Certification for Configurable Program Analysis*

Marie-Christine Jakobs
University of Paderborn
Department of Computer Science
Germany
marie.christine.jakobs@uni-paderborn.de

Heike Wehrheim
University of Paderborn
Department of Computer Science
Germany
wehrheim@uni-paderborn.de

ABSTRACT

Configurable program analysis (CPA) is a *generic* concept for the formalization of different software analysis techniques in a single framework. With the tool CPAchecker, this framework allows for an easy configuration and subsequent automatic execution of analysis procedures ranging from data flow analysis to model checking. The focus of the tool CPAchecker is thus on *analysis*.

In this paper, we study configurability from the point of view of software *certification*. Certification aims at providing (via a prior analysis) a certificate of correctness for a program which is (a) tamper-proof and (b) more efficient to check for validity than a full analysis. Here, we will show how, given an analysis instance of a CPA, to construct a corresponding sound certification instance, thereby arriving at *configurable program certification*. We report on experiments with certification based on different analysis techniques, and in particular explain which characteristics of an underlying analysis allow us to design an efficient (in the above (b) sense) certification procedure.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Correctness proofs; F.3 [**Logic and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Semantics of Programming Languages**]: Program Analysis; G.4 [**Mathematical Software**]: Certification

General Terms

Theory, Verification

Keywords

Proof Carrying Code, Certification, Program Analysis, Configuration

*This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

Figure 1: Our approach for configurable program certification

1. INTRODUCTION

Today, software analysis has become an established research area within the field of software engineering, and the recent advances in software analysis allow to verify industrial size programs. Still, software verification remains a task which cannot be done "on-the-fly" since it requires significant computational resources. Software *certification* together with the validation of certificates for programs is one option to speed up the process of gaining confidence in the program's correctness. Certification[1] proceeds by having some analysis technique verify the correctness of the program (w.r.t. some property) and at the same time produce a proof of correctness (the certificate). Ideally, checking validity of the certificate (i.e., whether it shows correctness of the program) should be computationally more efficient than the original analysis. Seen as a very general idea, this principle has already been in use for some time, namely in various forms of *proof carrying code* [20]: Code producers attach proofs to their code which should then be easily checkable by code consumers. Today ranging from areas as diverse as hardware [16] to security applications [6, 5], all these techniques have in common that they target a specific type of property (most often memory or type safety [19, 14]) and develop specific analysis techniques tailored towards proof/certificate generation.

In this paper, we take the opposite approach. Instead of developing a certification technique for a specific property, we will present a *generic* framework for program certification which is configurable to specific analysis tasks. Our ap-

[1]Note that we do not consider certification processes like carried out by the Software Certification Consortium SCC.

proach builds on an existing framework for configurable program analysis with tool support in the form of CPAchecker [11]. CPAchecker executes an analysis *meta algorithm* generating a (structured) abstract reachability set of a given program. The meta algorithm can be steered by a number of user-supplied inputs (e.g., telling CPAchecker when to stop the analysis, when to merge states). This presents a way of uniting different program analysis techniques, ranging from data flow analysis', computing abstract information for control flow graphs, to model checking, computing a tree-like abstract structure. The generated reach set is then subject to property checking.

For the certification process, we use the – anyway generated – reach set as certificate. Similar to the analysis we develop a generic configurable certificate validation framework with a corresponding meta algorithm for certificate checking. In addition, we provide a way of (in a large number of cases automatically) generating the configuration of the certificate validation from a given configuration of the analysis. Our approach is tamper-proof in that the certificate validator only outputs "yes" in case that the program P remains unchanged ($P = P'$) and the obtained (and possibly corrupted) certificate C' is a valid certificate for the program P with respect to a desired property U (see Fig. 1). We have implemented our technique within the CPAchecker framework, and evaluated it on a number of different analysis techniques. For all of these, certificate validation is faster than analysis. We furthermore explain which characteristics of the underlying analysis technique and program at hand will bring us significant speed-up.

