Formalizing Communication Between Numerical Abstract Domains*

Jacques-Henri Jourdan

Inria Paris-Rocquencourt

Abstract. In the context of the formally verified static analyzer Verasco, we report on the implementation and formalization, using the Coq proof assistant, of a communication system for extensible combinations of numerical abstract domains which is simpler, more flexible and more modular than the full reduced product. Information can either be shared on the initiative of the sender or queried by the receiver. Through examples of use, we demonstrate that this system allows a modular and precise combination of abstract domains.

1 Introduction

Static analysis tools are being increasingly used in software industry. They use many different technologies, such as, abstract interpretation, model checking, deductive program verification and interactive theorem proving. Some of them aim at finding bugs in large and complex code bases. In contrast, in this paper, we focus on tools that give formal properties about software.

These tools face two challenges. Their soundness is essential, since their results are actually used to increase the user confidence in the software. Nevertheless, critical software can be very complex, and proving properties on them can require advanced arguments. This explains why static analyzers based on abstract interpretation, like Astrée [1], contain several abstract domains that need to communicate with each other.

Verasco [9] is a C static analyzer based on abstract interpretation [3,4]. It aims at proving the absence of undefined runtime behavior from programs written in a large subset of C99. In order to address the soundness challenge, Verasco has the particularity of being entirely programmed and formally verified using the Coq proof assistant. Its implementation comes with a formally proved soundness theorem stating that if the static analyzer returns no alarm, then the program will not have any undefined behavior. This theorem uses the formal semantic of an intermediate language of the CompCert formally verified compiler [10], so that, combined with the CompCert soundness theorem, Verasco gives formal guaranties about the generated machine code.

Several abstract domains were needed in order to be able to check non-trivial programs without raising alarms [9]. One deals with memory and pointers; one

* This work was supported by Agence Nationale de la Recherche, grant ANR-11-INSE-003.

handles machine-integer arithmetic; and others build precise approximations of numerical environments. As an example, in the current version of our analyzer, the hierarchy of numerical domains contains a simple symbolic abstract domain, a congruence abstract domain, an interval abstract domain, and, optionally, a polyhedral abstract domain for integer variables.

In this paper, we focus on the combination of numerical abstract domains in Verasco, and especially on the formalization of communication between them. It is well-known that the full reduced product [4], where each domain receives the maximal amount of information from others, is highly non-modular. Its implementation requires knowing exactly in advance which domains are going to be combined. Moreover, the reduced product may not even exist when the abstract domains have no best abstraction function, or its computation may be too hard in practice. On the other extreme, direct products, where each domain reason independently of the others are too weak; for example, in our analyzer, the interval domain is the only numerical domain able to deal with all numerical operations (including bitwise and floating-point operations), while other domains may be able to use this information. We need an intermediate solution where domains can communicate with others, while keeping the analyzer easy to maintain from the point of view of its programmer. It should be easy to modify the hierarchy by adding, removing or replacing domains. Even though we do not expect the system to be as precise as the reduced product, we need the communication mechanism itself to be easily extensible, so that the sources of imprecision can be mitigated

Following some of the ideas of Astrée [5], we describe a system of communication channels between numerical domains. We use this system to share information between domains in a modular way. Information can be either queried by a domain from another domain, or notified by sending a message between domains. Our contribution is twofold: first, we improved this system and proved it correct using the Coq proof assistant; second, we used it to enable the communication between the different domains of Verasco. We believe that those ideas are very flexible and can be easily adapted in the context of different static analyzers using several abstract domains.

We first present our system of communications (§2), before giving some examples of uses we have in Verasco (§3). Then, we discuss related work (§4) and conclude (§5).

2 Combining Numerical Domains Using Products and Channels

2.1 Context : Numerical Abstract Domains in Verasco

We are interested in (possibly relational) numerical abstract domains. They concretize to numerical environments, where values are elements of type `num`, which are either unbounded integers or IEEE 754 double precision floating-point numbers.

