Towards Complete Specifications with an Error Calculus

Quang Loc Le[1], Asankhaya Sharma[1], Florin Craciun[2], and Wei-Ngan Chin[1]

(1) Department of Computer Science, National University of Singapore
(2) Faculty of Mathematics and Computer Science, Babes-Bolyai University, Romania

Abstract. We present an error calculus to support a novel specification mechanism for sound and/or complete safety properties that are to be given by users. With such specifications, our calculus can form a foundation for both proving program safety and/or discovering real bugs. The basis of our calculus is an algebra with *a lattice domain* of four abstract statuses (namely *unreachability*, *validity*, *must-error* and *may-error*) on possible program states and *four operators* for this domain to calculate suitable program status. We show how *proof search* and *error localization* can be supported by our calculus. Our calculus can also be extended to *separation logic* with support for user-defined predicates and lemmas. We have implemented our calculus in an automated verification tool for pointer-based programs. Initial experiments have confirmed that it can achieve the dual objectives, namely of safety proving and bug finding, with modest overheads.

1 Introduction

Traditionally, program specifications are given primarily for safety scenarios and are used to describe the states under which program execution would occur safely. When successfully verified, such specifications are said to be *sound* for their specified input scenarios. That is, a specification is said to be *sound* if it has identified input scenarios (or preconditions) that are guaranteed to lead to safe program execution. However, we are also interested in *complete* specifications that will additionally verify the remaining input scenarios (that lead to execution failure) as invalid ones. Informally, a specification is said to be *complete* if it has unambiguously identified both input scenarios that lead to safe code execution, and input scenarios that lead to code execution failure.

Such complete specifications for programs are helpful for two reasons. Firstly, they can be used to specify precisely (through weakest precondition[1]) when inputs can be handled correctly by programs. Conversely, we are also able to precisely identify when programs would fail to work correctly (or safely). Secondly, the specifications on erroneous inputs can be used to help pinpoint actual software *bugs* in programs as they could be used to precisely indicate where each given error occurs.

Though useful, the task of capturing complete specifications is very challenging, and may not always be possible since the input scenarios under which failures could

[1] While it may be desirable to have weakest precondition that guarantee safety or correctness, we also allow flexibility for users to specify a wider range of specifications that include those with either stronger preconditions and/or weaker postconditions. Though weaker specifications give fewer guarantees, they are more easily verified and may be enough to ensure reliability.

occur may not be unambiguously specified and verified. In this paper, we shall provide the basic mechanisms that can help specify complete specifications, where possible. To achieve this goal, we propose *a lattice domain* of four abstract statuses (namely *unreachability*, *validity*, *must-error* and *may-error*) and make use of the validity (must-error) status for specifying safe (unsafe, resp.) execution scenarios. Furthermore, when the complete requirements are hard (or impossible) to specify, we have also provided approximation mechanisms that can help us specify *near-complete* specifications through the use of *may-error* as opposed to *must-error* classification in weakened postcondition.

Our motivation for developing complete specifications for programs was further heightened by the recent VSTTE competition [1] that was held in November 2011. Out of five problems that the participants were asked to verify for safety and correctness, there were two problems (problem 4 and problem 5) where more complex specifications that satisfy *completeness* were requested. As complete specifications must additionally address erroneous scenarios, we have recently developed a comprehensive verification framework that could just as easily deal with input scenarios that invoke errors, as it would with input scenarios that led to safe program execution

At the heart of our proposal is a calculus that can uniformly specify both safe and unsafe execution scenarios. Our calculus uses *an algebra* with the lattice domain of four-point program statuses and four binary operations over these program statuses. The program statuses can be used for each program state, and also to decorate more precisely the post-conditions of program specification. To support modular verification, we provide our calculus with *two entailment procedures* (one for pre-condition checks at method calls, and another for post-condition checks) and a set of *sound structural rules*. Furthermore, this extension also helps to classify (into must or may) as well as to localize errors when the verification fails. This enables our verifier to work both as a safety and correctness proving tool and as a bug finding tool.

The paper makes the following main contributions

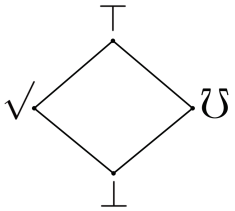
- a lattice domain with four distinct statuses on possible program states.
- a specification mechanism to support both sound and complete properties.
- a calculus (for the lattice) to reason about safety and must/may errors (Sec. 3)
  - support for separation logic with user-defined predicates and lemmas (Sec. 4).
  - support for error calculus within a modular verification framework (Sec. 5).
- an extension to support error localization (Sec. 4.3).

We also demonstrate the calculus capability of proving safety and detecting bugs with modest overheads through an implementation and two experiments in Sec. 6. Next section presents the algebra and new specification mechanism. It also illustrates the use of calculus through examples on modular verification and error localization.

2 Motivation and Overview

2.1 An Algebra on Status of Program States

The basis of our proposal is the identification of an algebra $(\mathcal{E}, \mathcal{F})$ in which $\mathcal{E}$ is a lattice domain with four points used to capture the status of each program state, while $\mathcal{F}$ is a set of four binary operators (meet ($\sqcap$), join ($\sqcup$), compose ($\otimes$) and search ($\oplus$)) to combine the statuses of program states. The four points that are used for program status are as follows:



$\sqcap$	$\top$	$\mho$	$\surd$	$\bot$
$\top$	$\top$	$\mho$	$\surd$	$\bot$
$\mho$	$\mho$	$\mho$	$\bot$	$\bot$
$\surd$	$\surd$	$\bot$	$\surd$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

$\sqcup$	$\top$	$\mho$	$\surd$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$
$\mho$	$\top$	$\mho$	$\top$	$\mho$
$\surd$	$\top$	$\top$	$\surd$	$\surd$
$\bot$	$\top$	$\mho$	$\surd$	$\bot$

$\otimes$	$\top$	$\mho$	$\surd$	$\bot$
$\top$	$\top$	$\mho$	$\top$	$\bot$
$\mho$	$\mho$	$\mho$	$\mho$	$\bot$
$\surd$	$\top$	$\mho$	$\surd$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

$\oplus$	$\top$	$\mho$	$\surd$	$\bot$
$\top$	$\top$	$\top$	$\surd$	$\bot$
$\mho$	$\top$	$\mho$	$\surd$	$\bot$
$\surd$	$\surd$	$\surd$	$\surd$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

(a) (b)

Fig. 1. An Algebra on Status of Program States.

- $\bot$: denotes an unreachable state.
- $\surd$: denotes a valid program state from normal program execution.
- $\mho$: denotes a state that corresponds to a must (or definite) error scenario.
- $\top$: denotes a state that corresponds to a may error or an unknown scenario. That is, it could either be $\bot$, or $\surd$ or $\mho$.

Note that the must error status ($\mho$) subsumes the unreachable $\bot$ status. The may error status ($\top$) comes from imprecision or from dependency on some unknown input. In our system, potential sources of imprecision include imprecise specifications, imprecise invariants of complex data structures and incomplete domains. Although we could separately identify those kinds of imprecision, for simplicity we uniformly specify them with the $\top$ status value. In the implementation, we distinguish them through different messages with status (see Sec. 4.3).

Let $\preceq$ be a partial ordering relation on status whereby $\tau_1 \preceq \tau_2$ means status τ_1 is more precise than status τ_2. The $\sqcup$ and $\sqcap$ operators denote the least upper bound and the greatest lower bound, respectively, over the lattice domain. The domain $\mathcal{E}$ and two operations $\sqcap$, $\sqcup$ form a complete lattice $\mathcal{D} = \langle \mathcal{E}, \preceq, \sqcup, \sqcap, \bot, \top \rangle$ organized as shown in Fig. 1a. This lattice forms a core part of the underlying abstract semantics for our system. Furthermore, $\bot$ is *zero* element of $\otimes$ and $\oplus$ operations; it means $\mathtt{x} \oplus \bot = \bot$ and $\bot \otimes \mathtt{x} = \bot$ for any values $\mathtt{x}$. The remaining calculations of $\otimes$ and $\oplus$ are illustrated in Fig. 1b. The $\otimes$ operator is meant to support conjunctive proving, and searches for *failures* from $\mho$ and $\top$ status . The $\oplus$ operator is meant to support *proof search*, and searches for $\surd$ status to succeed in proving. Thus the priority order of the $\otimes$ operator is $\mho$, $\top$ and lastly $\surd$, and the priority order of the $\oplus$ operator is $\surd$, $\top$ and lastly $\mho$. Contrast this with the $\sqcup$ operator which doesn't have any priority between $\surd$ and $\mho$. So it would simply yield $\top$ when the two statuses are combined together.

2.2 Mechanism for Sound and Complete Specifications

To illustrate our new specification mechanism, we consider a method that returns the data which its input points to, as shown below

```
int get_data(node x)
 case{ x≠null → requires x↦node⟨d, p⟩ ensures (res=d) √;
       x=null → ensures (true) ℧; }
```

where `res` is a reserved identifier denoting the method's result and the data structure `node` is declared as: `data node { int val; node next }`.

In our system, each method is specified by pre- and post-conditions (through separation logic formulas), denoted by `requires` and `ensures` keyword, respectively. In the

specification above, we also use structured specifications [9] where disjoint conditions are expressed using case construct for expressing both sound (with x≠null condition) and complete (with x=null condition) requirements, as can be seen for the above specification of get_data (with the $\surd$ and $\mho$ statuses in postconditions, resp.). In comparison, if we are only interested in sound specification, we could just use the following instead:

```
int get_data(node x)
 requires x↦node⟨d, p⟩ ensures (res=d) √;
```

Occasionally, it may be possible to automatically generate complete specification by negating the input conditions of sound specification. However, this may not always be feasible. Firstly, negation computation may be hard to implement in complex domains. For example, it is unclear how to compute negation in separation logic (which our system relies on). Secondly, not all methods have clearly delineated boundary between sound and complete conditions, as an example consider the interactive schedule (ischedule) method in Fig. 2. With prio=0 condition, this method's status depends on the user input which is unknown at verification time. Therefore, there exists a gap between soundness and completeness that cannot be derived simply through the negation operation. For this example, we can instead provide a *near-complete* specification, as shown in the bottom right of Fig. 2. Informally, a specification is said to be *near-complete* if it captures all possible input conditions but contains either $\top$ program status or an ambiguous disjunction, comprising of both $\surd$ and $\mho$ statuses, in one or more of its postconditions.

```
1. int ischedule(int prio){
2.   if (prio>0)/*run it */ return 0;
3.   else if (prio<0) abort();
4.   else{
5.     printf("Allow this task to run? y or n");
6.     char c=getc();
7.     if (c =' y')/*run it */ return 0;
8.     else abort(); } }
```

Sound Specification:

```
l1. int ischedule(int prio)
l2.   requires prio>0 ensures (res=0)√;
```

Near-Complete Specification:

```
l3. int ischedule(int prio)
l4. case { prio>0 → ensures (res=0)√;
l5.        prio<0 → ensures (true )℧;
l6.        prio=0 → ensures (true )⊤; }
```

Fig. 2. Code and Specification of *ischedule* Method.

We note that our approach for proving the completeness of program is based on the assumption that the user-supplied specification is complete; namely that it covers all values of the input domain and that each error program state denotes an input scenario where no valid output state is possible. Checking (or even inferring) the completeness of specifications is a challenging research direction that could be investigated in future.

2.3 Essence of Error Calculus

To highlight how our calculus can be used to verify programs, consider the method foo in Fig. 3. We shall verify the code of foo in a forward manner, and would compute a program state for each of its program point. Each program state, Φ, is a formula on the state of variables and heap. Each program state can be combined with a status and is represented by (Φ, τ) where τ denotes a status value from our lattice.

```
1 int foo(int x, int y)
2  requires x≥0
3  ensures (res>0)√; {
4   if (x<0) return −1; /*L1*/
5   else{
6    if (y>1) return 1; /*L2*/
7    else if (y<0) return −1; /*L3*/
8     else return y; /*L4*/
9  }}
```

Fig. 3. Code of *foo* Method.

