pi$.

Case $\exists \pi_i \in \Pi, \exists \sigma_i, \sigma_i \prec \sigma$ **and** $\sigma_i \not\models \pi_i$**.** By assumption 1 (blocking semantics), as $\sigma_i \prec \sigma$ and $\sigma_i \not\models \pi_i$, $\sigma \not\leftrightarrow \pi$. Thus, by definition of $\models$ (for trace), $\sigma \models \pi$.

Now, we introduce a well-founded relation which explains why the consolidation algorithm terminates. It is actually not so trivial: informally, our algorithm performs a topological iteration over a graph where vertices are properties and each edge indicates that a property is used as hypothesis of another one. But, this graph may contain cycles, while topological iteration is not well defined for such graphs. That is the *raison d'être* of the set of already visited properties in the algorithm. Thus this set must be taken into account in our proof. So, for any program P, let us introduce the following relation $\ll_P$ over $\Phi_P \times \mathscr{P}(\Phi_P)$ as the transitive closure of $\ll_P^1$ defined as follows:

$$\begin{aligned}(\pi_1, \Psi_1) \ll_P^1 (\pi_2, \Psi_2) &\iff \pi_1 \lessdot_P^1 \pi_2 \text{ and } \pi_1 \notin \Psi_1 = \Psi_2 \cup \{\pi_2\} \\ \text{with } \pi_1 \lessdot_P^1 \pi_2 &\iff \exists (\lambda, \xi_2) \in \mathcal{L}_P(\pi_2), \pi_1 \in \xi_2\end{aligned}$$

Informally, $\pi_1 \lessdot_P^1 \pi_2$ says that π_1 is used as hypothesis of π_2 (or there is an edge from π_2 to π_1 in the graph), while $(\pi_1, \Psi_1) \ll_P (\pi_2, \Psi_2)$ indicates that there is a path from π_2 to π_1 in the graph. It also requires that π_2 is the property currently visited (thus being included in the set of already visited properties) and π_1 is not already visited (in order to break cycles).

Lemma 2 ($\ll_P$ is a well-founded relation). *For any program P, $\ll_p$ is a well founded relation. More precisely: $\ll_P$ is a strict partial order (*i.e. *an anti-reflexive, antisymmetric and transitive relation) and every non-empty subset of $\Phi_P \times \mathscr{P}(\Phi_P)$ has a $\ll_P$-minimal element. Furthermore, the set of all these $\ll_P$-minimals is:*

$$\aleph_P \triangleq \{(\pi, \Psi) \,|\, \forall (\lambda, \xi) \in \mathcal{L}_P(\pi), \xi \setminus \Psi = \emptyset\}.$$

Proof. Consider a program P.

Anti-reflexivity. Since, for any $\pi \in \Phi_P$ and $\Psi \in \mathscr{P}(\Phi_P)$, $\pi \in \Psi \cup \{\pi\}$, $(\pi, \Psi) \not\ll_P (\pi, \Psi)$ by definition of $\ll_P$. Hence $\ll_P$ is anti-reflexive.

Antisymmetry. Let π_1 and π_2 being in Φ_P and Ψ_1 and Ψ_2 being in $\mathscr{P}(\Phi_P)$ such that $(\pi_1, \Psi_1) \ll_P (\pi_2, \Psi_2)$ and $(\pi_2, \Psi_2) \ll_P (\pi_1, \Psi_1)$. By definition of $\ll_P$:

$$\begin{aligned}
&\pi_1 \notin \Psi_1 && \text{since } (\pi_1, \Psi_1) \ll_P (\pi_2, \Psi_2)\\
&\notin \Psi_2 \cup \{\pi_2\} && \text{since } \Psi_1 = \Psi_2 \cup \{\pi_2\} \text{ because of } (\pi_1, \Psi_1) \ll_P (\pi_2, \Psi_2)\\
&\notin \Psi_1 \cup \{\pi_1\} \cup \{\pi_2\} && \text{since } \Psi_2 = \Psi_1 \cup \{\pi_1\} \text{ because of } (\pi_2, \Psi_2) \ll_P (\pi_1, \Psi_1).
\end{aligned}$$

The last line is a contradiction since $\pi_1 \in \{\pi_1\}$. Hence $\ll_P$ is antisymmetric.