Given a type of variables $\mathcal{V}$, an abstract environment is represented by a Coq type `t` with a concretization function $\gamma : \texttt{t} \to \mathcal{P}(\mathcal{V} \to \texttt{num})$, where $\mathcal{P}(\tau)$ denotes $\tau \to$ `Prop`, the type of sets of elements of type τ. Thus, for an abstract environment `ab` and a concrete environment ρ, we will write γ `ab` ρ, or, equivalently, $\rho \in \gamma$ `ab`, to denote the fact that ρ is in the concretization of `ab`.

Abstract environments implement the standard semi-lattice operations: top, comparison and join. They have standard soundness properties, taken from abstract interpretation theory:

`top_correct`: $\forall\ \rho,\ \rho \in \gamma(\top)$;

`leb_correct`: $\forall$ `ab1 ab2` $\rho,\ $ `ab1` $\sqsubseteq$ `ab2` $=$ `true` $\to\ \ \rho \in \gamma$ `ab1` $\to\ \rho \in \gamma$ `ab2`;

`join_correct`: $\forall$ `ab1 ab2` $\rho,\ \ \rho \in \gamma$ `ab1` $\vee\ \rho \in \gamma$ `ab2` $\to\ \ \rho \in \gamma$ (`join ab1 ab2`)

Abstract environments also provide abstract transfer functions. For example, if no domain cooperation were involved, the `assign` operation, which models variable assignments would have the following type:

`assign`: $\mathcal{V} \to$ `expr` $\to$ `t` $\to$ `t`;

Its specification (using the `eval_expr` predicates which gives semantics to expressions) says that the concretization of the result of an abstract assignment contains the corresponding assigned concrete environments:

`assign_correct`:
$\forall\ \rho$ `ab`, $\rho \in \gamma$ `ab` $\to$
$\forall$ `e n`, `n` $\in$ `eval_expr` ρ `e` $\to$
$\forall$ `x`, ρ+[`x` $\Rightarrow$ `n`] $\in \gamma$ (`assign x e ab`);

Here `eval_expr` ρ `e` represents the set of values an expression can have in a given environment.

Our notion of abstract domain is weaker than the typical one based on Galois connections [4]. Indeed, our goal is to prove formally the soundness of the analyzer, and the questions of optimality is not essential, as long as the analyzer performs well in practice. This is the reason why the abstraction function α is not included in our development; but only the concretization function γ is. Lattice operations are provided, but the only properties provided by the abstract domains are soundness properties relative to γ. Likewise, the termination of the analyzer, although ensured informally, is not proved in our development. Thus, while a `widen` function is provided, it has no specification and is only used as an oracle to accelerate convergence.

In the Verasco static analyzer, we use a hierarchy of such numerical abstract domains. They are combined using a product functor, taking two numerical abstract domains, and returning a numerical abstract domain, containing information from both domains. It is important to understand that the product between domains is not symmetric: the abstract operations are computed from left to right, and the information made available by the right component is not available when computing the left component.

2.2 Sharing the Bottom Element

The first cooperation between domains is the sharing of the bottom element. Given a type τ of an abstract domain, we define $\tau+\bot$ as the type τ with one additional $\bot$ element. Then, in the implementation of abstract domains, this new $\bot$ element is returned when a transfer function finds a contradiction. When manipulating a product of abstract domains `A` and `B`, its reduced version $(\texttt{A}*\texttt{B})+\bot$ is used instead of the direct product $(\texttt{A}+\bot) * (\texttt{B}+\bot)$, .