As part of compositional verification, the precondition of each callee is checked against the current calling context and the postcondition is checked at the exit of the method's body. In the example, we can identify four program states of interests that correspond to four exits (L1, L2, L3 and L4) of the method. The following illustrates how the statuses are decided at exits through proof obligations discharged for postcondition checking with the help of the entailment procedure $\vdash_C$ that conforms to our error calculus. Given a program state π_a and a *post-condition* π_c, we can determine the status s for such checking with the help of the following judgment: $\pi_a \vdash_C \pi_c \rightsquigarrow s$. The resulting statuses generated by the entailment procedure are as follows:

$$\begin{array}{ll}
\texttt{L1} : x{\geq}0 \wedge x{<}0 \wedge \texttt{res}{=}{-1} & \vdash_C \texttt{res}{>}0 \rightsquigarrow \bot \\
\texttt{L2} : x{\geq}0 \wedge \neg(x{<}0) \wedge y{>}1 \wedge \texttt{res}{=}1 & \vdash_C \texttt{res}{>}0 \rightsquigarrow \surd \\
\texttt{L3} : x{\geq}0 \wedge \neg(x{<}0) \wedge \neg(y{>}1) \wedge y{<}0 \wedge \texttt{res}{=}{-1} & \vdash_C \texttt{res}{>}0 \rightsquigarrow \mho \\
\texttt{L4} : x{\geq}0 \wedge \neg(x{<}0) \wedge \neg(y{>}1) \wedge \neg(y{<}0) \wedge \texttt{res}{=}y & \vdash_C \texttt{res}{>}0 \rightsquigarrow \top
\end{array}$$

Each of the above proofs yields a status based on the outcome of its entailment. This status can be added to program state for each of these program points. At L1, the antecedent is unsatisfiable which corresponds to an unreachable scenario (either infinite loop[2] or dead code) that can be captured by $(\texttt{false}, \bot)$ with `false` denoting contradiction at that program point. At L2, the consequent can be directly proven using the antecedent. This yields a valid program state that can be represented by $(x{\geq}0{\wedge}\neg(x{<}0){\wedge}y{>}1{\wedge}\texttt{res}{=}1, \surd)$. This program state indicates that the method will exit safely at this location with $\texttt{res}{=}1$. At L3, the negation of the consequent can be proven from its antecedent. The program state at L3 can be computed to be a must error as $x{\geq}0{\wedge}\neg(x{<}0){\wedge}y{<}0, \mho)$. The sub-formula on result $\texttt{res}{=}{-1}$ is dropped since we have a must error outcome where the output state is unimportant. At L4, the antecedent can neither prove the consequent nor its negation. Hence, we would need to classify this program point as a may error whose state is $(x{\geq}0{\wedge}\neg(x{<}0) \wedge \neg(y{>}1) \wedge \neg(y{<}0) \wedge \texttt{res}{=}y, \top)$. A formula on result $\texttt{res}{=}\texttt{y}$ is still captured since the $\top$ status includes possibly safe output.

When an entailment checking fails, an error messages is generated with relevant information to help debugging process. For example, the error message at $\texttt{L}_3$ is:

> Verify method `foo`. Proving postcondition fails:
> Failure (must):
> $(x{\geq}0, 2) \wedge (\neg(x{<}0), 5) \wedge (\neg(y{>}1), 6) \wedge (y{<}0, 7) \wedge (\texttt{res}{=}{-1}, 7) \vdash_C (\texttt{res}{>}0, 3)$

where irrelevant formulas are sliced away and failures are localized by pairs of the relevant failing formulas and their corresponding statement code or specification line numbers.

[2] Although we provide a mechanism to specify infinite loop, proving termination is beyond the scope of this paper.

3 Assertion Language

In this section, we introduce the concepts and terminology that are used to describe our calculus throughout the paper. Our formalism includes inductive predicates in separation logic which are written in an assertion language. We extend this language with program status (τ) to support error calculus with different program states.

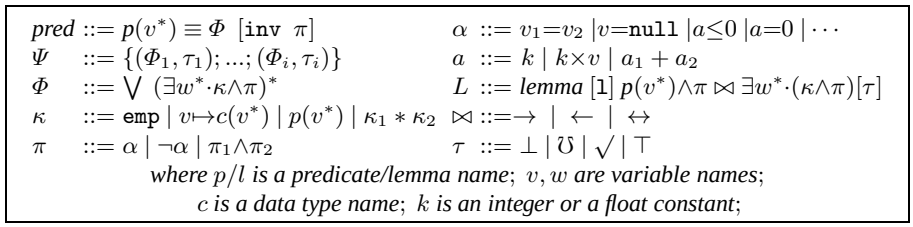


Fig. 4. The Assertion Language

Separation logic can provide concise and precise notations for specifying pointer-based programs and their data structures. We enhance the separation logic fragment presented in [2, 18]. Figure 4 describes our assertion language. Each data structure and its properties can be defined by an inductive predicate *pred*, that consists of a name *p*, a main separation formula Φ and an optional pure invariant formula π that must hold for every predicate instance. The separation logic formula Φ is a disjunction of symbolic heap. Each symbolic heap is a conjunction of a heap formula κ and a pure formula π. The pure part captures a rich constraint from the domains of Presburger arithmetic, monadic set or polynomial real arithmetic. The heap part includes points-to predicate $\mapsto$, spatial conjunction predicate $*$ for combining two disjoint heap memory, and user-defined predicates $\mathtt{p}\langle \mathtt{v}_1, .., \mathtt{v}_n \rangle$ to capture more complex data structures with selected properties. For examples, with the simple data structure $node$ declared in Sec. 2.2, we define variants of list segment, as follows:

$$\begin{array}{l}
\mathtt{pred\ lseg}\langle \mathtt{root}, \mathtt{n}, \mathtt{p}\rangle \equiv (\mathtt{root}{=}\mathtt{p} \wedge \mathtt{n}{=}0) \\
\quad \vee \exists\, \mathtt{d}, \mathtt{q} \cdot (\mathtt{root}{\mapsto}\mathtt{node}\langle \mathtt{d}, \mathtt{q}\rangle * \mathtt{lseg}\langle \mathtt{q}, \mathtt{n}{-}1, \mathtt{p}\rangle)\ \ \mathbf{inv}\ \mathtt{n}{\geq}0 \\
\mathtt{pred\ plseg}\langle \mathtt{root}, \mathtt{n}, \mathtt{p}\rangle \equiv \exists\, \mathtt{d} \cdot (\mathtt{root}{\mapsto}\mathtt{node}\langle \mathtt{d}, \mathtt{p}\rangle \wedge \mathtt{n}{=}1 \wedge \mathtt{d}{\geq}0) \\
\quad \vee \exists\, \mathtt{d}, \mathtt{q} \cdot (\mathtt{root}{\mapsto}\mathtt{node}\langle \mathtt{d}, \mathtt{q}\rangle * \mathtt{plseg}\langle \mathtt{q}, \mathtt{n}{-}1, \mathtt{p}\rangle \wedge \mathtt{d}{\geq}0)\ \ \mathbf{inv}\ \mathtt{n}{\geq}1
\end{array}$$

The predicate `lseg` describes a list segment of nodes whose length is captured by the parameter `n`. Similarly, the predicate `plseg` describes a list segment with only non-negative integers.

Lemmas are used to relate data structures beyond their original predicate definitions [17]. A lemma specification consists of a head $p(v^*)$, a guard π, a body Φ and a direction to apply (left $\rightarrow$, right $\leftarrow$ or both $\leftrightarrow$) that denotes a weakening, strengthening or equivalence, respectively. For example, to illustrate that $\mathtt{plseg}\langle \mathtt{root}, \mathtt{n}, \mathtt{p}\rangle$ is an instance of $\mathtt{lseg}\langle \mathtt{root}, \mathtt{n}, \mathtt{p}\rangle$, we can use the following left (or weakening) coercion lemma:

$$\mathtt{lemma\ w}_1\ \mathtt{plseg}\langle \mathtt{root}, \mathtt{n}, \mathtt{p}\rangle \wedge \mathtt{n}{>}0 \rightarrow \mathtt{lseg}\langle \mathtt{root}, \mathtt{n}, \mathtt{p}\rangle$$

4 A Calculus on Errors

In this section, we initially formalize the calculus with pure (without heap) formulas π. The extension of the calculus to heap formulas will be presented in the next section.

4.1 The Entailment Procedures

In this subsection, we introduce two entailment procedures for discharging the proof obligations with support for the four-points status.

Entailment Procedure for Postconditions Checking. The basic machinery for the judgment $\pi_a \vdash_C \pi_c \rightsquigarrow s$ is captured by the following four rules. We use underlying theorem solvers for answering sastifiability. Note that $\textbf{UNSAT}(\pi)$ denotes that π is definitely unsatisfiable and $\textbf{PSAT}(\pi)$ denotes that π is possibly satisfiable (as a complement of unsatisfiability checking and due to its incompleteness).

$$\frac{[\textbf{EC}\text{-}[\textbf{BOTTOM}]]\quad \textbf{UNSAT}(\pi_1)}{\pi_1 \vdash_C \pi_2 \rightsquigarrow \bot} \qquad \frac{[\textbf{EC}\text{-}[\textbf{OK}]]\quad \textbf{PSAT}(\pi_1)\ \ \textbf{UNSAT}(\pi_1 \wedge \neg\pi_2)}{\pi_1 \vdash_C \pi_2 \rightsquigarrow \surd}$$

$$\frac{[\textbf{EC}\text{-}[\textbf{MUST}\text{-}\textbf{ERROR}]]\quad \textbf{PSAT}(\pi_1)\ \ \textbf{UNSAT}(\pi_1 \wedge \pi_2)}{\pi_1 \vdash_C \pi_2 \rightsquigarrow \mho} \qquad \frac{[\textbf{EC}\text{-}[\textbf{MAY}\text{-}\textbf{ERROR}]]\quad \textbf{PSAT}(\pi_1 \wedge \neg\pi_2)\ \ \textbf{PSAT}(\pi_1 \wedge \pi_2)}{\pi_1 \vdash_C \pi_2 \rightsquigarrow \top}$$

Two rules at the first line check the success of the entailment and classify it as unreachable ($\bot$) or valid ($\surd$) as usual (checking $\textbf{UNSAT}(\pi_1 \wedge \neg\pi_2)$ is equivalent to checking $\pi_1 \Longrightarrow \pi_2$). Next two rules at the second line check and classify the must/may error scenarios; in the first rule, a must error is identified when $\pi_a \Longrightarrow \neg\pi_c$ is provable: lastly, due to the imprecision, entailments which has not been proven so far are marked with unknown status through the second rule. (In the last rule, the condition $\textbf{PSAT}(\pi_1)$ is discarded because it can be implied from two present conditions.)