2. BACKGROUND

For the sake of presentation we restrict the programming language to a simple imperative language which is limited to assignments and assume statements on integer variables.[2] For this paper, we follow the program notation of [9] and model a program as a *control-flow automaton* (CFA) $P = (L, G, l_0)$, which consists of a set of locations L, a set of control flow edges $G \subseteq L \times Ops \times L$ and a program entry location $l_0 \in L$. Furthermore, V denotes the set of program variables – all variables which occur in an operation op of an edge $(\cdot, op, \cdot) \in G$. Figure 2 shows our example program `SUM` and its CFA. The CFA contains two assignment edges, one for each assignment in the program, and two assume edges, reflecting the two evaluations of the condition of the while statement.

The semantics of a program $P = (L, G, l_0)$ is defined by a labeled transition system $T(P) = (C, G, \rightarrow)$ made up by the set of all concrete states C, labels G (the control flow edges of P) and the transition relation $\rightarrow \subseteq C \times G \times C$. We write $c \xrightarrow{g} c'$ for $(c, g, c') \in \rightarrow$. A *concrete state* c is a variable assignment which assigns control variable pc to a location $l \in L$, $c(pc)$, and every program variable $v \in V$ to an integer value, $c(v)$, as well as a boolean value, $c(v_{\text{init}})$ indicating if v has been initialized. A concrete state $c \in C$ is *reachable* from a set of states $I \subseteq C$, denoted by $c \in Reach_P(I)$, if there exists a path $c_0 \xrightarrow{g_0} c_1 \xrightarrow{g_1} \cdots \xrightarrow{g_{n-1}} c_n$ in $T(P)$ such that $c_0 \in I$, $c_n = c$ and $\forall 0 \leq i < n : c_i \xrightarrow{g_i} c_{i+1}$.

[2]Our implementation in CPAchecker [11] supports programs written in the C Intermediate Language (CIL) [21].

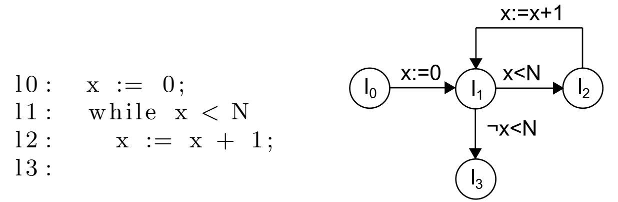

Figure 2: Program `SUM` and its control flow automaton

The *Configurable Program Analysis* (CPA), on which we build our certification approach, is a framework in which customized, abstract interpretation based program analyses are described. A CPA specifies the abstract domain together with the transfer relation for computing successors of abstract states. To integrate different analysis techniques into the framework, it furthermore has two operators steering the reachability analysis: a merge and a stop operator. Data flow analyses typically compute information along the edges of the CFA, and merge information (abstract states) at join points. They terminate once a fixpoint has been reached. On the other side, we have model checking algorithms which do not merge at join points and instead generate tree-like reachability structures. They stop exploration once an abstract state is "covered" by an already existing element. To put these two ends into a common frameworks, CPAs have merge and stop operators as parameters.

Following the definition of [9], a CPA for a program P is a four-tuple $\mathbb{A} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ composed of an abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator merge and a termination check operator stop.

1. The *abstract domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set of concrete states C, a semi-lattice $\mathcal{E}$ on a set of abstract states E and a concretization function $[\![\cdot]\!] : E \rightarrow 2^C$, which assigns every abstract state $e \in E$ its meaning.

   The semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ is defined by the set of abstract states E, a top ($\top$) and a bottom ($\bot$) element , a partial order $\sqsubseteq$ and the join operator $\sqcup$, a total function $E \times E \rightarrow E$. For soundness of the CPA $\mathbb{A}$, the semi-lattice $\mathcal{E}$ must ensure that

$$[\![\top]\!] = C \text{ and } [\![\bot]\!] = \emptyset, \tag{1}$$

$$\forall e, e' \in E : e \sqsubseteq e' \implies [\![e]\!] \subseteq [\![e']\!] \ , \tag{2}$$

$$\forall e, e' \in E : [\![e]\!] \cup [\![e']\!] \subseteq [\![e \sqcup e']\!] \ . \tag{3}$$

   Based on the partial order $\sqsubseteq$ on abstract states, we define a partial order on sets of abstract states $\sqsubseteq$:

$$S \sqsubseteq S' \text{ iff } \forall e \in S \exists e' \in S' : e \sqsubseteq e' \ .$$