Following this idea, the type of the `assign` transfer function, for example, becomes:

`assign`: $\mathcal{V} \to$ `expr` $\to$ `t` $\to$ `t`$+\bot$

Its specification is written exactly the same way as previously. It can be noted that the γ function is used for different abstract objects: for the type `t` of abstract environments, but also for `t`$+\bot$. Coq disambiguates this notation thanks to a type class for concretization functions, together with generic instances. If there is a concretization function for the type τ, then a concretization for $\tau+\bot$ is defined by takgin $\gamma(\bot) = \emptyset =$ `fun _ ⇒ False`

It is worth noting that $+\bot$ can be seen as an error monad so that standard monadic operators (along with the `do` notation, inspired from Haskell) are used to simplify the code. Given the implementation of `assign` for abstract domains `A` and `B`, it is straightforward to define it for `A` $*$ `B`, returning an element of (`A` $*$ `B`)$+\bot$:

```
assign_{A*B} x e ab :=
  let '(a, b) := ab in
  do a ← assign_A x e a;
  do b ← assign_B x e b;
  NotBot (a, b)
```

The proof of soundness of this new definition is trivial.

This technique is a simple system of abstract domain cooperation. Indeed, when a domain finds a contradiction, it is shared between all domains. As it is very limited, a more evolved system is clearly needed to share more precise information.

2.3 Message Channels

Message channels are another kind of cooperation. The general idea is that when an abstract domain assesses that some new fact may be useful for other domains, it sends a message containing this new information. A message is an element of a Coq inductive type, customizable depending on the kind of information that needs to be shared. It is given a concretization function, which extends to message channels, defined as messages lists. For example, if we needed to define a message `Known_value_msg x v` stating that the actual value of variable `x` is known to be `v`, we would write:

```
Inductive message : Type :=
  | Known_value_msg: 𝒱 → num → message
  [...].

Definition message_gamma (m:message): 𝒫(𝒱 → num) :=
  match m with
  | Known_value_msg x v ⇒ fun ρ ⇒ ρ x = v
  [...]
  end.

Definition message_chan := list message.

Definition message_chan_gamma : (mc:message_chan): 𝒫(𝒱 → num) :=
  fun ρ ⇒ List.Forall (fun m ⇒ ρ ∈ γ m) mc.
```

Each abstract operator returning an abstract environment (including abstract transfer functions and lattice operators) now also returns a message channel:

```
assign: 𝒱 → expr → t → (t * message_chan)+⊥
```

This `assign` function still has the exact same specification as in §2.1. The concretization of a pair is defined as the intersection of concretizations of its components. That is, when this function returns a pair (`ab`, `chan`) of an abstract environment and a message channel, the corresponding concrete environments are elements of both $\gamma($`ab`$)$ and $\gamma($`chan`$)$[1].

Then, in every domain, a new primitive is implemented to process messages. The role of this primitive is to internalize the information brought by a message given as argument inside an abstract environment given as a second argument. Its type and specification follows:

```
process_msg: message → t → (t * message_chan)+⊥;
process_msg_correct:
  ∀ m ρ ab, ρ ∈ γ ab → ρ ∈ γ m → ρ ∈ γ (process_msg m ab)
```

It is worth noting that this function returns a message channel. While processing a message, it can decide to forward it, to discard it (for example when it already knows this piece of information, so that the following abstract domains in the hierarchy should already have received this particular piece of information), or to make it more precise before forwarding.

In the product functor, the abstract operations are executed in order, and then the messages produced by the first component are sent to the second component. To this end, we first define a function that extends `process_msg` to message lists, by successively applying it, and concatenate returned messages channels:

[1] There is no constraint on whether $\gamma($`ab`$)$ should be a subset of $\gamma($`chan`$)$. While expected to be true in practice for most implementations of most operations, this property is wrong for typical implementations of `process_msg`.

```
Fixpoint process_msg_list_B (mchan:list message) (accu:B * message_chan)
          : (B*message_chan)+⊥ :=
  match mchan with
    | nil ⇒ NotBot accu
    | cons m mchanq ⇒
      do_bot (ab, mchan') ← process_msg m (fst accu);
      process_msg_list mchanq (ab, mchan'++snd accu)
  end.
```

Then, the abstract operations of the product functor are defined. As an example, the definition for `assign` is:

```
assign_A*B x e ab :=
  let '( a,  b) := ab in
  do (a,  mchana) ← assign_A x e a;
  do (b,  mchanb) ← assign_B x e b;
  do (b,  mchan) ← process_msg_list_B mchana (b, mchanb);
  NotBot ((a,  b),  mchan)
```