Transitivity. Trivial by definition of transitive closure.

Elements of $\aleph_P$ are minimal. Let $(\pi, \Psi) \in \aleph_P$ and $(\pi', \Psi') \in \Phi_P \times \mathscr{P}(\Phi_P)$. Let us prove by contradiction that $(\pi', \Psi') \not\ll_P (\pi, \Psi)$. So let us suppose that $(\pi', \Psi') \ll_P (\pi, \Psi)$.

By definition of $\ll_P$, since $(\pi', \Psi') \ll_P (\pi, \Psi)$, $\pi' \lessdot_P \pi$, it exists π'' such that $\pi' \ll_P \pi''$ and $\pi'' \ll_P^1 \pi$. Thus, $\pi'' \lessdot_P^1 \pi$ and, by definition of $\lessdot_P^1$, it exists $(\lambda, \xi) \in \mathcal{L}_P(\pi)$ such that $\pi'' \in \xi$. By definition of $\aleph_P$, since $\pi \in \aleph_P$, $\pi'' \in \Psi$. However, by definition of $\ll_P^1$, $\pi'' \notin \Psi$, leading to a contradiction.

Only elements of $\aleph_P$ are minimal. Let $(\pi, \Psi) \notin \aleph_P$. Let us prove that it exists a couple $(\pi', \Psi') \in \aleph_P$ such that $(\pi', \Psi') \ll_P (\pi, \Psi)$. Consider the set Σ of sequences $(\pi_n, \Psi_n)_n$ such than $\pi_0 = (\pi, \Psi)$ and $(\pi_i, \Psi_i) \gg_P (\pi_{i+1}, \Psi_{i+1})$. Since, Φ_P is a finite set and, forall $i \geq 0$, $\Psi_i \subset \Psi_{i+1}$ by definition of $\gg_P$, any $(\pi_n, \Psi_n)_n$ is a finite sequence. If σ is a trace of Σ, π is a property and Ψ is a set of properties, we note $\sigma \gg_P (\pi, \Psi)$ the extension of σ with (π, Ψ) such that the resulting trace belongs to Σ. Thanks to this notation, we define a distance δ over traces of Σ as follows:

$$\delta((\pi_n, \Psi_n)_n) = 0 \qquad \text{if } \forall (\pi, \Psi) \in \Phi_P \times \mathscr{P}(\Phi_P), (\pi_n, \Psi_n) \not\gg_P (\pi, \Psi)$$

$$\delta((\pi_n, \Psi_n)_n) = \min \left\{ k \in \mathbb{N} \;\middle|\; \begin{array}{l} \exists (\pi, \Psi) \in \Phi_P \times \mathscr{P}(\Phi_P), \\ \quad \delta((\pi_n, \Psi_n)_n \gg_P (\pi, \Psi)) = k - 1 \end{array} \right\} \text{ otherwise.}$$

Informally, δ measures the minimal distance of a trace to a Σ-sequence of maximal length. Now let us show by induction over δ that any element of any sequence of Σ is $\gg_P$-smaller than, or equal to, some $(\pi', \Psi') \in \aleph_P$: that will prove our goal. Let $\sigma = (\pi_0, \Psi_0) \gg_P \cdots \gg_p (\pi_{n-1}, \Psi_{n-1})$ be a sequence of Σ.