To illustrate this entailment procedure, let us consider a postcondition check, $x{\geq}0$, under four different program states, as shown below.

$$\begin{array}{llll} x{\leq}{-}1 \wedge x{=}0 & \vdash_C x{\geq}0 \rightsquigarrow \bot & x{\leq}{-}1 & \vdash_C x{\geq}0 \rightsquigarrow \mho \\ x{>}0 & \vdash_C x{\geq}0 \rightsquigarrow \surd & \texttt{true} & \vdash_C x{\geq}0 \rightsquigarrow \top \end{array}$$

Entailment Procedure for Preconditions Checking. Furthermore, to support the checking of preconditions from specifications with soundness and/or completeness, we introduce another entailment judgment of the form: $\pi_a \vdash_E \pi_c \rightsquigarrow s$.

$$\frac{[\textbf{EE}\text{-}[\textbf{BOTTOM}]]\quad \textbf{UNSAT}(\pi_1)}{\pi_1 \vdash_E \pi_2 \rightsquigarrow \bot} \qquad \frac{[\textbf{EE}\text{-}[\textbf{OK}]]\quad \textbf{PSAT}(\pi_1)\ \ \textbf{UNSAT}(\pi_1 \wedge \neg\pi_2)}{\pi_1 \vdash_E \pi_2 \rightsquigarrow \surd} \qquad \frac{[\textbf{EE}\text{-}[\textbf{MAY}]]\quad \textbf{PSAT}(\pi_1 \wedge \neg\pi_2)}{\pi_1 \vdash_E \pi_2 \rightsquigarrow \top}$$

The status for this entailment is now limited to only three possible values, namely $\bot$, $\surd$ and $\top$, without the $\mho$ status, as illustrated below:

$$\begin{array}{llll} x{\leq}{-}1 \wedge x{=}0 & \vdash_E x{\geq}0 \rightsquigarrow \bot & x{\leq}{-}1 & \vdash_E x{\geq}0 \rightsquigarrow \top \\ x{>}0 & \vdash_E x{\geq}0 \rightsquigarrow \surd & \texttt{true} & \vdash_E x{\geq}0 \rightsquigarrow \top \end{array}$$

Unlike the earlier entailment procedure, this new entailment has introduced a $\top$ status value where $\mho$ was derived previously, since the precondition may be under-approximated. We can recover from this lack of information by leveraging on the status from postconditions, where applicable. We defer formalization of the recovery to Sec. 6, we now illustrate it through the check of the calling context, `prio`$<$`0`, against the near-complete specification of the `ischedule` procedure (presented in Fig. 2) as follows:

$$\begin{array}{l}\texttt{prio<0} \vdash \texttt{case}\,\{\,\texttt{prio>0} \rightarrow \texttt{ensures}\,(\texttt{res=0})\surd;\\ \qquad\qquad\qquad\quad \texttt{prio<0} \rightarrow \texttt{ensures}\,(\texttt{true})\mho;\\ \qquad\qquad\qquad\quad \texttt{prio=0} \rightarrow \texttt{ensures}\,(\texttt{true})\top;\,\}\\ \qquad\qquad \rightsquigarrow\ (\bot\otimes\surd)\sqcup(\surd\otimes\mho)\sqcup(\bot\otimes\top)\ \rightsquigarrow\ \bot\sqcup\mho\sqcup\bot\\ \qquad\qquad \rightsquigarrow\ \mho\end{array}$$

This compositional check is performed through two steps. Firstly, for each scenario (1) the calling context is combined with the condition of current scenario; (2) unsatisfiability check is performed by the $\vdash_E$ procedure; and (3) the status from postcondition is combined (by $\otimes$). Secondly, those scenarios are joined (by $\sqcup$).

4.2 Structural Rules

We provide sound structural rules that would carry out the entailment proving process in smaller entailments. These rules support error localization, separation entailment procedure and modular verification.

$$\frac{\pi_1 \vdash \pi \rightsquigarrow \tau_1 \quad \pi_2 \vdash \pi \rightsquigarrow \tau_2}{\pi_1 \vee \pi_2 \vdash \pi \rightsquigarrow \tau_1 \sqcup \tau_2}\ [\underline{\mathbf{SE}-[\sqcup\ \mathbf{JOIN}]}] \qquad \frac{\pi \vdash \pi_1 \rightsquigarrow \tau_1 \quad \pi \vdash \pi_2 \rightsquigarrow \tau_2}{\pi \vdash \pi_1 \wedge \pi_2 \rightsquigarrow \tau_1 \otimes \tau_2}\ [\underline{\mathbf{SE}-[\otimes\ \mathbf{COMPOSE}]}] \qquad \frac{\pi \vdash \pi_1 \rightsquigarrow \tau_1 \quad \pi \vdash \pi_2 \rightsquigarrow \tau_2}{\pi \vdash \pi_1 \vee \pi_2 \rightsquigarrow \tau_1 \oplus \tau_2}\ [\underline{\mathbf{SE}-[\oplus\ \mathbf{SEARCH}]}]$$

These rules use the algebraic operations presented in Sec. 2.1 to combine the results. Note that, $\vdash$ is generic, and can be $\vdash_C$ or $\vdash_E$. The first rule decomposes disjunction on the antecedent, while the second rule decomposes conjunction on the consequent. Both these rules can be implemented without any loss of information. The third rule performs a search over a disjunction in the consequent. This search returns a set of possible proofs for the entailment. According to the $\oplus$ operator, if at least one $\surd$ status is found in this solution set, the entailment will succeed.

Theorem 1 (Soundness of the Structural Rules). *Given an entailment $\pi_1 \vdash \pi_2$. ($\vdash$ is either $\vdash_C$ or $\vdash_E$). If the application of the structural rules $[\underline{\mathbf{SE}-[...]}]$ on the given antecedent π_1 and consequent π_2 returns the result τ, then the application of the $[\underline{\mathbf{EC}-[...]}]$ ($[\underline{\mathbf{EE}-[...]}]$) rules on the given antecedent π_1 and consequent π_2 returns the same result τ, namely $\pi_1 \vdash_C \pi_2 \rightsquigarrow \tau$ ($\pi_1 \vdash_E \pi_2 \rightsquigarrow \tau$, respectively).*

The proof is by an induction on the structural rules $[\underline{\mathbf{SE}-[...]}]$ and a case analysis on the returned result τ. We present full proof of the theorem in the Appendix A. (proof of $\vdash_C$) and Appendix B (proof for $\vdash_E$).

4.3 Error Localization Extension to Calculus

$$\begin{array}{lll}\tau[m] & ::= \bot[\emptyset] \mid \mho[m] \mid \surd[m] \mid \top[m] & \tau_1[m_1] \diamond \tau_2[m_2] \Rightarrow (\tau_1 \diamond \tau_2)[m_1 \diamond m_2]\\ m & ::= bm \mid m_1 \sqcup m_2 \mid m_1 \otimes m_2 \mid m_1 \oplus m_2 & m \diamond \emptyset \Rightarrow m\\ bm & ::= \pi_1 \Longrightarrow \pi_2\ (\texttt{valid}) & \emptyset \diamond m \Rightarrow m\\ & \mid \pi_1 \Longrightarrow \overline{\pi_2}\ (\texttt{must error}) & \bot[m] \Rightarrow \bot[\emptyset]\\ & \mid \pi_1 \Longrightarrow \underline{\pi_2}\ (\texttt{may error}) & \end{array}$$

Fig. 5. Program State: Status and Message

To provide support for error localization, we must extend the four-point lattice with messages that capture the reason for each success or failure (see the left of Fig. 5).

Status $\bot$ does not carry any message which is denoted by $\emptyset$. When faced with a message with error from $m_1 \sqcup m_2$ and $m_1 \otimes m_2$, both of the two smaller messages (with possible errors), denoted by m_1 and m_2, must be resolved, before the main message is said to be resolved. When faced with a message with error of the form $m_1 \oplus m_2$, only one of the messages with errors from either m_1 or m_2 needs to be resolved, before the main message $m_1 \oplus m_2$ is resolved. We may now modify the three operators $\sqcup$, $\otimes$ and $\oplus$, to propagate messages capturing the localizations for successes and failures. Let us denote this using a generic name $\diamond$ for three operators. We propagate every message, where possible, as shown at the right of Fig. 5. In case empty message $\emptyset$ is generated, we remove it from the main message as shown in the second and third rules. In case the resulting status from $\tau_1 \diamond \tau_2$ is $\bot$, we remove its messages, as shown in the last rule.

5 Error Calculus for Separation Logic

In this section, we show how our calculus can be used to support the reasoning of pointer-based programs via the fragment of separation logic presented in Sec.3. As separation logic is a sub-structural logic, we have to account for heap memory as a resource. Thus, entailment in separation logic is typically supported with a frame inference capability [2, 18], similar to the following format:

$$\Phi_1 \vdash \Phi_2 * \Phi_3$$

whereby antecedent Φ_1 entails Φ_2 with a residue frame captured by Φ_3. Logically, the above entailment is equivalent to $\Phi_1 \Longrightarrow \Phi_2 * \Phi_3$ where Φ_3 may contain existential variables that have been instantiated and pure formula that were already established in Φ_1.

We enhance the entailment procedure for separation logic in two steps. First, we extend the entailment procedure above to support the error calculus by the following judgment:

$$\Phi_1 \vdash \Phi_2 \rightsquigarrow (\Phi_3, \tau)$$

If the antecedent semantically entails the consequent, the entailment succeeds and we expect status τ to be set to $\surd$. Otherwise, the entailment fails and we expect τ to be set to either $\mho$ or $\top$. Second, this procedure is extended to support proof search with disjunctive formulas and lemma as elaborated in Sec. 5.1.

To illustrate the first step, let us examine four simple examples to better understand how status outcome is being determined by the entailment procedure of separation logic.

Entailment 1

$$\begin{array}{l} x{\mapsto}\mathtt{node}(\_, q) * q{\mapsto}\mathtt{node}(\_, \mathtt{null}) \\ \quad \vdash_C x{\mapsto}\mathtt{node}(\_, p) \\ \rightsquigarrow (q{\mapsto}\mathtt{node}(\_, \mathtt{null}) \wedge p{=}q \wedge x{\neq}\mathtt{null}, \surd) \end{array}$$

Entailment 2

$$\begin{array}{l} x{\mapsto}\mathtt{node}(\_, q) * q{\mapsto}\mathtt{node}(\_, \mathtt{null}) \\ \quad \vdash_C x{\mapsto}\mathtt{node}(\_, \mathtt{null}) \\ \rightsquigarrow (q{\mapsto}\mathtt{node}(\_, \mathtt{null}), \mho) \end{array}$$

The entailment 1 yields a residue $q{\mapsto}\mathtt{node}(\_, \mathtt{null})$ and an instantiation $p{=}q$ from (implicit) existential variable p. It also carries a pure formula $x{\neq}\mathtt{null}$ from the antecedent. The entailment 2 yields a must failure, denoted by $\mho$. The consequent expects $q{=}\mathtt{null}$, but the antecedent had $q{\mapsto}\mathtt{node}(\_, \mathtt{null})$. This contradiction has caused a $\mho$ failure to be raised. The residue captures the state when failure was detected.

Entailment 3

$$\begin{array}{c} x{\mapsto}\mathtt{node}(\_, q) * q{\mapsto}\mathtt{node}(\_, \mathtt{null}) \vdash_C x{\mapsto}\mathtt{node}(3, p) \\ \rightsquigarrow (q{\mapsto}\mathtt{node}(\_, \mathtt{null}) \wedge p{=}q \wedge x{\neq}\mathtt{null}, \top) \end{array}$$

The entailment 3 yields a may failure, denoted by $\top$. The consequent expects value 3 to be proven as the data field of x. However, the antecedent has no information on that field position. Hence, a $\top$ failure was raised.

5.1 Separation Entailment with Proof Search

To support proof search the entailment procedure for separation logic shall now be presented as a judgment of the following (full) form:

$$\Phi_1 \vdash \Phi_2 \rightsquigarrow (\Psi, \tau)$$

whereby Ψ captures a set of residual program states with status information. We use a set of program states (Ψ) since our entailment procedure may have to conduct a proof search with the help of lemmas. Furthermore, we must extend our entailment procedure in the following ways. First, rules are added to support proof search that adds to our set of outcomes with the help of lemmas. Proof search is performed in the order as follows:

- Status values of the proof search with lemmas are combined by the union ($\oplus$) operator (where $\surd$ or $\top$ take priority over $\mho$). Hence, if a proof search attempt fails, we return a $\top$ (unknown) status, rather than a $\mho$ status since the latter prevents a $\surd$ success from being reported, even if they can be confirmed in a different proof search.
- If a complete set of lemmas have already been explored, then a must error status is returned.