Broadcasting messages. Note that in this setting, the information cannot be propagated from the last domains to the first ones. To fix this limitation, we introduce another sort of messages, broadcast messages, which embed another message, and keeps its concretization unchanged:

```
Inductive message : Type :=
  [...]
  | Broadcasted_msg : message → message
  [...].

Fixpoint message_gamma (m:message): 𝒫(𝒱 → num) :=
  match m with
  [...]
  | Broadcasted_msg m ⇒ message_gamma m
  [...]
  end.
```

Domains simply forward those messages without analyzing their content. Then, when such a message returns to the root of the domain hierarchy, it is unboxed and sent back to the whole hierarchy. In order to avoid termination problems, broadcasting messages produced during this second step are not processed. This second step is simply ignored during widening, because even if the left-to-right information propagation cannot break fixpoint computation termination, this broadcasting mechanism could.

Code maintenance for message channels. From the point of view of the code maintainer, we believe the message channels are very convenient. They provide a simple, powerful abstraction for domain cooperation, and they are very flexible. In order to add some new kind of communication, one can just add

a new constructor to the `message` type, give it a concretization, and implement and prove correct the sender and receiver. If removing or replacing an abstract domain is needed, modifying the hierarchy is possible. Then, the messages that were processed by the changed domain will simply be ignored.

2.4 Query Channels

The message channel approach is useful, but an additional mechanism is needed. Indeed, when a message comes from another domain, the receiver may not be able to internalize the information at the time of reception. For example, if a congruence domain sends a message stating that some variable has remainder 2 modulo 5, the interval domain is not able to express this constraint using its own abstraction. Thus, in the future, when it is going to learn the new interval $[2, 15]$ for this variable, it will not be able to use the previously received congruence message to refine it to $[2, 12]$. A solution would be for the interval domain to query, when needed, the congruence information for this variable.

This example shows that sometimes, a domain may need to *query* another domain for some information. That is, the initiative of sharing information may come from the receiving domain. This leads to our second kind of channels: query channels. A query channel is an element of a record type, whose fields are functions that may be called to query some information. For example, in Verasco, a component of query channels returns the interval of values an expression may take. Query channels have their concretization function; informally, an environment is in the concretization of a query channel if every query answer is valid for this environment:

```
Record query_chan : Type :=
  { get_itv: expr → interval+⊥; [...] }.

Record query_chan_gamma (chan:query_chan) (ρ:𝒱→num) : Prop :=
  { get_itv_correct: ∀ e, eval_expr ρ e ⊆ γ (chan.(get_itv) e); [...] }.
```

Each domain implements a function, `enrich_query_chan`, which refines the answers given by a query channel with the information contained in the abstract environment:

```
enrich_query_chan: t → query_chan → query_chan;
enrich_query_chan_correct:
  ∀ ab chan ρ, ρ ∈ γ ab → ρ ∈ γ chan → ρ ∈ γ (enrich_query_chan ab chan)
```

When a query is issued to the returned query channel, it can either decide to directly forward the query to the query channel given as argument (typically when the query has nothing to do with the kind of information the enriching domain stores), or answer it directly, or a combination. For example, an interval domain can forward a `get_itv` query to the base channel, and then intersect the result with a value it computes itself.

The implementation of `enrich_query_chan` for products of domains is a straightforward composition of implementations for both components. Then, for computing the query channel associated with a hierarchy of abstract domains,

it suffices to call `enrich_query_chan` with a dummy query channel, whose concretization is the whole concrete domain.