Case $\delta(\sigma) = 0$. By definition of δ, there is no (π_n, Ψ_n) such that $(\pi_{n-1}, \Psi_{n-1}) \gg_P (\pi_n, \Psi_n)$. Hence, by definition of $\ll_P$, either forall $(\lambda, \xi_{n-1}) \in \mathcal{L}_P(\pi_{n-1})$, ξ_{n-1} is the empty set or $\psi_{n-1} = \Phi_P$. In both cases, we can trivially conclude than $(\pi_{n-1}, \Psi_{n-1}) \in \aleph_P$. So the $\ll_P$-smallest element of σ belongs to $\aleph_P$: by transitivity and antisymmetry of $\gg_P$ any other element of σ is $\aleph_P$ $\gg_P$-bigger than (π, Ψ).

Case $\delta(\sigma) > 0$. By definition of δ, there exists (π_n, Ψ_n) such that σ is a prefix of the sequence $\sigma' = (\pi_0, \Psi_0) \gg_P \cdots \gg_P (\pi_n, \Psi_n)$ of length $n + 1$. Since $N - (n + 1) < N - n$, we can apply the induction hypothesis on σ': any element of σ' is $\ll_P$-bigger or equal to some couple $(\pi', \Psi') \in \aleph_P$. Hence, σ too by transitivity of $\gg_P$.

We now introduce the last definition before proving both theorems of the paper. Informally, it restricts the sets of already visited properties to those verifying the implicit

invariants of our algorithm. Let P be a program and π be a property of P. We note Υ_π^n the set of finite sequences $\Psi_n = (\pi_n)_n$ of length n inductively defined by (consider that $\pi = \pi_{n+1}$):

$$\begin{aligned} \Psi_0 &= \emptyset & \\ \Psi_{i+1} &= \Psi_i \cup \{\pi_{i+1}\} & 0 \leq i < n \\ \text{with } (\pi_{i+1}, \Psi_i) &\ll_P (\pi_i, \Psi_{i-1}) & 1 \leq i \leq n \\ \text{and } \exists (\lambda_i, \xi_i) \in \mathcal{L}_P(\pi_i), \lambda_i &\neq \textit{Dont\_know} \text{ and } \pi_{i+1} \in \xi_i & 1 \leq i \leq n. \end{aligned}$$

With these preliminary definitions and properties, we are now able to prove both theorems of the paper.

Theorem 1 (Correctness of the Consolidation Algorithm). *Under assumptions 1 (blocking semantics) and 3 (weak correctness of analyzers), if the consolidation algorithm returns Valid (resp. Invalid) for a given property π, then π is valid (resp. invalid). In case of inconsistency, we can deduce both $\models \pi$ and $\not\models \pi$,* i.e. *an inconsistency. More formally, for a given program P:*

$$\forall \pi \in \Phi_P, \begin{cases} \textit{if } \mathcal{S}_P(\pi) = \textit{Valid}, & \textit{then } \models \pi; \\ \textit{if } \mathcal{S}_P(\pi) = \textit{Invalid}, & \textit{then } \not\models \pi; \\ \textit{if } \mathcal{S}_P(\pi) = \textit{Inconsistent} & \textit{then } \models \pi \textit{ and } \not\models \pi. \end{cases}$$

Proof. We actually prove the following more general result:

$$\forall n \in \mathbb{N}, \forall \pi \in \Phi_P, \forall \Psi_n \in \Upsilon_\pi^n, \begin{cases} \text{if } \mathcal{S}_P^{\Psi_n}(\pi) = \textit{Valid}, & \text{then } \models \pi; \\ \text{if } \mathcal{S}_P^{\Psi_n}(\pi) = \textit{Invalid}, & \text{then } \not\models \pi; \\ \text{if } \mathcal{S}_P(\pi) = \textit{Inconsistent} & \text{then } \models \pi \text{ and } \not\models \pi. \end{cases}$$

Let $n \in \mathbb{N}$. We prove the expected property by $\ll_P$-induction over (π, Ψ_n) (possible by lemma 2, well-foundedness of $\ll_P$):

Case $(\pi, \Psi_n) \in \aleph_P$. Let us prove separately the three expected properties.