Second, when our entailment procedure becomes stuck with a non-empty consequent, comprising some heap predicates, we shall firstly determine a pure approximation of the consequent for both heap and pure data through `XPure` procedure [2]. For examples:

$$\begin{array}{ll} \texttt{XPure}(\texttt{x}{\mapsto}\texttt{node}\langle \_, \_\rangle) & \Longrightarrow \texttt{x}{\neq}\texttt{null} \\ \texttt{XPure}(\texttt{lseg}\langle \texttt{root}, \texttt{n}, \texttt{p}\rangle) & \Longrightarrow \texttt{root}{=}\texttt{p} \wedge \texttt{n} = 0 \vee \texttt{root}{\neq}\texttt{null} \wedge \texttt{n}{>}0 \end{array}$$

where $\Longrightarrow$ denotes our over-approximation, and lseg is a predicate defined in Sec 3. We may then determine if there is any contradiction with the antecedent to decide whether must or may failure is going to be reported.

6 Modular Verification with Error Calculus

Code verification is typically formalised using Hoare triples of the form $\{pre\}c\{post\}$, where *pre*, *post* are the initial and final states of program code *c*. To incorporate status into our program state, we shall use disjunctive program state of form $\bigvee(\Phi, \tau)$, giving us a new Hoare triple of the form $\{\bigvee(\Phi_1, \tau_1)\}\ c\ \{\bigvee(\Phi_1, \tau_1)\}$. To simplify our presentation, we shall use (Φ, τ) instead of the more general disjunctive program state $\bigvee(\Phi, \tau)$ that was implemented. To provide sound and complete requirements, we shall also use structured specification from [9] of the form below:

$$\mathsf{Y} ::= \texttt{requires}\ \Phi\ \mathsf{Y} \mid \texttt{case}\{\pi_1 \Rightarrow \mathsf{Y}_1; \ldots; \pi_n \Rightarrow \mathsf{Y}_n\} \mid \texttt{ensures}\ (\Phi)\tau$$

This extends the pre/post specifications to support case analysis and staged verification. The verification requirement for methods can be affected by progressively collecting the precondition in the structured specification, prior to the verification of its method body. As this process is straightforward, we omit the details here.

The abstract semantics of each method call is captured by its specifications. We encode its verification with the rule $[\underline{\mathbf{FV}-[\mathbf{CALL}]}]$. Note that $(\texttt{t v})^*$ and $(\texttt{ref t u})^*$ denote *pass-by-value* and *pass-by-reference* parameters, respectively. Each method call $\texttt{mn}(\texttt{v}^*, \texttt{u}^*)$ in our core language has only variables as arguments. To avoid the need for argument substitutions, we assume that each method declaration from `Program` has been suitably renamed so that actual arguments are identical to the formal arguments.

$$\frac{\begin{array}{c}[\underline{\mathbf{FV}-[\mathbf{CALL}]}] \\ t_0\ mn\ ((t\ v)^*, (ref\ t\ u)^*)\ \mathsf{Y}\ \{c\} \in \texttt{Program} \\ \Phi_1 \vdash \mathsf{Y} \rightsquigarrow (\Phi_2, \tau_2) \\ \Phi_R = \texttt{if}\ \tau_1{=}\surd\ \texttt{then}\ (\exists v'^* \cdot \Phi_2)\ \texttt{else}\ \Phi_1\end{array}}{\{(\Phi_1, \tau_1)\}\ mn\ ((t\ v)^*, (ref\ t\ u)^*)\{(\Phi_R, \tau_1 \otimes \tau_2)\}}$$

The proof obligations are generated and verified at the second line, provided that the incoming status τ_1 is $\surd$. Furthermore, output states from proving entailment are composed with status from pre-state at the third line. By default, if the caller context contains errors, such errors are simply propagated to the next instruction in a similar manner as exceptions. However, unlike exceptions, error states are never caught. To generate proof obligations for the extended specification, we propose to extend the entailment procedure to handle specification with separation formulas. The revised judgment has the form $\Phi_1 \vdash \mathsf{Y} \rightsquigarrow (\Phi_2, \tau_2)$, where Φ_1 is the current state, Y is the specification and (Φ_2, τ_2) is the residual state and its status. Three syntax-directed rules are extended. They are used to prove each precondition and assume its respective postcondition for the callee, as shown below:

$$\frac{[\underline{\mathbf{FV}-[\mathbf{C}-\mathbf{REQUIRES}]}]}{\Phi_1 \vdash_E \Phi \rightsquigarrow (\Phi_2, \tau_2) \qquad (\Phi_2) \vdash \mathsf{Y} \rightsquigarrow (\Phi_3, \tau_3)}{\Phi_1 \vdash \texttt{requires}\ \Phi\ \mathsf{Y} \rightsquigarrow (\Phi_3, \tau_2 \otimes \tau_3)}$$

$$\frac{[\underline{\mathbf{FV}-[\mathbf{C}-\mathbf{CASE}]}]}{\Phi \wedge \pi_i \vdash \mathsf{Y}_i \rightsquigarrow (\Phi_i, \tau_i)\ i = 1 \ldots n}{\Phi \vdash \texttt{case}\{\pi_i \Rightarrow \mathsf{Y}_i\}^* \rightsquigarrow (\bigvee \Phi_i, \sqcup \tau_i)} \qquad \frac{[\underline{\mathbf{FV}-[\mathbf{C}-\mathbf{ENSURES}]}]}{\Phi_1 \vdash_C \texttt{true} \rightsquigarrow (\Phi, \tau_1)}{\Phi_1 \vdash \texttt{ensures}\ (\Phi_2)\tau_2 \rightsquigarrow (\Phi_1 * \Phi_2, \tau_1 \otimes \tau_2)}$$

7 Implementation and Experiments

We have implemented our error calculus inside a program verification system for separation logic, called HIPEE. We use HIPEE to verify C-based programs against user-given specifications. The verification is performed compositionally for each method, and loops are transformed to recursive methods. HIPEE eventually translates separation logic proof obligations to pure formulae that can be discharged by different theorem provers. Our system uses Omega [20], MONA [15], Redlog [7] and Z3 [4] as underlying theorem provers for answering the satisfiability and simplification queries. When program code is not successfully verified against safety properties, HIPEE not only further classifies the failures into the must or may errors but also localizes program statements and specifications relevant to the errors.

7.1 Calculus Performance for Heap-Based Programs

To evaluate the overheads of error calculus, we executed our system HIPEE twice, once *with* error calculus and a second time *without*, on a suite of bug-free pointer-based programs. We stress that although the sizes of these programs are fairly small, they deal

Programs (specified props)	Size		Proc	Time(sec.)		Invo.(#)	
	LOC	LOS	#	wo	w	wo	w
Linked list (size,interval)	327	50	26	0.44	0.46	2738	3202
Linked list (size,sets)	157	27	13	0.58	0.6	1520	1724
Sorted llist (size,sness,sets)	98	11	6	0.46	0.49	955	1060
Doubly llist (size,interval)	186	23	13	0.34	0.34	1864	2083
Doubly llist (size,sets)	91	13	5	0.5	0.5	1309	1429
CompleteT (size,minheight)	106	12	5	0.87	0.94	2149	2533
Heap trees (size,maxelem)	179	13	5	1.9	1.91	4540	4954
AVL (height, size)	313	27	12	3.44	3.59	7863	8585
AVL2 (height,size,bal)	152	37	7	2.83	3	6959	7876
BST (size,height)	177	18	9	0.35	0.37	1883	2192
BST (size,height,interval)	153	12	6	0.3	0.31	1581	1836
RBT (size,blackheight)	508	48	19	3.32	3.38	13069	16687
Bubble sort (size)	75	9	4	0.21	0.21	1092	1254
Quick sort (size, sets)	82	10	4	0.27	0.28	778	832
Merge sort (size,sets)	109	11	6	0.47	0.5	1035	1074
Quick sort - queue (size)	127	4	2	4.25	5.27	13218	21139
Total	2840	325	142	20.53	22.15	62553	78460

Table 1. Verification Performance with (*w*) and without (*wo*) Error Calculus

with fairly complex heap-based data structures, such as linked lists, doubly-linked lists and AVL-trees. Therefore, these programs can be used to *fully* evaluate the performance of our calculus which has been embedded inside a separation logic prover. The results are summarized in Table 1. The first column contains the list of the verified programs and their proven properties while the second, third and fourth columns describe number of lines of code (LOC), number of lines of specification (LOS) and number of procedures in each program. On average, LOS is around 12% of LOC and specifications are complicated enough to demonstrate the performance of our calculus. The fifth and sixth columns show the total verification time (in seconds) for the system HIPEE without and with error calculus, respectively. The last two columns capture the number of satisfiability and simplification queries sent to the provers for each experiment.

In Table 1, the results show that the total overhead introduced by our error calculus is around 1.62 seconds (8%). This overhead is proportional to the number of extra satisfiability and simplification queries shown in the last two columns. These experimental results have shown that must/may error calculus with messages can be supported with modest overhead.

7.2 Calculus Usability

In order to show the usability of our error calculus on bugs finding and localizing, we evaluated our system on the Siemens test suite [12] of programs. The test suite contains programs with complex data structures (e.g. linked lists, queues), arrays and loops. Each program in the suite has one non-faulty version, v_0, and a number of seeded faulty versions (#Ver. column in Table 2) from v_1 to v_n. Each of these faulty versions has one or more (seeded) faults. Total number of faults is captured in #Fault column. These faulty versions are suitable for checking the ability of tools in finding bugs and localizing errors (as used in [14]).We provide specifications for each program such that HIPEE

Programs	LOC	LOS	#Proc.	#Ver.	#Fault	$\mho$	$\top^*$	$\top$	LOE	time(s)
tcas	173	48	9	41	48	31	14	3	3.48	3.06
schedule2	374	108	16	10	10	5	0	3	3	8.25
schedule1a	412	50	18	10	16	15	0	1	4.38	18.13
schedule1b	413	50	18	9	8	7	0	1	4.25	32.29
replace	564	73	21	24	24	18	0	6	4.21	17.89
print_tokens2	570	64	19	10	10	7	0	1	4.88	20.42
print_tokens	726	87	18	7	9	8	0	1	3.67	6.73
Total/(Average)	3232	480	119	111	125	91	14	16	(3.98)	(15.25)

Table 2. Bugs finding and localizing with small programs in the Siemens Test Suite

(1) successfully verifies safety (sound or complete requirements) in the non-faulty versions, and (2) captures potential must-bug errors that are complementary to the safety scenarios. We emphasize that these specifications were designed primarily to verify safety scenarios *without* considering the faulty versions of each program. Nevertheless, HIPEE is able to utilize the same specification to find and explain the presence of bugs in the faulty versions, as elaborated below.

Table 2 shows the result of running our system on six programs from the suite. The properties our tool proved include: (i) memory safety (all), (ii) size of data structures (`schedule1a`, `schedule1b` and `schedule2` program), (iii) array-related properties (`tcas`, `print_token`, `print_token2` and `replace` program), (iv) functional arithmetic constraints between input and output (all). We are interested in finding out all the errors in the programs and classifying them as must ($\mho$), disjunctive may ($\top^*$) or may ($\top$) errors. For instance, from 48 faults of program `tcas`, HIPEE was able to detect all the errors in the program, and classified 31 of them as must ($\mho$) errors, 14 as disjunctive may ($\top^*$) errors and 3 as may ($\top$) errors. In summary, HIPEE detected 97% of real bugs including 73%, 11% and 13% of $\mho$, $\top^*$ and $\top$ errors, respectively.