To enable domains to query, instead of giving an abstract environment as parameter of abstract operations, a pair of an abstract environment and a query channel is passed, abstracting the same set of concrete environments. This query channel contains the contribution of every abstract domains. The type of `assign` thus becomes:

```
assign: 𝒱 → expr → t * query_chan → (t * message_chan)+⊥;
```

On the other hand, when an abstract operation returns an abstract environment, an abstract domain may want to query the other domains about the concrete states whose abstraction is *currently being computed*. For example, when evaluating an abstract assignment, a domain may want to query other domains about the new value of the variable. For this reason, a query channel corresponding to the *result* of the operation is also given as an argument. Note that, as not every abstract domains have computed the operation, this query channel is partial and contains only the contributions of the domains located before it in the hierarchy. The final type and specification of the `assign` function follows:

```
assign: 𝒱 → expr → t * query_chan → query_chan → (t * message_chan)+⊥;
assign_correct:
  ∀ x e ρ n ab chan,
    n ∈ eval_expr ρ e → ρ ∈ γ ab →
    ρ+[x ⇒ n] ∈ γ chan →
    ρ+[x ⇒ n] ∈ γ (assign x e ab chan);
```

Every abstract operation can be adapted similarly, except the comparison operator. Without query channel, its type would be $\sqsubseteq$: `t → t → bool`, with the specification given in §2.1. So, logically, its type would be, with query channels, $\sqsubseteq$: `t * query_chan → t * query_chan → bool`. However, the query channel for the second argument would forbid any correct non-trivial implementation. Indeed, to prove an implementation returning `true` for some inputs, it would be necessary to prove that for those inputs, the concretization of the left-hand side is included in the concretization of the query channel of the right-hand side (which is impossible without enumerating every possible queries). Thus, instead, the comparison operator does not take a query channel in its second argument:

```
⊑: t * query_chan → t → bool;
leb_correct: ∀ ab1 ab2 ρ,
  ab1 ⊑ ab2 = true →
  ρ ∈ γ ab1 → ρ ∈ γ ab2;
```

This signature can also be viewed another way: an abstract domain answer whether it is able to deduce the properties corresponding to the right-hand side from properties of the left-hand side abstract environment and query channel.

The implementation and proof of the product functor, combining both message channels and query channels are a bit tedious, but do not present noticeable difficulties. As an example, we give the implementation of the `assign` function, obtained simply by adding query channels to the implementation given in §2.3:

```
assign_{A*B} x e ab qchan :=
  let '(( a,  b),  abqchan) := ab in
  do (a,  mchana) ← assign_A x e (a, abqchan) qchan;
  let qchan := enrich_query_chan a qchan in
  do (b,  mchanb) ← assign_B x e (b, abqchan) qchan;
  do (b,  mchan) ← process_msg_list_B mchana (b, qchan) mchanb;
  NotBot ((a, b),  mchan)
```

Code maintenance for query channels. Query channels have similar advantages as message channels. It is simple to add a new kind of query. To this end, one has to add a field to the `query_chan` record, update its concretization function, and provide a dummy implementation. Similarly, it is robust to replacements or deletions in the domains hierarchy.

3 Examples of Uses of Channels

We claim the channel system we described is very flexible and can be used to implement various kind of cooperation. In this section, we explain how it is used in the Verasco static analyzer. In particular, we describe three use cases: communications between the front-end of the analyzer and numerical domains (§3.1), the use of a simple symbolic domain to reconstruct Boolean expressions (§3.2), and a weak reduced product between a non-relational congruence domain and an interval domain (§3.3).

3.1 Channels Used by the Analyzer Front-End

While originally designed for communications inside the numerical part of the Verasco analyzer, channels are also used in Verasco by the analyzer front-end: these mechanisms supersede some primitive that would be needed otherwise, simplifying the interface.

In a previous version of Verasco, we had an `assume` primitive in the numerical abstract domain interface. Its role was to make the front-end able to inform the numerical domains that some condition is met (or not), typically when entering either branch of an `if` statement. Message channels replace this `assume` primitive: a special kind of messages is used instead. It is only sent by the upper layers of Verasco when they would have called `assume`. This message reflects that a concrete Boolean expression evaluates to a known Boolean:

```
Inductive message : Type :=
  [...]
  | Fact_msg : expr → bool → message
  [...].

Fixpoint message_gamma (m:message): 𝒫(𝒱 → num) :=
  match m with
  [...]
  | Fact_msg c b ⇒ fun ρ ⇒ num_of_bool b ∈ eval_expr ρ c
  [...].
```

Then, the domains have to implement the processing of those messages, just as they would have to implement an `assume` function.