Case $\mathcal{S}_P^{\Psi_n}(\pi) =$ *Valid*. We have to prove $\models \pi$. According to the definitions of $\bigvee^I$ and $\overset{\text{HL}}{\Longrightarrow}$, there are two cases.

Case *True* $\in \Lambda_\pi$. By definition of $\bigvee^L$ and $\vee^L$, $(\textit{True}, \_) \in \mathcal{L}_P(\pi)$. Then by assumption 3 (weak correctness of analyzers), it exists $\Pi \subseteq \Phi_P$ such that $\Pi \models \pi$. Hence the expected result by lemma 1 (local validity implies validity).

Case *False* $\in \Lambda_\pi$. By definition of $\bigvee^L$, $(\textit{False}, \_) \in \mathcal{L}_P(\pi)$. Then:

$$\bigvee_{\xi \in \Xi_{\lambda_\pi}}^{H} \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Invalid} \qquad \text{definition of } \overset{\text{HL}}{\Longrightarrow}$$

$$\exists \xi \in \Xi_{\lambda_\pi}, \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Invalid} \quad \text{definition of } \vee^H \text{ and } \bigvee^H.$$

However, since $(\pi, \Psi_n) \in \aleph_P$, $\xi \setminus \Psi_n$ is empty. Hence, by definition of $\bigwedge^H$:

$$\bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid}.$$

Absurd since we previously demonstrated that this conjunction is invalid.

Case $\mathcal{S}_P^{\Psi_n}(\pi) = \textbf{\textit{Invalid}}$**.** We have to prove $\not\models \pi$. According to the definitions of $\bigvee^I$ and $\overset{\text{HL}}{\Longrightarrow}$, $\textit{False} \in \Lambda_\pi$. Then:

$$\bigvee_{\xi \in \Xi_{\lambda_\pi}}^{H} \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \overset{\text{HL}}{\Longrightarrow}$$

$$\exists \xi \in \Xi_{\lambda_\pi}, \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \vee^H \text{ and } \bigvee^H.$$

By assumption 3 (weak correctness of analyzers) which restricts the hypotheses of π when emitting *False* to at best $\{\text{reach}(\pi)\}$, ξ is either empty or $\{\text{reach}(\pi)\}$. Furthermore, since $(\pi, \Psi_n) \in \aleph_P$, $\xi \setminus \Psi_n$ is empty. So there only remains two cases.

Case $\xi = \emptyset$**.** Trivial by assumption 3 (weak correctness of analyzers).

Case $\text{reach}(\pi) \in \Psi_n$**.** Let us first prove that it exists $(\lambda, \xi) \in \mathcal{L}_P(\text{reach}(\pi))$ such that $\lambda = \textit{True}$. Since $\text{reach}(\pi) \in \Psi_n$, and by definition of Ψ_n, there exists $(\lambda, \xi) \in \mathcal{L}_P(\text{reach}(\pi))$ and π' such that $\lambda \neq \textit{Dont\_know}$ and $\pi' \in \xi$. Following assumption 3 (weak correctness of analyzers), π' must be $\text{reach}(\text{reach}(\pi))$. But, if $\lambda = \textit{False}$, that contradicts assumption 4 (do not prove unreachability with reachability). Hence $\lambda = \textit{True}$. So, by assumption 3 (weak correctness of analyzers) followed by lemma 1 (local validity implies validity), $\models \text{reach}(\pi)$. Furthermore, by assumption 3 (weak correctness of analyzers) again, since $(\textit{False}, \{\text{reach}(\pi)\}) \in \mathcal{L}_P(\pi)$, $\{\text{reach}(\pi)\} \not\models \pi$. Hence $\not\models \pi$ by definition of local invalidity.

Case $\mathcal{S}_P^{\Psi_n}(\pi) = \textbf{\textit{Inconsistent}}$**.** Both *True* and *False* belong to Λ_π. So, following both cases which we just proved, we can trivially deduce $\models \pi$ and $\not\models \pi$ which is the expected result.