2. The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E$ defines the abstract semantics.[3] Based on the control-flow edges G it determines for each abstract state its abstract

[3]More formally, we have one transfer relation per program, i.e., a relation $\rightsquigarrow_P$. Following [9] we omit P here, and assume it to be clear from the context, both as parameter to $\rightsquigarrow$ and as input to the algorithms.

successor states. For soundness of the CPA $\mathbb{A}$ the transfer relation $\rightsquigarrow$ must comply to the requirement:

$$\forall e \in E, g \in G: \quad \{c' | c \in [\![e]\!] \wedge c \xrightarrow{g} c' \in T(P)\} \subseteq \bigcup_{(e,g,e') \in \rightsquigarrow} [\![e']\!] \ . \quad (4)$$

3. The *merge operator* $\mathsf{merge} : E \times E \rightarrow E$, a total function, specifies how the information of two abstract states is combined. For soundness of the CPA $\mathbb{A}$ the result of merge may only be more abstract than the second parameter:

$$\forall e, e' \in E : e' \sqsubseteq \mathsf{merge}(e, e') \ . \quad (5)$$

4. The *termination check operator* $\mathsf{stop} : E \times 2^E \rightarrow \mathbb{B}$, a total function, examines if an abstract state is covered by a set of abstract states. To guarantee soundness of the CPA $\mathbb{A}$ the operator stop must fulfill the following requirement:

$$\forall e \in E, S \subseteq E : \mathsf{stop}(e, S) \implies [\![e]\!] \subseteq \bigcup_{e' \in S} [\![e']\!] \ . \quad (6)$$

Remark 1. [9] also describes the systematic composition of different CPAs to a Composite CPA, basically via a product construction. Since the result is also a CPA, we do not give formal details here. We will see examples of Composite CPAs in the evaluation section.

We shortly explain CPAs by introducing the CPA $\mathbb{RD}$, a reaching definition analysis [22], on our example. The aim of a reaching definition analysis (a data flow analysis) is to find out for every program variable $v \in V$ which definitions of v may reach a certain location l of a program P.

Example 1. The abstract states of the CPA $\mathbb{RD}$ are pairs (l, R) of a location $l \in L$ and a set of definitions R (plus the top $\top_{\mathbb{RD}}$ and bottom element $\bot_{\mathbb{RD}}$). A definition $r = (v, d)$ is a pair of a variable $v \in V$ and a definition point d which is either $(l, l') \in L \times L$ (v defined on edge l to l') or symbol ? (undefined). The boxes in Fig. 3 are examples for abstract states. The partial ordering on abstract states $e = (l, R)$ and $e' = (l', R')$ is $e \sqsubseteq e'$ iff $l = l'$ and $R \subseteq R'$.

The transfer relation $\rightsquigarrow_{\mathbb{RD}}$ works as follows: for an assume edge $(l, expr, l') \in G$ the abstract successor e' of abstract state $e = (l, R)$ gets the same set of reaching definitions, $e' = (l', R)$. For an assignment edge $(l, v := expr, l') \in G$ the abstract successor e' removes from R all definitions for v ($R^v = \{(v, d) | d \in L \times L \cup \{?\}\}$) and adds the new definition: $e' = (l', R')$ and $R' = ((R \setminus R^v) \cup \{(v, (l, l'))\}$. Like any data flow analysis, the CPA $\mathbb{RD}$ combines information obtained by the exploration of different branches. Hence, $\mathsf{merge}_{\mathbb{RD}}$ joins abstract states at the same control locations, $\mathsf{merge}_{\mathbb{RD}}(e, e') = (l, R \cup R')$, if $e = (l, R)$ and $e' = (l, R')$, and otherwise $\mathsf{merge}_{\mathbb{RD}}(e, e') = e'$. The termination check operator, $\mathsf{stop}_{\mathbb{RD}}(e, S) = \exists e' \in S : e \sqsubseteq e'$, checks if e does not contain any new information for e's location.