Similarly, query channels are used in Verasco when the front-end needs to get some information from the numerical domains. Typically, for example, the abstract domain functor of Verasco dealing with machine-integer arithmetic will issue interval queries to analyze the overflow behavior of some arithmetic expressions.

3.2 Reconstructing Boolean Expressions

The Verasco static analyzer uses C as its input language. In C, expressions may have side-effects. However, in order to have simpler abstractions, the abstract domains in the analyzer only deal with pure expressions. To bridge this gap, we use a compilation pass, shared with the CompCert compiler, which splits expressions to extract side effects. One important example of side effects in expressions that is processed by this compilation pass are Boolean expressions. Indeed, being lazy in the C language, the `&&` and `||` operators are transformed into control flow. While needed to preserve the purity invariant, this transformation can significantly decrease the precision of the analysis.

For example, consider the following piece of code:

```
int x = any_int();
if(x > 3 && x < 10) {
  verasco_assert(x > 3);
  verasco_assert(x < 10);
}
```

After the compilation pass, it is transformed into a low-level equivalent of:

```
int x = any_int(), b;
if(x > 3) { b = x < 10; } else { b = 0 };
if(b) {
  verasco_assert(x > 3);
  verasco_assert(x < 10);
}
```

This is difficult to analyse, because some relational information between `x` and `b` is needed when analyzing the second `if` statement in order to discharge the assertions. Moreover, to recover full precision, a linear numerical abstract domain is of no help.

To solve this problem, we designed an abstract domain of symbolic equalities. This domain records symbolic equalities of the form $x = e_c$ where x is a variable and e_c is an expression of type `ite_expr`, as well as facts of the form $e_c =$ `true` or $e_c =$ `false`. The type `ite_expr` is an extension of the type `expr` of pure expressions, with if-then-else constructs in prenex position:

```
Inductive ite_expr : Type :=
  | Leaf : expr → ite_expr
  | Ite : ite_expr → ite_expr → ite_expr → ite_expr.
```

The domain is able to build precise equations for the kind of Boolean temporary variables that are generated by the preprocessing of Boolean expressions. In our example, it will infer `b = x > 3 ? x < 10 : 0`. Then, we use the query channel to transfer this information to other domains. More precisely, we add a new kind of query returning an expression known to evaluate to the value of the given variable:

```
Record query_chan : Type :=
  { [...]
    get_eq_expr: 𝒱 → option ite_expr;
    [...]   }.

Record query_chan_gamma (chan:query_chan) (ρ:𝒱→num) : Prop :=
  { [...]
    get_eq_expr_correct:
      ∀ x e, chan.(get_eq_expr) x = Some e → eval_ite_expr ρ e (ρ x);
    [...]   }.
```

These queries are answered by the symbolic equalities abstract domain by looking for equations involving the variable passed in parameter.

Then, other abstract domains can use these queries to improve precision by unfolding some variables. In particular, when processing a `Fact_msg`, for an expression containing only a variable, it will automatically unfold it in the hope of reconstituting a Boolean expression. In order to handle nested Boolean expressions, this process is recursive up to a predetermined maximum depth: if the leaf of some returned `ite_expr` is a variable, it will be unfolded as well.

3.3 Intervals and Congruences Weak Reduced Product

In Verasco, we use two different non-relational abstract domains: congruences [7] and intervals. While both domains are useful in finding some properties themselves, cooperation can further refine inferred invariants. As a simple example, consider the following C code fragment:

```
for(int i = 2; i < 16; i += 5)
  verasco_assert(i <= 12);
```

In order to discharge the obligation `i <= 12` without unrolling the loop, the analyzer has to infer the invariant `i` mod $5 = 2$, which is what the congruence domain is designed for; and then use this information to refine the interval for `i` known from the loop bounds.