Case $(\pi, \Psi_n) \notin \aleph_P$**.** Let us prove separately the three expected properties.

Case $\mathcal{S}_P^{\Psi_n}(\pi) = \textbf{\textit{Valid}}$**.** We have to prove $\models \pi$. According to the definitions of $\bigvee^I$ and $\overset{\text{HL}}{\Longrightarrow}$, there are two cases.

Case $\textbf{\textit{True}} \in \Lambda_\pi$**.** This case is exactly equivalent to the same subcase of the basic case of the induction.

Case $\textbf{\textit{False}} \in \Lambda_\pi$**.** Similarly to the same subcase of the basic case of the induction, we get:

$$\exists \xi \in \Xi_{\lambda_\pi}, \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Invalid}.$$

By assumption 3 (weak correctness of analyzers) which restricts the hypotheses of π when emitting *False* to at best $\{\text{reach}(\pi)\}$, ξ is either empty or $\{\text{reach}(\pi)\}$. If $\xi \setminus \Psi_n$ is empty, then the case is absurd (see basic case of the induction). Otherwise, by definition of $\bigwedge^H$, we get

$$\mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\text{reach}(\pi)) = \textit{Invalid}.$$

As $\text{reach}(\pi) \lessdot_P^1 \pi$ and $\text{reach}(\pi) \notin \Psi_n \cup \{\pi\}$, $(\text{reach}(\pi), \Psi_n \cup \{\pi\}) \ll_P (\pi, \Psi_n)$. Furthermore $\Psi_n \cup \{\pi\} \in \Upsilon^n_{\text{reach}(\pi)}$. So we can apply the induction hypothesis on $(\text{reach}(\pi), \Psi_n)$ to deduce $\not\models \text{reach}(\pi)$. So, by definition of $\text{reach}(\pi)$, no trace ends at π. Hence $\models \pi$ by definition of $\models$.

Case $\mathcal{S}_P^{\Psi_n}(\pi) = \textbf{\textit{Invalid}}$**.** We have to prove $\not\models \pi$. According to the definitions of $\bigvee^I$ and $\overset{\text{HL}}{\Longrightarrow}$, $\textit{False} \in \Lambda_\pi$. Then:

$$\bigvee_{\xi \in \Xi_{\lambda_\pi}}^{H} \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \overset{\text{HL}}{\Longrightarrow}$$

$$\exists \xi \in \Xi_{\lambda_\pi}, \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \vee^H \text{ and } \bigvee^H.$$

By assumption 3 (weak correctness of analyzers) which restricts the hypotheses of π when emitting *False* to at best $\{\text{reach}(\pi)\}$, ξ is either empty or $\{\text{reach}(\pi)\}$. If ξ is empty or $\{\text{reach}(\pi)\} \in \Psi_n$, then the proof is the same as the one of the basic case of the induction. Thus the remaining case is $\text{reach}(\pi) \in \xi \setminus \Psi_n$. So $(\text{reach}(\pi), \Psi_n \cup \{\pi\}) \ll_P (\pi, \Psi_n)$. Furthermore $\Psi_n \cup \{\pi\} \in \Upsilon^n_{\text{reach}(\pi)}$: we can apply the induction hypothesis on $(\text{reach}(\pi), \Psi_n \cup \{\pi\})$ to deduce $\models \text{reach}(\pi)$. By definition of $\text{reach}(\pi)$, there exists a trace σ which ends at π. Furthermore, by assumption 3 (weak correctness of analyzers), since $(\textit{False}, \{\text{reach}(\pi)\}) \in \mathcal{L}_P(\pi)$, $\text{reach}(\pi) \not\models \pi$. Hence $\not\models \pi$ by definition of $\not\models$.

Case $\mathcal{S}_P^{\Psi_n}(\pi) = \textbf{\textit{Inconsistent}}$**.** This case is similar to the same subcase of the basic case of the induction.