However, a few errors were not detected by our system, e.g. $\mathtt{v}_4$, $\mathtt{v}_9$ of `schedule2` and $\mathtt{v}_1$, $\mathtt{v}_2$ of `print_tokens2` were verified successfully by HIPEE. Upon careful examination, we found that the substituted statement in $\mathtt{v}_9$ is semantically equivalent with the non-faulty one in $\mathtt{v}_0$. Hence, we consider it as a bug in constructing the benchmark rather than a real program bug. For $\mathtt{v}_1$, $\mathtt{v}_2$ and $\mathtt{v}_4$, there were omitted statements that are related to the I/O systems. For instance, the following statement is omitted in $\mathtt{v}_1$:

$$\mathtt{if}(\mathtt{ch} == \mathtt{EOF})\ \mathtt{fprintf}(\mathtt{stdout}, \text{"It can not get character"});$$

This was not picked up by our system since the specification of I/O operations were not being modelled. It would be interesting to see I/O operations being modelled in future.

Our calculus further supports debugging in localizing the errors. The LOE column shows the average number of lines of program code and specification relevant to the errors for each program. We are able to provide concise (between 3-5 lines) error locations for all the bugs in the suite. Such short but accurate localizations make it easier for users to comprehend the discovered errors. The last column shows the average time which HIPEE took for verifying a faulty version of each program.

Purely from the system point of view and on the assumption that specifications have already been provided, HIPEE took on average 16 seconds for safety proving, bug finding and error localization on one faulty version of each program.

8 Related Work and Conclusion

The most relevant idea to our new specification mechanism is exception safety in Spec# language [16]. While Spec# uses *otherwise* keyword to explicate scenarios which definitely lead to exceptions, our proposal uses must error values $\mho$ to model erroneous scenarios. Hence, it is possible to integrate our mechanism into exception handling. Moreover, our specification mechanism with the error calculus has well supported our verifier not only in proving safety/functional correctness and validating input parameter (like Spec#) but also in finding and classifying bugs.

Static analysis based bug finding is not new and already exists [6, 8, 11, 13]. Recent work in first order relational logic [6, 13] also addresses the problem of finding bugs in programs with pointers and linked data structures. The method is based on under approximation for loops and heap, thus it only finds the must bugs ($\mho$) in the code. Similarly, Exorcise [11] is only capable of detecting must errors ($\mho$) based on evaluating *weakest liberal preconditions*. Since both consider only postcondition violation as a must error, they do not report on the more common bugs that are due to preconditions. Our calculus is more expressive (with uncovering not only must error but may error and with proving safety) through the help of new specification mechanism on sound and/or complete properties. Moreover, to handle pointer-based programs, while the underlying assumption in [13] is that most bugs can be found in the programs with small scope (loop unrolling) and small heap size, we have also shown how our error calculus can handle data structures with aliasing through a simple integration with separation logic.

As static analysis suffers from precision problem, there have been attempts to use dynamic or hybrid analysis for safety proving and bugs finding. An approach based on dynamic analysis to infer likely invariants from code is implemented in [3]. Invariants discovered can be used as method annotations or assumptions, which can aid static checkers in detecting bugs. This hybrid analysis uses a combination of under approximation and over approximation in different phases of analysis. Similarly, SMASH [10] integrates safety with bug finding via a synergy between static analysis and testing. In our approach we do not rely on dynamic analysis as our complete lattice can symbolically capture a richer set of possible program states. Our method integrates both bug-finding and safety proving within a single calculus, without prejudice to working with dynamic-based analyses for unknown scenarios. Other attempts are based on dual static analysis. An over-approximation for safety and another over-approximation for bugs finding was presented in [19] but it has only been applied to numerical imperative programs. Another related approach using over- and under-approximation was presented in [5]. In [5], the may and must queries correspond to safety and liveness properties. Their conditions are computed with respect to a finite abstraction for each particular property. In comparison, the conditions for our must/may error are captured in terms of symbolic (infinite) domain that relies *only* on over-approximation mechanisms.

Conclusion In this paper, we described a novel specification mechanism for both sound and complete requirements via the calculus for must/may errors. The calculus also enables bug finding (with safety checking) during modular verification. We can provide fairly precise and concise failure localization from our calculus. Using separation logic, we can support sound and complete safety verification, in the presence of data structures with sophisticated invariants, via user-defined predicates and lemmas. We have

extended an automated tool for verifying complex data structures to use our error calculus. Initial sets of experiments have shown that bug finding and safety checking via the modular vefification can be supported with modest overheads.

Acknowledgement This work is being supported by MoE research grant 2009-T2-1-063. Florin Craciun is supported by the project POSDRU 89/1.5/S/60189 "Postdoctoral Programs for Sustainable Development in a Knowledge Based Society".

References

1. VSTTE 2012 Software Verification Competition. https://sites.google.com/site/vstte2012/compet, 2012. [Online; accessed 27-July-2012].
2. W.N. Chin, C. David, H.H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.
3. Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.
4. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
5. Isil Dillig, Thomas Dillig, and Alex Aiken. Reasoning about the unknown in static analysis. *Commun. ACM*, 53(8):115–123, 2010.
6. Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a SAT solver. In *ESEC/SIGSOFT FSE*, pages 195–204, 2007.
7. A. Dolzmann and Thomas Sturm. Redlog: computer algebra meets computer logic. *SIGSAM Bull.*, 31:2–9, June 1997.
8. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.
9. Cristian Gherghina, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Structured specifications for better verification of heap-manipulating programs. In *FM*, pages 386–401, 2011.
10. P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL'10*, pages 43–56. ACM, 2010.
11. Jochen Hoenicke, K. Rustan Leino, Andreas Podelski, Martin Schäf, and Thomas Wies. It's doomed; we can prove it. FM '09, pages 338–353, 2009.
12. D. Hyunsook, E. Sebastian, and R. Gregg. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10:405–435, Oct. 2005.
13. Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. ISSTA '00, pages 14–25, 2000.
14. Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, pages 437–446, New York, NY, USA, 2011. ACM.
15. N. Klarlund and A. Moller. MONA Version 1.4 - User Manual. BRICS Notes Series, 2001.
16. K. Rustan M. Leino and Wolfram Schulte. Exception safety for c#. In *SEFM*, pages 218–227, 2004.
17. H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *CAV*, pages 355–369, 2008.
18. Peter O'Hearn. Tutorial on separation logic (invited tutorial). In *CAV*, 2008.
19. C. Popeea and W.N. Chin. Dual analysis for proving safety and finding bugs. In *SAC*, 2010.
20. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.

A Proof of the Soundness of the Structural Rules for $\vdash_C$

We prove Theorem 1 inductively on the structural rules through $\sqcup$ operator, $\otimes$ operator and $\oplus$ operator.

A.1 JOIN ($\sqcup$) Operator

$$\frac{\begin{array}{c}[\underline{\mathbf{EC}\text{–}[\sqcup\ \mathbf{JOIN}]}] \\ \pi_1 \vdash_C \pi \rightsquigarrow \tau_1 \\ \pi_2 \vdash_C \pi \rightsquigarrow \tau_2\end{array}}{\pi_1 \vee \pi_2 \vdash_C \pi \rightsquigarrow \tau \textit{ and } \tau_1 \sqcup \tau_2 = \tau}$$

We prove Theorem 1 by the case analysis on the returned τ.

Case $\tau = \bot$. Based on the lattice of program status, $\tau_1 \sqcup \tau_2 = \bot$ if $\tau_1 = \bot$ and $\tau_2 = \bot$. It means $\tau_1 \sqcup \tau_2 = \bot$ if $\pi_1 \vdash_C \pi \rightsquigarrow \bot$ and $\pi_2 \vdash_C \pi \rightsquigarrow \bot$.

Follow the entailment procedure $\vdash_C$, we have $\pi_i \vdash_C \pi \rightsquigarrow \bot$ infers that $\mathbf{UNSAT}(\pi_\mathbf{i})$ with $i \in \{1, 2\}$.

We have:

$$\begin{aligned} & \mathbf{UNSAT}(\pi_\mathbf{1}) \wedge \mathbf{UNSAT}(\pi_\mathbf{2}) \\ \equiv\ & \neg\pi_1 \wedge \neg\pi_2 \\ \equiv\ & \neg(\pi_1 \vee \pi_2) \\ \equiv\ & \mathbf{UNSAT}(\pi_\mathbf{1} \vee \pi_\mathbf{2}) \end{aligned}$$

Again, follow the entailment procedure $\vdash_C$ we conclude: $\pi_1 \vee \pi_2 \vdash_C \pi \rightsquigarrow \bot$

Therefore, $\tau_1 \sqcup \tau_2 = \tau$.

Case $\tau = \surd$. Based on the lattice of program status, $\tau_1 \sqcup \tau_2 = \surd$ if

1. $\tau_1 = \surd$ and $\tau_2 = \surd$. Or
2. One of them is $\bot$ and another is $\surd$. We assume $\tau_1 = \bot$ and $\tau_2 = \surd$

Case $\tau_1 = \surd$ and $\tau_2 = \surd$

$\tau_1 = \surd$, it means $\pi_1 \vdash_C \pi \rightsquigarrow \surd$. Follow the entailment procedure $\vdash_C$, we have:

$$\begin{aligned} & \mathbf{PSAT}(\pi_\mathbf{1}) \wedge & \text{(1.a.1)} \\ & \mathbf{UNSAT}(\pi_\mathbf{1} \wedge \neg\pi) & \text{(1.a.2)} \end{aligned}$$

Similarly, with $\tau_2 = \surd$, we have:

$$\begin{aligned} & \mathbf{PSAT}(\pi_\mathbf{2}) \wedge & \text{(1.a.3)} \\ & \mathbf{UNSAT}(\pi_\mathbf{2} \wedge \neg\pi) & \text{(1.a.4)} \end{aligned}$$

From (1.a.1) and (1.a.3), we have:

$$\begin{aligned} & \mathbf{PSAT}(\pi_\mathbf{1}) \ \wedge\ \mathbf{PSAT}(\pi_\mathbf{2}) \\ & \Rightarrow \mathbf{PSAT}(\pi_\mathbf{1} \ \vee\ \pi_\mathbf{2}) \qquad (1.1) \end{aligned}$$

From (1.a.2) and (1.a.4), we have:

$$\begin{aligned} & \neg\left(\pi_1 \wedge \neg\pi\right) \wedge \neg\left(\pi_2 \wedge \neg\pi\right) \\ \equiv & \left(\neg\pi_1 \vee \pi\right) \wedge \left(\neg\pi_2 \vee \pi\right) \\ \equiv & \equiv \left(\neg\pi_1 \wedge \neg\pi_2\right) \vee \pi \\ \equiv & \neg(\pi_1 \vee \pi_2) \vee \pi \\ \equiv & \neg((\pi_1 \vee \pi_2) \wedge \neg\pi) \\ \equiv & \textbf{UNSAT}(\pi_1 \vee \pi_2) \wedge \neg\,\pi) \end{aligned} \qquad (1.2)$$

From (1.1), (1.2), and follow the entailment procedure $\vdash_C$ we conclude: $\pi_1 \vee \pi_2 \vdash_C \pi \rightsquigarrow \surd$.
So, $\tau_1 \sqcup \tau_2 = \tau$.

Case $\tau_1 = \perp$ **and** $\tau_2 = \surd$

It means $\pi_1 \vdash_C \pi \rightsquigarrow \perp$ and $\pi_2 \vdash_C \pi \rightsquigarrow \surd$.
Follow the entailment procedure $\vdash_C$, we have:

$$\begin{aligned} & \textbf{UNSAT}(\pi_1) \wedge && (1.b.1) \\ & \textbf{PSAT}(\pi_2) \wedge && (1.b.2) \\ & \textbf{UNSAT}(\pi_2 \wedge \neg\pi) && (1.b.3) \end{aligned}$$

From (1.b.1) and (1.b.2), we have:

$$\begin{aligned} & \textbf{UNSAT}(\pi_1) \wedge \textbf{PSAT}(\pi_2) \\ \Rightarrow\ & \textbf{PSAT}(\pi_1 \vee \pi_2) \end{aligned} \qquad (1.3)$$

From (1.b.2) and (1.b.3), we infer that $\textbf{PSAT}(\pi)$ and combined with $\textbf{UNSAT}(\pi_1)$, we have $\pi_1 \implies \pi$.