The first step towards cooperation is the use of the `Known_value_msg` message described in §2.3: as soon as one domain knows the value for a variable, typically when assigning a variable or receiving a `Fact_msg` message, it will broadcast a `Known_value_msg` to every other domains. When a domain (in particular the interval domain or the congruence domain) receives such a message, it will check the consistency of the value with its internal state, and refine its abstraction. However, in the above example, another mechanism is needed, as no variable has a statically known value.

The full cooperation mechanism involves both message and query channels. The communication can be initiated by the congruence domain: when it learns a new congruence value for a variable, the congruence domain will send a message in the hope it can refine the interval for this variable. The initiative may also come from the interval domain: when it learns a new interval for a variable, it will query the congruence domain to refine it.

More precisely, we add a new message constructor `Congr_msg`, carrying congruence information for a variable:

```
Inductive message : Type :=
  [...]
  | Congr_msg : 𝒱 → congruence → message
  [...].

Fixpoint message_gamma (m:message) : 𝒫(𝒱→num) :=
  match m with
  [...]
  | Congr_msg x c ⇒ fun ρ ⇒ ρ x ∈ γ c
  [...]
  end.
```

We also add a new query, which, similarly to `get_itv`, let domains ask for congruence information on a given expression:

```
Record query_chan : Type :=
  { [...]
    get_congr: expr → congruence+⊥;
    [...]   }.

Record query_chan_gamma (chan:query_chan) (ρ:𝒱→num) : Prop :=
  { [...]
    get_congr_correct: ∀ e, eval_expr ρ e ⊆ γ (chan.(get_congr) e);
    [...]   }.
```

When processing an assignment, if the congruence domain is able to get a non-trivial abstract value for the variable, it will send a `Congr_msg` message for this variable. Similarly, after processing a message, it will send a `Congr_msg` for

each improvement in the abstract value of a variable. When receiving such a message, the interval domain will attempt to refine its interval for the variable in question by using the property:

$$\begin{cases} x \in [a, b] \\ x \bmod m = r \end{cases} \implies c \in [a + (r - a) \bmod m, \quad b - (b - r) \bmod m] \qquad (1)$$

Conversely, when processing a message, if the interval domains learns a new interval for an integer variable, it will call `get_congr` and use (1) to refine it.

This process makes Verasco able to properly analyze the previous example. Indeed, when analyzing the loop exit condition, the analyzer front-end will issue a `Fact_msg`. Its processing by the interval domain will produce the constraint $\texttt{i} \in [2, 15]$. After querying the congruence domain, the interval domain will deduce that $\texttt{i} \in [2, 12]$ in the loop body.

We call this sharing of information between both domains a *weak* reduced product, because abstract values are only reduced for variables, but not for sub-expressions. However, for most cases, this loss in precision is acceptable.

Another approach would be to implement the reduced product directly at the level of non-relational abstract values. Our communication system has a major advantage over this alternative. It keeps both domains separated, thus the architecture stays modular. This is important since in Verasco, the implementation of intervals is complex: they can be floating-point intervals, and they deal with every operations of the language (including bitwise operations, for example). Moreover, one can easily remove or modify one of the two domains without caring too much about the interaction with the other. Finally this is a small addition in those domains, and the change is very localized.

4 Related Work

Combining abstract domains in order to get more precise abstract interpreters is not a new idea: Cousot and Cousot proposed this idea in one of the very first papers on abstract interpretation [4]. They described the direct product and the reduced product. However, as we explained, neither product is adapted to our setting: direct product is not precise enough, and reduced product is not practical, leading to non-modular, and hard to maintain code.

The Nelson-Oppen procedure [12] is a decision procedure for combinations of decidable logic theories. Several authors [6,8] tried to adapt this method to abstract interpretation. Compared to ours, this method has several drawbacks in the context of static analysis. First, its theoretical foundations requires that function and predicate symbols of both theories must be disjoint (except for the equality predicate), so that the only pieces of information that is shared are (in-)equalities between expressions. Second, the reduction phase used by this procedure have a higher computational complexity. Lastly, because they rely on saturating the set of shared (in-)equalities, these methods do not include a way for an abstract domain to query others.