Theorem 2 (Completeness). *Under assumptions 1 and 2 (strong correctness of analyzers), if a property is valid (resp. invalid) and an emitter emits a local status different from Dont_know under recursively valid hypotheses, then the consolidation algorithm returns Valid (resp. Invalid). More formally, for a given program P:*

$$\mathcal{D}(\pi) \triangleq \exists \lambda \neq \textit{Dont\_know}, (\lambda, \Pi) \in \mathcal{L}_P(\pi) \textit{ and } \forall \pi_i, \mathcal{D}(\pi_i) \textit{ and } \models \pi_i.$$

then

$$\forall \pi \in \Phi_P, \begin{cases} \textit{if } \mathcal{D}(\pi) \textit{ and } \models \pi, \textit{ then } \mathcal{S}_P(\pi) = \textit{Valid}; \\ \textit{if } \mathcal{D}(\pi) \textit{ and } \not\models \pi, \textit{ then } \mathcal{S}_P(\pi) = \textit{Invalid}. \end{cases}$$

Proof. Let P be a program. Let us note, for any property π and set of properties Ψ:

$$\mathcal{D}^\Psi(\pi) \triangleq \exists \lambda \neq \textit{Dont\_know}, (\lambda, \Pi) \in \mathcal{L}_P(\pi) \text{ and } \forall \pi_i, \mathcal{D}^{\Psi \cup \{\pi\}}(\pi_i) \text{ and } \models \pi_i.$$

We actually prove the following more general result:

$$\forall n \in \mathbb{N}, \forall \pi \in \Phi_P, \forall \Psi_n \in \Upsilon_\pi^n, \forall \pi \in \Phi_P,$$
$$\begin{cases} \text{if } \mathcal{D}^{\Psi_n}(\pi) \text{ and } \models \pi, \text{ then } \mathcal{S}_P^{\Psi_n}(\pi) = \textit{Valid}; \\ \text{if } \mathcal{D}^{\Psi_n}(\pi) \text{ and } \not\models \pi, \text{ then } \mathcal{S}_P^{\Psi_n}(\pi) = \textit{Invalid}. \end{cases}$$

Let $n \in \mathbb{N}$. We prove the expected property by $\ll_P$-induction over (π, Ψ_n) (possible by lemma 2, well-foundness of $\ll_P$).

Case $(\pi, \Psi_n) \in \aleph_P$**.** Let λ be a local status different of *Dont_know*, and $\xi = \{\pi_i\}_i$ such than $(\lambda, \xi) \in \mathcal{L}_P(\pi)$ (if no such λ and ξ exist, the expected property is trivially true). We split the proof in two cases according to the value of λ.

Case $\lambda =$ ***True*****.** By definition of $\bigvee^L$ and $\bigvee^I$, $\lambda \in \Lambda_\pi$. By assumption 2 (strong correctness of analyzers), $\xi \models \pi$. Thus, by lemma 1 (local validity implies validity), $\models \pi$. So we have to prove $\mathcal{S}_P^{\Psi_n}(\pi) = \textit{Valid}$. According to the definition of $\aleph_P$, $\xi \setminus \Psi_n = \emptyset$. Then:

$$\bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \textstyle\bigwedge^H$$
$$\bigvee_{\xi \in \Xi_{\lambda\pi}}^{H} \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \quad \text{definition of } \vee^H \text{ and } \textstyle\bigvee^H$$
$$\bigvee_{\xi \in \Xi_\lambda}^{H} \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) \overset{\text{HL}}{\Longrightarrow} \lambda = \textit{Valid} \qquad \text{definition of } \overset{\text{HL}}{\Longrightarrow}.$$

By definition of $\bigvee^I$, $\mathcal{S}_P^{\Psi_n}(\pi)$ is either *Valid* or *Inconsistent*. In the former case, we directly get the expected result. In the latter case, by Theorem 1[6], we get an inconsistency from which we can trivially deduce the expected result.