Moreover, with $\pi_1 \implies \pi$ and $\pi_2 \implies \pi$, follow the same proof leading to (1.2) of **case** $\tau_1 = \surd$ **and** $\tau_2 = \surd$ we have:

$$(\pi_1 \vee \pi_2) \implies \pi \qquad (1.4)$$

From (1.3), (1.4), and follow the entailment procedure $\vdash_C$ we conclude: $\pi_1 \vee \pi_2 \vdash_C \pi \rightsquigarrow \surd$.
Therefore, $\tau_1 \sqcup \tau_2 = \tau$.

Case $\boldsymbol{\tau = \mho}$**.** Based on the lattice of program status, $\tau_1 \sqcup \tau_2 = \mho$ if

1. $\tau_1 = \mho$ and $\tau_2 = \mho$. Or
2. One of them is $\perp$ and another is $\mho$. We assume $\tau_1 = \perp$ and $\tau_2 = \mho$

Case $\tau_1 = \mho$ **and** $\tau_2 = \mho$

$\tau_1 = \mho$, it means $\pi_1 \vdash_C \pi \rightsquigarrow \mho$. Follow the entailment procedure $\vdash_C$, we have:

$$\begin{aligned} & \textbf{PSAT}(\pi_1) \wedge && (1.c.1) \\ & \textbf{UNSAT}(\pi_1 \wedge \pi) && (1.c.2) \end{aligned}$$

Similarly, with $\tau_2 = \mho$, we have:

$$\begin{aligned} & \textbf{PSAT}(\pi_2) \wedge && (1.c.3) \\ & \textbf{UNSAT}(\pi_2 \wedge \pi) && (1.c.4) \end{aligned}$$

From (1.c.1) and (1.c.3), we have:

$$\begin{aligned} & \mathbf{PSAT}(\pi_{\mathbf{1}}) \ \wedge \ \mathbf{PSAT}(\pi_{\mathbf{2}}) \\ \Rightarrow\ & \mathbf{PSAT}(\pi_{\mathbf{1}} \ \vee \ \pi_{\mathbf{2}}) \end{aligned} \qquad (1.5)$$

From (1.c.2) and (1.c.4), we have:

$$\begin{aligned} & \mathbf{UNSAT}(\pi_{\mathbf{1}} \wedge \pi) \wedge \mathbf{UNSAT}(\pi_{\mathbf{2}} \wedge \pi) \\ \equiv\ & \neg(\pi_1 \wedge \pi) \wedge \neg(\pi_2 \wedge \pi) \\ \equiv\ & (\neg\pi_1 \vee \neg\pi) \wedge (\neg\pi_2 \vee \neg\pi) \\ \equiv\ & (\neg\pi_1 \wedge \neg\pi_2) \vee \neg\pi \\ \equiv\ & \neg(\pi_1 \vee \pi_2) \vee \neg\pi \\ \equiv\ & \neg((\pi_1 \vee \pi_2) \wedge \pi) \\ \equiv\ & \mathbf{UNSAT}((\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}}) \wedge \pi) \end{aligned} \qquad (1.6)$$

From (1.5), (1.6), and follow the entailment procedure $\vdash_C$ we conclude: $\pi_1 \vee \pi_2 \vdash_C \pi \rightsquigarrow \mho$. So, $\tau_1 \sqcup \tau_2 = \tau$.

Case $\tau_1 = \bot$ **and** $\tau_2 = \mho$

$\tau_1 = \bot$, it means $\pi_1 \vdash_C \pi \rightsquigarrow \bot$. Follow the entailment procedure $\vdash_C$, we have:

$$\mathbf{UNSAT}(\pi_{\mathbf{1}}) \qquad (1.d.1)$$

$\tau_2 = \mho$, it means $\pi_2 \vdash_C \pi \rightsquigarrow \mho$. Follow the entailment procedure $\vdash_C$, we have:

$$\begin{aligned} & \mathbf{PSAT}(\pi_{\mathbf{2}}) \ \wedge \quad \text{(1.d.2)} \\ & \mathbf{UNSAT}(\pi_{\mathbf{2}} \wedge \pi) \ \text{(1.d.3)} \end{aligned}$$

From (1.d.1) and (1.d.2) we infer that:

$$\mathbf{PSAT}(\pi_{\mathbf{1}} \ \vee \ \pi_{\mathbf{2}}) \qquad (1.7)$$

From (1.d.1) we have:

$$\begin{aligned} & \mathbf{UNSAT}(\pi_{\mathbf{1}}) \\ \equiv\ & \neg\pi_1 \\ \Rightarrow\ & \neg\pi_1 \vee \neg\pi \end{aligned} \qquad \text{(1.d.4)}$$

From (1.d.3) we have:

$$\begin{aligned} & \mathbf{UNSAT}(\pi_{\mathbf{2}} \wedge \pi) \\ \equiv\ & \neg(\pi_2 \wedge \pi) \\ \equiv\ & \neg\pi_2 \vee \neg\pi \end{aligned} \qquad \text{(1.d.5)}$$

From (1.d.4) and (1.d.5) we have:

$$\begin{aligned} & (\neg\pi_1 \vee \neg\pi) \wedge (\neg\pi_2 \vee \neg\pi) \\ \equiv\ & (\neg\pi_1 \wedge \neg\pi_2) \vee \neg\pi \\ \equiv\ & \neg(\pi_1 \vee \pi_2) \vee \neg\pi \\ \equiv\ & \neg((\pi_1 \vee \pi_2) \wedge \pi) \\ \equiv\ & \mathbf{UNSAT}((\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}}) \wedge \pi) \end{aligned} \qquad (1.8)$$

From (1.7), (1.8), and follow the entailment procedure $\vdash_C$ we conclude: $\pi_1 \vee \pi_2 \vdash_C \pi \rightsquigarrow \mho$. Therefore, $\tau_1 \sqcup \tau_2 = \tau$.

Case $\tau = \top$. Based on the lattice of program status, $\tau_1 \sqcup \tau_2 = \top$ if

1. Either τ_1 or τ_2 is $\top$. Assume $\tau_1 = \top$. Or
2. $\tau_1 = \mho$ and $\tau_2 = \surd$.

Case $\tau_1 = \top$

$\tau_1 = \top$, it means $\pi_1 \vdash_C \pi \rightsquigarrow \top$.

Follow the entailment procedure $\vdash_C$, we have:

$$\begin{aligned} &\mathbf{PSAT}(\pi_\mathbf{1} \wedge \neg\pi) \wedge && (1.e.1)\\ &\mathbf{PSAT}(\pi_\mathbf{1} \wedge \pi) && (1.e.2) \end{aligned}$$

From (1.e.1) we have:

$$\begin{aligned} &\mathbf{PSAT}(\pi_\mathbf{1} \wedge \neg\pi)\\ \Rightarrow\ &\mathbf{PSAT}((\pi_\mathbf{1} \wedge \neg\pi) \vee (\pi_\mathbf{2} \wedge \neg\pi))\\ \equiv\ &\mathbf{PSAT}((\pi_\mathbf{1} \vee \pi_\mathbf{2}) \wedge \neg\pi) && (1.9) \end{aligned}$$

From (1.e.2) we have:

$$\begin{aligned} &\mathbf{PSAT}(\pi_\mathbf{1} \wedge \pi)\\ \Rightarrow\ &\mathbf{PSAT}((\pi_\mathbf{1} \wedge \pi) \vee (\pi_\mathbf{2} \wedge \pi))\\ \equiv\ &\mathbf{PSAT}((\pi_\mathbf{1} \vee \pi_\mathbf{2}) \wedge \pi) && (1.10) \end{aligned}$$

From (1.9), (1.10) and follow the entailment procedure $\vdash_C$ we conclude: $\pi_1 \vee \pi_2 \vdash_C \pi \rightsquigarrow \top$.

Therefore, $\tau_1 \sqcup \tau_2 = \tau$.

Case $\tau_1 = \mho$ **and** $\tau_2 = \surd$

$\tau_1 = \mho$, it means $\pi_1 \vdash_C \pi \rightsquigarrow \mho$. Follow the entailment procedure $\vdash_C$, we have:

$$\begin{aligned} &\mathbf{PSAT}(\pi_\mathbf{1}) \wedge && (1.f.1)\\ &\mathbf{UNSAT}(\pi_\mathbf{1} \wedge \pi) && (1.f.2) \end{aligned}$$

$\tau_2 = \surd$, it means $\pi_2 \vdash_C \pi \rightsquigarrow \surd$. Follow the entailment procedure $\vdash_C$, we have:

$$\begin{aligned} &\mathbf{PSAT}(\pi_\mathbf{2}) \wedge && (1.f.3)\\ &\mathbf{UNSAT}(\pi_\mathbf{2} \wedge \neg\pi) && (1.f.4) \end{aligned}$$

We prove $\mathbf{PSAT}(\pi_\mathbf{2} \wedge \pi)$ by contradiction. Assume that $\mathbf{UNSAT}(\pi_\mathbf{2} \wedge \pi)$.

$$\begin{aligned} &\mathbf{UNSAT}(\pi_\mathbf{2} \wedge \pi)\\ \equiv\ &\neg(\pi_2 \wedge \pi)\\ \equiv\ &\neg\pi_2 \vee \neg\pi \end{aligned}$$

Combined with (1.f.4), we have:

$$\begin{aligned} &(\neg\pi_2 \vee \neg\pi) \wedge (\neg\pi_2 \vee \pi)\\ \equiv\ &\neg\pi_2 \wedge (\neg\pi \vee \pi)\\ \equiv\ &\neg\pi_2 \quad \text{contradict with (1.f.4)} \end{aligned}$$

Hence, we conclude $\mathbf{PSAT}(\pi_{\mathbf{2}} \wedge \pi)$.

$$\begin{aligned} & \mathbf{PSAT}(\pi_{\mathbf{2}} \wedge \pi) \\ \Rightarrow\ & \mathbf{PSAT}((\pi_{\mathbf{1}} \wedge \pi) \vee (\pi_{\mathbf{2}} \wedge \pi)) \\ \equiv\ & \mathbf{PSAT}((\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}}) \wedge \pi) \qquad (1.11) \end{aligned}$$

Similarly, we can prove that

$$\mathbf{PSAT}((\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}}) \wedge \neg\pi) \quad (1.12)$$

From (1.11), (1.12) and follow the entailment procedure $\vdash_C$ we conclude: $\pi_1 \vee \pi_2 \vdash_C \pi \rightsquigarrow \top$.

Therefore, $\tau_1 \sqcup \tau_2 = \tau$.

A.2 COMPOSE ($\otimes$) Operator

$$\frac{\begin{array}{c}[\mathbf{EC}\text{-}[\otimes\ \mathbf{COMPOSE}]] \\ \pi \vdash_C \pi_1 \rightsquigarrow \tau_1 \\ \pi \vdash_C \pi_2 \rightsquigarrow \tau_2 \end{array}}{\pi \vdash_C \pi_1 \wedge \pi_2 \rightsquigarrow \tau \textit{ and } \tau_1 \otimes \tau_2 = \tau}$$

We prove Theorem 1 by the case analysis on the returned τ.