Corstesi, Le Charlier and Van Hentenryck [2] introduced the notion of *open product*, which is a refinement of the direct product, based on queries. This idea is close to our notion of query channels. However, their extensions to the notion of abstract operation, with additional query parameters, has several limitations. First, it prevents the implementation of some operation to query other domains about the result of the operation. For this kind of communication, they rely on a refinement step after the operation. In our implementation of the weak reduced product between intervals and congruences in §3.3, implementing such a refinement would require us to do a query for each variable, which is not practical. Second, their notion of queries is limited, as they returns only Booleans. Third, unlike in our system, it seems that this communication system is not used for binary operations, like join, widening or even comparison. Lastly, their system do not provide any means of communication on the initiative of the sender, which our message channels do.

The inspiration of our work mainly comes from the combination of domains in the Astrée static analyzer [5]. This analyzer contains both *input channels* (the equivalent of our query channels) and *output channels* (the equivalent of our message channels). Our main addition is the formalization of this mechanism using the Coq proof assistant. Another difference is their use of the same formalism to describe the implementation of both channels, based on two operators: $\text{EXTRACT}_{D^\#}$ and $\text{REFINE}_{D^\#}$. $\text{EXTRACT}_{D^\#}$ can be seen as an equivalent to `enrich_query_chan` for query channels, and $\text{REFINE}_{D^\#}$ as an equivalent to `process_msg` for message channels. However, we do not have a refining operation using only query channels, nor an operation that would generically extract messages from an abstract environment. For the same reasons as for refinements in open products, we believe these should be integrated in abstract operations, and not part of the interface.

5 Conclusion

The use of this communication system in Verasco has largely contributed to its precision and ease of development. While keeping the code relatively modular and extensible, message channels and query channels provide a flexible and convenient alternative to reduced products. Message channels provide a way to transfer information on the initiative of the sender, and query channels make domains able to query others when needed. Our development is part of Verasco 1.1, whose source code and formalization can be found on the website of Verasco: `http://compcert.inria.fr/verasco/`.

We have described several concrete uses of this system in Verasco, where the channels are used in a simple an flexible way. However, we are convinced this approach can be very easily adapted to other static analyzers in other contexts and for other purposes.

In the Verasco static analyzer, we would like to extend the numerical abstract domain hierarchy in several ways. As an example, a weakly relational abstract domain, such as octagons [11], together with a generic linearization abstract

domain and a variable packing mechanism would necessarily need some kind of communication. This system seems to be a very good candidate for sharing linearized forms of expressions or variable packing policies.

References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI. pp. 196–207. ACM (2003)
2. Cortesi, A., Le Charlier, B., Van Hentenryck, P.: Combinations of abstract domains for logic programming. In: POPL. pp. 227–239. ACM (1994)
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. pp. 269–282. ACM (1979)
5. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the astrée static analyzer. In: ASIAN, LNCS, vol. 4435, pp. 272–300. Springer (2006)
6. Cousot, P., Cousot, R., Mauborgne, L.: The reduced product of abstract domains and the combination of decision procedures. In: FoSSaCS. pp. 456–472. Springer (2015)
7. Granger, P.: Static analysis of arithmetical congruences. International Journal of Computer Mathematics 30(3-4), 165–190 (1989)
8. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: PLDI. pp. 376–386. ACM (2006)
9. Jourdan, J., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: POPL. pp. 247–259. ACM (2015)
10. Leroy, X.: Formal verification of a realistic compiler. Comm. ACM 52(7), 107–115 (2009)
11. Miné, A.: Weakly relational numerical abstract domains. Ph.D. thesis, École Polytechnique (Dec 2004)
12. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. TOPLAS 1(2), 245–257 (1979)