Case $\lambda =$ ***False*****.** By definition of $\bigvee^L$ and $\bigvee^I$, $\lambda \in \Lambda_\pi$. By assumption 2 (strong correctness of analyzers), $\xi \not\models \pi$. Furthermore, according to the definition of $\aleph_P$, $\xi \setminus \Psi_n = \emptyset$. It follows:

$$\bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \textstyle\bigwedge^H$$
$$\bigvee_{\xi \in \Xi_{\lambda\pi}}^{H} \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \quad \text{definition of } \vee^H \text{ and } \textstyle\bigvee^H)$$
$$\bigvee_{\xi \in \Xi_\lambda}^{H} \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) \overset{\text{HL}}{\Longrightarrow} \lambda = \textit{Invalid} \qquad \text{definition of } \overset{\text{HL}}{\Longrightarrow}.$$

[6] Actually that is not precisely what Theorem 1 says : it expresses a statement for $\mathcal{S}_P$ and not $\mathcal{S}_P^{\Psi_n}$, but the proof of this theorem encloses the proof of the same property for $\mathcal{S}_P^{\Psi_n}$.

By definition of $\bigvee^I$, $\mathcal{S}_P^{\Psi_n}(\pi)$ is either *Inconsistent* or *Invalid*. In the former case, by Theorem 1[7], we get an inconsistency from which we can trivially deduce the expected result. So let us suppose $\mathcal{S}_P^{\Psi_n}(\pi) = \textit{Invalid}$ and prove $\not\models \pi$ to be able to conclude. By assumption 2 (strong correctness of analyzers) which restricts the hypotheses of π when emitting *False* to at best $\{\mathrm{reach}(\pi)\}$, ξ is either empty or $\{\mathrm{reach}(\pi)\}$. Furthermore Since $(\pi, \Psi_n) \in \aleph_P$, $\xi \setminus \Psi_n$ is empty. So it only remains two cases.

Case $\xi = \emptyset$**.** Immediate by assumption 2 (strong correctness of analyzers).

Case $\mathrm{reach}(\pi) \in \Psi_n$**.** Let us first prove that it exists $(\lambda', \xi) \in \mathcal{L}_P(\mathrm{reach}(\pi))$ such that $\lambda' = \textit{True}$. Since $\mathrm{reach}(\pi) \in \Psi_n$, and by definition of Ψ_n, it exists $(\lambda', \xi) \in \mathcal{L}_P(\mathrm{reach}(\pi))$ and π' such that $\lambda' \neq \textit{Dont\_know}$ and $\pi' \in \xi$. Following assumption 2 (strong correctness of analyzers), π' must be $\mathrm{reach}(\mathrm{reach}(\pi))$. But, if $\lambda' = \textit{False}$, that contradicts assumption 4 (do not prove unreachability with reachability). Hence $\lambda' = \textit{True}$. So, by assumption 2 (strong correctness of analyzers) followed by lemma 1 (local validity implies validity), $\models \mathrm{reach}(\pi)$. Furthermore, by assumption 2 (strong correctness of analyzers) again, $\{\mathrm{reach}(\pi)\} \not\models \pi$. Hence $\not\models \pi$ by definition of local invalidity.

Case $(\pi, \Psi_n) \notin \aleph_P$**.** Let λ be a local status different of *Dont_know*, and $\xi = \{\pi_i\}_i$ such than $(\lambda, \xi \in \mathcal{L}_P(\pi)$ and, for each $\pi_i \in \xi$, $\mathcal{D}^{\Psi_n}(\pi_i)$ and $\models \pi_i$ (if no such λ and ξ exist, the expected property is trivially true). We split the proof in two cases according to the value of λ.