Case $\tau = \perp$. Based on $\otimes$ operator, the result of $\tau_1 \otimes \tau_2$ is $\perp$ if either τ_1 or τ_2 is $\perp$. Assume $\tau_1 = \perp$. It means $\pi \vdash_C \pi_1 \rightsquigarrow \perp$.

Follow the entailment procedure $\vdash_C$, we infer: $\mathbf{UNSAT}(\pi_{\mathbf{1}})$.

Again, follow the entailment procedure $\vdash_C$ we conclude $\pi \vdash_C \pi_1 \wedge \pi_2 \rightsquigarrow \perp$

So, $\tau_1 \otimes \tau_2 = \tau$.

Case $\tau = \surd$. Based on $\otimes$ operator, the result of $\tau_1 \otimes \tau_2$ is $\surd$ if both τ_1 and τ_2 are $\surd$. It means $\pi \vdash_C \pi_1 \rightsquigarrow \surd$ and $\pi \vdash_C \pi_2 \rightsquigarrow \surd$.

Follow the entailment procedure $\vdash_C$, we have:

$$\begin{aligned} & \mathbf{PSAT}(\pi) \wedge && (2.a.1) \\ & \mathbf{UNSAT}(\pi \wedge \neg\pi_{\mathbf{1}}) \wedge && (2.a.2) \\ & \mathbf{UNSAT}(\pi \wedge \neg\pi_{\mathbf{2}}) && (2.a.3) \end{aligned}$$

From (2.a.2) and (2.a.3), we have:

$$\begin{aligned} & \mathbf{UNSAT}(\pi \wedge \neg\pi_{\mathbf{1}}) \wedge \mathbf{UNSAT}(\pi \wedge \neg\pi_{\mathbf{2}}) \\ \Rightarrow\ & (\neg\pi \vee \pi_1) \wedge (\neg\pi \vee \pi_2) \\ \equiv\ & \neg\pi \vee (\pi_1 \wedge \pi_2) \\ \equiv\ & \mathbf{UNSAT}(\pi \wedge \neg(\pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}})) \qquad (2.1) \end{aligned}$$

From (2.a.1), (2.1), and follow the entailment procedure $\vdash_C$ we conclude: $\pi \vdash_C \pi_1 \wedge \pi_2 \rightsquigarrow \surd$

Therefore, $\tau_1 \otimes \tau_2 = \tau$.

Case $\tau = \mho$. Based on $\otimes$ operator, the result of $\tau_1 \otimes \tau_2$ is $\mho$ if one of them (τ_1, τ_2) is $\mho$, and another is not $\bot$. Assumme $\tau_1 = \mho$ and $\tau_2 \neq \bot$.

$\tau_1 = \mho$ means $\pi \vdash_C \pi_1 \rightsquigarrow \mho$. Follow the entailment procedure $\vdash_C$, we have:

$$\begin{array}{ll} \mathbf{PSAT}(\pi) \wedge & (2.b.1) \\ \mathbf{PSAT}(\pi) \wedge & (2.b.2) \\ \mathbf{UNSAT}(\pi \wedge \pi_{\mathbf{1}}) & (2.b.3) \end{array}$$

From (2.b.3), we have:

$$\begin{array}{ll} & \mathbf{UNSAT}(\pi \wedge \pi_{\mathbf{1}}) \\ \Rightarrow & \mathbf{UNSAT}(\pi \wedge \pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}}) \qquad (2.2) \end{array}$$

From (2.b.1), (2.2) and follow the entailment procedure $\vdash_C$ we conclude: $\pi \vdash_C \pi_1 \wedge \pi_2 \rightsquigarrow \mho$

Therefore, $\tau_1 \otimes \tau_2 = \tau$.

Case $\tau = \top$. Based on $\otimes$ operator, $\tau_1 \otimes \tau_2 = \top$ if

1. $\tau_1 = \top$ and $\tau_2 = \top$. Or
2. One of them (τ_1, τ_2) is $\top$, another is $\surd$. Assume $\tau_1 = \top$ and $\tau_2 = \surd$.

Case $\tau_1 = \top$ and $\tau_2 = \top$

$\tau_1 = \top$ means $\pi \vdash_C \pi_1 \rightsquigarrow \top$. Follow the entailment procedure $\vdash_C$, we have:

$$\begin{array}{ll} \mathbf{PSAT}(\pi \wedge \neg\pi_{\mathbf{1}}) \wedge & (2.c.1) \\ \mathbf{PSAT}(\pi \wedge \pi_{\mathbf{1}}) & (2.c.2) \end{array}$$

Similary, with $\tau_2 = \top$, we have:

$$\begin{array}{ll} \mathbf{PSAT}(\pi \wedge \neg\pi_{\mathbf{2}}) \wedge & (2.c.3) \\ \mathbf{PSAT}(\pi \wedge \pi_{\mathbf{2}}) & (2.c.4) \end{array}$$

From (2.c.1), (2.c.3), we have:

$$\begin{array}{ll} & \mathbf{PSAT}(\pi \wedge \neg\pi_{\mathbf{1}}) \wedge \mathbf{PSAT}(\pi \wedge \neg\pi_{\mathbf{2}}) \\ \Rightarrow & \mathbf{PSAT}((\pi \wedge \neg\pi_{\mathbf{1}}) \vee (\pi \wedge \neg\pi_{\mathbf{2}})) \\ \equiv & \mathbf{PSAT}(\pi \wedge (\neg\pi_{\mathbf{1}} \vee \neg\pi_{\mathbf{2}})) \\ \equiv & \mathbf{PSAT}((\pi \wedge \neg(\pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}})) \qquad (2.3) \end{array}$$

We prove $\mathbf{PSAT}(\pi \wedge \pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}})$ by contradiction. Assume $\neg(\pi \wedge \pi_1 \wedge \pi_2)$.

$$\begin{array}{ll} & \neg(\pi \wedge \pi_1 \wedge \pi_2) \\ \equiv & \neg((\pi \wedge \pi_1) \wedge (\pi \wedge \pi_2)) \\ \equiv & \neg(\pi \wedge \pi_1) \vee \neg(\pi \wedge \pi_2) \ (2.c.5) \end{array}$$

(2.c.5) contradicts with both (2.c.2) and (2.c.4). Hence, we conclude:

$$\mathbf{PSAT}(\pi \wedge \pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}}) \qquad (2.4)$$

From (2.3), (2.4), and follow the entailment procedure $\vdash_C$ we conclude: $\pi \vdash_C \pi_1 \wedge \pi_2 \rightsquigarrow \top$

Therefore, $\tau_1 \otimes \tau_2 = \tau$.

Case $\tau_1 = \top$ **and** $\tau_2 = \surd$

$\tau_1{=}\top$ means $\pi \vdash_C \pi_1 \rightsquigarrow \top$. Follow the entailment procedure $\vdash_C$, we have:

$$\begin{array}{ll} \mathbf{PSAT}(\pi \wedge \neg\pi_{\mathbf{1}}) \wedge & \text{(2.d.1)} \\ \mathbf{PSAT}(\pi \wedge \pi_{\mathbf{1}}) & \text{(2.d.2)} \end{array}$$

$\tau_2{=}\surd$ means $\pi \vdash_C \pi_2 \rightsquigarrow \surd$.

$$\begin{array}{ll} \mathbf{PSAT}(\pi) \wedge & \text{(2.d.3)} \\ \mathbf{UNSAT}(\pi \wedge \neg\pi_{\mathbf{2}}) & \text{(2.d.4)} \end{array}$$

From (2.d.1), we have:

$$\begin{array}{ll} & \mathbf{PSAT}(\pi \wedge \neg\pi_{\mathbf{1}}) \\ \Rightarrow & \mathbf{PSAT}((\pi \wedge \neg\pi_{\mathbf{1}}) \vee (\pi \wedge \neg\pi_{\mathbf{2}})) \\ \equiv & \mathbf{PSAT}(\pi \wedge (\neg\pi_{\mathbf{1}} \vee \neg\pi_{\mathbf{2}})) \\ \equiv & \mathbf{PSAT}((\pi \wedge \neg(\pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}})) \end{array} \qquad (2.5)$$

We prove $\mathbf{PSAT}(\pi \wedge \pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}})$ by contradiction. Assume $\neg(\pi \wedge \pi_1 \wedge \pi_2)$.

$$\begin{array}{ll} & \neg(\pi \wedge \pi_1 \wedge \pi_2) \\ \equiv & \neg(\pi \wedge \pi_1) \vee \neg\pi_2) \end{array}$$

Combined with (2.d.4), we have:

$$\begin{array}{ll} & \neg(\pi \wedge \pi_1) \vee \neg\pi_2) \wedge (\neg\pi \vee \pi_2) \\ \Rightarrow & \neg\pi \end{array}$$

This contradicts with (2.d.3).

Hence, we conlcude:

$$\mathbf{PSAT}(\pi \wedge \pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}}) \qquad (2.6)$$

From (2.5), (2.6), and follow the entailment procedure $\vdash_C$ we conclude: $\pi \vdash_C \pi_1 \wedge \pi_2 \rightsquigarrow \top$

Therefore, $\tau_1 \otimes \tau_2 = \tau$.

A.3 UNION ($\oplus$) Operator

$$\frac{\begin{array}{c}[\mathbf{EC}{-}[\oplus\ \mathbf{UNION}]] \\ \pi \vdash_C \pi_1 \rightsquigarrow \tau_1 \\ \pi \vdash_C \pi_2 \rightsquigarrow \tau_2 \end{array}}{\pi \vdash_C \pi_1 \vee \pi_2 \rightsquigarrow \tau \textit{ and } \tau_1 \oplus \tau_2 = \tau}$$

We prove Theorem 1 by the case analysis on the returned τ.

Case $\tau = \bot$. Based on $\oplus$ operator, the result of $\tau_1 \oplus \tau_2$ is $\bot$ if either τ_1 or τ_2 is $\bot$. Assume $\tau_1 = \bot$. It means $\pi \vdash_C \pi_1 \rightsquigarrow \bot$.

Follow the entailment procedure $\vdash_C$, we infer: $\textbf{UNSAT}(\pi_\mathbf{1})$.

Again, follow the entailment procedure $\vdash_C$ we conclude $\pi \vdash_C \pi_1 \vee \pi_2 \rightsquigarrow \bot$.

Therefore, $\tau_1 \oplus \tau_2 = \tau$.

Case $\tau = \surd$. Based on $\oplus$ operator, the result of $\tau_1 \oplus \tau_2$ is $\surd$ if either (τ_1 or τ_2) is $\surd$. Assume $\tau_1 = \surd$. It means $\pi \vdash_C \pi_1 \rightsquigarrow \surd$.

Follow the entailment procedure $\vdash_C$, we have:

$$\begin{array}{ll} \textbf{PSAT}(\pi) \wedge & \text{(3.a.1)} \\ \textbf{UNSAT}(\pi \wedge \neg\pi_\mathbf{1}) & \text{(3.a.2)} \end{array}$$

From (3.a.2), we have:

$$\begin{array}{ll} & \textbf{UNSAT}(\pi \wedge \neg\pi_\mathbf{1}) \\ \Rightarrow & (\neg\pi \vee \pi_1) \vee \pi_2 \\ \equiv & \neg\pi \vee (\pi_1 \vee \pi_2) \\ \equiv & \textbf{UNSAT}(\pi \wedge \neg(\pi_\mathbf{1} \vee \pi_\mathbf{2})) \quad (3.1) \end{array}$$

From (3.a.1), (3.1), and follow the entailment procedure $\vdash_C$ we conclude $\pi \vdash_C \pi_1 \vee \pi_2 \rightsquigarrow \surd$.

Therefore, $\tau_1 \oplus \tau_2 = \tau$.

Case $\tau = \mho$. Based on $\oplus$ operator, the result of $\tau_1 \oplus \tau_2$ is $\mho$ if both τ_1 and τ_2 are $\mho$.