Case $\lambda = \textit{True}$**.** By definition of $\bigvee^L$ and $\bigvee^I$, $\lambda \in \Lambda_\pi$. By assumption 2 (strong correctness of analyzers), $\xi \models \pi$. Thus, by lemma 1 (local validity implies validity), $\models \pi$. So we have to prove $\mathcal{S}_P^{\Psi_n}(\pi) = \textit{Valid}$. For each $\pi_i \in \xi \setminus \Psi_n$, $(\pi_i, \Psi_n \cup \{\pi\}) \ll_P (\pi, \Psi_n)$. Furthermore $\Psi_n \cup \{\pi\} \in \Upsilon^n_{\pi_i}$: we can apply the induction hypothesis to each $\pi_i \in \xi \setminus \Psi_n$ to deduce $\mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_i) = \textit{True}$ (since $\models \pi_i$ and $\mathcal{D}^{\Psi_n \cup \{\pi\}}(\pi_i)$). Then:

$$\bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \quad \text{definition of } \wedge^H \text{ and } \bigwedge^H$$

$$\bigvee^H_{\xi \in \Xi_{\lambda_\pi}} \bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \quad \text{definition of } \vee^H \text{ and } \bigvee^H$$

$$\bigvee^H_{\xi \in \Xi_\lambda} \bigwedge^H_{\pi_\xi \in \xi \setminus \Psi_n} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) \overset{\text{HL}}{\Longrightarrow} \lambda = \textit{Valid} \quad \text{definition of } \overset{\text{HL}}{\Longrightarrow}.$$

By definition of $\bigvee^I$, $\mathcal{S}_P^{\Psi_n}(\pi)$ is either *Valid* or *Inconsistent*. In the former case, we directly get the expected result. In the latter case, by Theorem 1[8], we get an inconsistency from which we can trivially deduce the expected result.

[7] See footnote 6.

[8] See footnote 6.

Case $\lambda = \textit{False}$**.** By definition of $\bigvee^L$ and $\bigvee^I$, $\lambda \in \Lambda_\pi$. By assumption 2 (strong correctness of analyzers), $\xi \not\models \pi$ and ξ is either empty or $\{\text{reach}(\pi)\}$. We have to show $\mathcal{S}_P^{\Psi_n}(\pi) = \textit{Invalid}$. If ξ is empty or $\{\text{reach}(\pi)\} \in \Psi_n$, then the proof is the same as the one of the basic case of the induction. Thus the remaining case is $\text{reach}(\pi) \in \xi \setminus \Psi_n$. So $(\text{reach}(\pi), \Psi_n \cup \{\pi\}) \ll_P (\pi, \Psi_n)$. Furthermore $\Psi_n \cup \{\pi\} \in \Upsilon^n_{\text{reach}(\pi)}$: we can apply the induction hypothesis on $(\text{reach}(\pi), \Psi_n \cup \{\pi\})$ to deduce $\mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\text{reach}(\pi)) = \textit{Valid}$ (since $\models \text{reach}(\pi)$ and $\mathcal{D}^{\Psi_n \cup \{\pi\}}(\text{reach}(\pi))$). Then:

$$\bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \wedge^H \text{ and } \bigwedge^H$$

$$\bigvee_{\xi \in \Xi_{\lambda\pi}}^{H} \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) = \textit{Valid} \qquad \text{definition of } \vee^H \text{ and } \bigvee^H$$

$$\bigvee_{\xi \in \Xi_\lambda}^{H} \bigwedge_{\pi_\xi \in \xi \setminus \Psi_n}^{H} \mathcal{S}_P^{\Psi_n \cup \{\pi\}}(\pi_\xi) \overset{\text{HL}}{\Longrightarrow} \lambda = \textit{Invalid} \qquad \text{definition of } \overset{\text{HL}}{\Longrightarrow}.$$

By definition of $\bigvee^I$, $\mathcal{S}_P^{\Psi_n}(\pi)$ is either *Invalid* or *Inconsistent*. In the former case, we directly get the expected result. In the latter case, by Theorem 1[9], we get an inconsistency from which we can trivially deduce the expected result.

[9] See footnote 6.