$\tau_1 = \mho$ means $\pi \vdash_C \pi_1 \rightsquigarrow \mho$.

Follow the entailment procedure $\vdash_C$, we have:

$$\begin{array}{ll} \textbf{PSAT}(\pi) \wedge & \text{(3.b.1)} \\ \textbf{UNSAT}(\pi \wedge \pi_\mathbf{1}) & \text{(3.b.2)} \end{array}$$

Similarly, with $\tau_1 = \mho$ we have:

$$\begin{array}{ll} \textbf{PSAT}(\pi) \wedge & \text{(3.b.1')} \\ \textbf{UNSAT}(\pi \wedge \pi_\mathbf{2}) & \text{(3.b.3)} \end{array}$$

From (3.b.2) and (3.b.3), we have:

$$\begin{array}{ll} & \textbf{UNSAT}(\pi \wedge \pi_\mathbf{1}) \wedge \textbf{UNSAT}(\pi \wedge \pi_\mathbf{2}) \\ \Rightarrow & (\neg\pi \vee \neg\pi_1) \wedge (\neg\pi \vee \neg\pi_2) \\ \equiv & \neg\pi \vee (\neg\pi_1 \wedge \neg\pi_2) \\ \equiv & \neg\pi \vee \neg(\pi_1 \vee \pi_2) \\ \equiv & \neg(\pi \wedge (\pi_1 \vee \pi_2)) \\ \equiv & \textbf{UNSAT}(\pi \wedge (\pi_\mathbf{1} \vee \pi_\mathbf{2})) \quad (3.2) \end{array}$$

From (3.b.1), (3.2) and follow the entailment procedure $\vdash_C$ we conclude $\pi \vdash_C \pi_1 \vee \pi_2 \rightsquigarrow \mho$. Therefore, $\tau_1 \oplus \tau_2 = \tau$.

Case $\tau = \top$. Based on $\oplus$ operator, the result of $\tau_1 \oplus \tau_2$ is $\mho$ if one of them (τ_1, τ_2) is $\top$, and another is neither $\bot$ nor $\surd$. We assume $\tau_1 = \top$ and τ_2 is neither $\bot$ nor $\surd$.

$\tau_1 = \top$ means $\pi \vdash_C \pi_1 \rightsquigarrow \top$.

Follow the entailment procedure $\vdash_C$, we have:

$$\begin{array}{ll} \mathbf{PSAT}(\pi \wedge \neg\pi_{\mathbf{1}}) \wedge & (3.c.1) \\ \mathbf{PSAT}(\pi \wedge \pi_{\mathbf{1}}) & (3.c.2) \end{array}$$

τ_2 is neither $\bot$ nor $\surd$, then $\pi \vdash_C \pi_1 \rightsquigarrow t$ and $t \neq \bot \wedge t \neq \surd$. Follow the entailment procedure $\vdash_C$, we have:

$$\begin{array}{ll} \mathbf{PSAT}(\pi) \wedge & (3.c.1') \\ \mathbf{PSAT}(\pi \wedge \neg\pi_{\mathbf{2}}) & (3.c.3) \end{array}$$

We prove $\mathbf{PSAT}(\pi \wedge \neg(\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}}))$ by contradiction. Assume $\neg(\pi \wedge \neg(\pi_1 \vee \pi_2))$.

$$\begin{array}{ll} & \neg(\pi \wedge \neg(\pi_1 \vee \pi_2)) \\ \equiv & \neg(\pi \wedge \neg\pi_1 \wedge \neg\pi_2) \\ \equiv & \neg((\pi \wedge \neg\pi_1) \wedge (\pi \wedge \neg\pi_2)) \\ \equiv & \neg(\pi \wedge \neg\pi_1) \vee \neg(\pi \wedge \neg\pi_2) \quad (3.c.4) \end{array}$$

(3.c.4) contradicts with both (3.c.2) and (3.c.3). Hence, we conclude:

$$\pi \wedge \neg(\pi_1 \vee \pi_2) \qquad (3.3)$$

From (3.c.2), we have:

$$\begin{array}{ll} & \mathbf{PSAT}(\pi \wedge \pi_{\mathbf{1}}) \\ \Rightarrow & \mathbf{PSAT}((\pi \wedge \pi_{\mathbf{1}}) \vee (\pi \wedge \pi_{\mathbf{2}})) \\ \equiv & \mathbf{PSAT}((\pi \wedge (\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}})) \qquad (3.4) \end{array}$$

From (3.3), (3.4), and follow the entailment procedure $\vdash_C$ we conclude $\pi \vdash_C \pi_1 \vee \pi_2 \rightsquigarrow \top$. Therefore, $\tau_1 \oplus \tau_2 = \tau$.

B Proof of the Soundness of the Structural Rules for $\vdash_E$

We prove Theorem 1 inductively on the structural rules through $\sqcup$ operator, $\otimes$ operator and $\oplus$ operator.

B.1 JOIN ($\sqcup$) Operator

$$\frac{\begin{array}{c}[\underline{\mathbf{EE}\text{–}[\sqcup\ \mathbf{JOIN}]}] \\ \pi_1 \vdash_E \pi \rightsquigarrow \tau_1 \\ \pi_2 \vdash_E \pi \rightsquigarrow \tau_2 \end{array}}{\pi_1 \vee \pi_2 \vdash_E \pi \rightsquigarrow \tau \textit{ and } \tau_1 \sqcup \tau_2 = \tau}$$

We prove Theorem 1 by the case analysis on the returned τ.

Case $\tau = \perp$. The proof is similar to the proof of the soundness of the join $\sqcup$ operator for $\vdash_C$ (see A.1).

Case $\tau = \surd$. The proof is similar to the proof of the soundness of the join $\sqcup$ operator for $\vdash_C$ (see A.1).

Case $\tau = \top$. Based on the lattice of program status, $\tau_1 \sqcup \tau_2 = \top$ if either $\tau_1 = \top$ or $\tau_2 = \top$. Assume $\tau_1 = \top$.

$\tau_1 = \top$ means $\pi_1 \vdash_E \pi \rightsquigarrow \top$. Follow the entailment procedure $\vdash_E$, we have:

$$\mathbf{PSAT}(\pi_{\mathbf{1}} \wedge \neg\pi) \qquad (4.a.1)$$

From (4.a.1), we have:

$$\begin{aligned} &\Rightarrow \mathbf{PSAT}((\pi_{\mathbf{1}} \wedge \neg\pi) \vee (\pi_{\mathbf{2}} \wedge \neg\pi)) \\ &\equiv \mathbf{PSAT}((\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}}) \wedge \neg\pi) \qquad (4.1) \end{aligned}$$

From (4.1) and follow the entailment procedure $\vdash_E$ we conclude: $\pi_1 \vee \pi_2 \vdash_C \pi \rightsquigarrow \top$. Therefore, $\tau_1 \sqcup \tau_2 = \tau$.

B.2 COMPOSE ($\otimes$) Operator

$$\frac{\begin{array}{c}[\underline{\mathbf{EE}\text{-}[\otimes\ \mathbf{COMPOSE}]}] \\ \pi \vdash \pi_1 \rightsquigarrow \tau_1 \\ \pi \vdash \pi_2 \rightsquigarrow \tau_2 \end{array}}{\pi \vdash_E \pi_1 \wedge \pi_2 \rightsquigarrow \tau \textit{ and } \tau_1 \otimes \tau_2 = \tau}$$

We prove Theorem 1 by the case analysis on the returned τ ($\perp$, $\surd$, $\mho$, $\top$).

Case $\tau = \perp$ The proof is similar to the proof of the soundness of the join $\otimes$ operator for $\vdash_C$ (see A.2).

Case $\tau = \surd$ The proof is similar to the proof of the soundness of the join $\otimes$ operator for $\vdash_C$ (see A.2).

Case $\tau = \top$ Based on $\otimes$ operator, $\tau_1 \otimes \tau_2 = \top$ if either $\tau_1 = \top$ or $\tau_2 = \top$. Assume $\tau_1 = \top$.

$\tau_1 = \top$ means $\pi \vdash_E \pi_1 \rightsquigarrow \top$. Follow the entailment procedure $\vdash_C$, we have:

$$\mathbf{PSAT}(\pi \wedge \neg\pi_{\mathbf{1}}) \qquad (5.a.1)$$

From (5.a.1) we have:

$$\begin{aligned} &\mathbf{PSAT}(\pi \wedge \neg\pi_{\mathbf{1}}) \\ &\Rightarrow \mathbf{PSAT}((\pi \wedge \neg\pi_{\mathbf{1}}) \vee (\pi \wedge \neg\pi_{\mathbf{2}})) \\ &\equiv \mathbf{PSAT}(\pi \wedge (\neg\pi_{\mathbf{1}} \vee \neg\pi_{\mathbf{2}})) \\ &\equiv \mathbf{PSAT}(\pi \wedge \neg(\pi_{\mathbf{1}} \wedge \pi_{\mathbf{2}})) \qquad (5.1) \end{aligned}$$

From (5.1), and follow the entailment procedure $\vdash_E$ we conclude: $\pi \vdash_E \pi_1 \wedge \pi_2 \rightsquigarrow \top$ Hence, $\tau_1 \otimes \tau_2 = \tau$.

B.3 UNION ($\oplus$) Operator

$$\frac{\frac{[\textbf{EE}\text{-}[\oplus\ \textbf{UNION}]]}{\pi \vdash \pi_1 \rightsquigarrow \tau_1} \quad \pi \vdash \pi_2 \rightsquigarrow \tau_2}{\pi \vdash_E \pi_1 \vee \pi_2 \rightsquigarrow \tau \textit{ and } \tau_1 \oplus \tau_2 = \tau}$$

We prove Theorem 1 by the case analysis on the returned τ.

Case $\tau = \bot$ The proof is similar to the proof of the soundness of the join $\oplus$ operator for $\vdash_C$ (see A.3).

Case $\tau = \surd$ The proof is similar to the proof of the soundness of the join $\oplus$ operator for $\vdash_C$ (see A.3).

Case $\tau = \top$ Based on $\oplus$ operator, $\tau_1 \otimes \tau_2 = \top$ if both $\tau_1 = \top$ and $\tau_2 = \top$.
$\tau_1 = \top$ means $\pi \vdash_E \pi_1 \rightsquigarrow \top$. Follow the entailment procedure $\vdash_C$, we have:

$$\textbf{PSAT}(\pi \wedge \neg\pi_{\mathbf{1}}) \qquad (6.a.1)$$

Similarly, with $\tau_2 = \top$, we have:

$$\textbf{PSAT}(\pi \wedge \neg\pi_{\mathbf{1}}) \qquad (6.a.2)$$

We prove $\textbf{PSAT}(\pi \wedge \neg(\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}}))$ by contradiction. Assume we have $\neg(\pi \wedge \neg(\pi_1 \vee \pi_2))$.

$$\begin{aligned} & \neg(\pi \wedge \neg(\pi_1 \vee \pi_2)) \\ \equiv\ & \neg(\pi \wedge \neg\pi_1 \wedge \neg\pi_2) \\ \equiv\ & \neg((\pi \wedge \neg\pi_1) \wedge (\pi \wedge \neg\pi_2)) \\ \equiv\ & \neg(\pi \wedge \neg\pi_1) \vee \neg(\pi \wedge \neg\pi_2) \qquad (6.1) \end{aligned}$$

(6.1) contradicts with both (6.a.1) and (6.a.2). Hence, we conclude:

$$\textbf{PSAT}(\pi \wedge \neg(\pi_{\mathbf{1}} \vee \pi_{\mathbf{2}})) \qquad (6.2)$$

From (6.2) and follow the entailment procedure $\vdash_E$ we conclude: $\pi \vdash_E \pi_1 \vee \pi_2 \rightsquigarrow \top$
Hence, $\tau_1 \oplus \tau_2 = \tau$.