A CPA only describes the abstract domain and how to configure the behavior of a reachability analysis. Algorithm 1 displays the CPA algorithm [9], a meta reachability analysis which adapts its behavior to the configuration given by the input CPA. Algorithm 1 computes an abstraction reached (a set of reachable abstract states) for a given CPA $\mathbb{A}$ for program P and an initial abstract state e_0. In line 6 it tries to combine e' with already reached states using merge. Often, $\mathsf{merge}(e', e'')$ will give e''' (and hence $e_{\mathrm{new}} \neq e''$) when the results from the exploration of different branches should be integrated. Note that in line 10 operator stop normally returns true if e' is combined with another state e'' into e_{new}.

Algorithm 1: CPA algorithm taken from [9]

Input: CPA $\mathbb{A} = ((C, (E, \top, \bot, \sqsubseteq, \sqcup), [\![\cdot]\!]), \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$, $e_0 \in E$
Output: a set of reachable abstract states
Data: a set reached of elements of E, a set waitlist of elements of E

```
1  waitlist:={e0}; reached:={e0};
2  while waitlist≠ ∅ do
3      pop e from waitlist;
4      for each e' with (e, ·, e') ∈⇝ do
5          for each e'' ∈ reached do
6              e_new := merge(e', e'');
7              if e_new ≠ e'' then
8                  waitlist := (waitlist ∪ {e_new}) \ {e''};
9                  reached := (reached ∪ {e_new}) \ {e''};
10         if ¬stop(e', reached) then
11             waitlist := waitlist ∪ {e'};
           reached := reached ∪ {e'};
12 return reached
```

Coming back to our running example the boxes in Fig. 3 describe the abstract states obtained when executing the CPA algorithm with reaching definition analysis $\mathbb{RD}$ applied to our example program SUM and initial abstract state $e_{0,\mathbb{RD}} = (l_0, \{(x, ?), (N, ?)\})$.

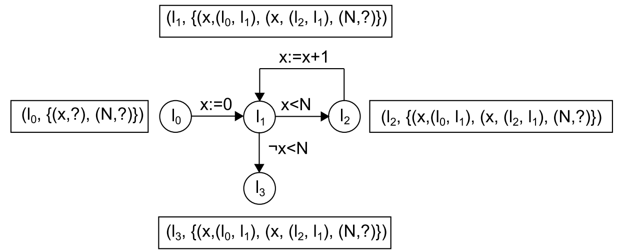

Figure 3: CFA of program SUM enriched by result of CPA $\mathbb{RD}$

To be useful the CPA algorithm must compute a sound overapproximation of the reachable concrete states. The following theorem, taken from [9], states that the abstraction computed by the CPA algorithm is indeed sound.

THEOREM 1 (SOUNDNESS OF THE CPA ALGORITHM[9]). *The result,* reached*, of the CPA algorithm for a given CPA* $\mathbb{A}$ *and an initial abstract state* e_0 *overapproximates the reachable concrete states:* $Reach_P([\![e_0]\!]) \subseteq \bigcup_{e \in \mathsf{reached}} [\![e]\!]$.

In our certification approach (like in verification) this abstraction is used to show that a program P is safe w.r.t. some property of interest. To also capture this aspect in the framework we introduce an extension of the CPA concept. In our setting safety simply means that certain unsafe states cannot be reached. To this end we use a *safety condition* which specifies the unsafe states in the abstract domain.

Definition 1. Let $D = (C, \mathcal{E}, [\![\cdot]\!])$ be an abstract domain. A *safety condition* is a set of (unsafe) abstract states $U \subseteq E_{\mathcal{E}}$.

A set of abstract states $S \subseteq E_{\mathcal{E}}$ is *safe w.r.t. a safety condition* U iff

$$\bigcup_{e \in S} [\![e]\!] \cap \bigcup_{e' \in U} [\![e']\!] = \emptyset \ .$$

A program P is *safe w.r.t. a safety condition* U *and a set of initial states* $I \subseteq C$ iff $Reach_P(I)$ is safe w.r.t. U.

To complete our running example we present a safety condition $U_{\mathbb{RD}}$ for our reaching definition analysis $\mathbb{RD}$. The safety condition $U_{\mathbb{RD}}$ states that variable N, a possible input parameter of our program `SUM`, is never redefined. Formally,

$$U_{\mathbb{RD}} = \{(l, R) | (l, R) \in E_{\mathbb{RD}} \wedge \exists l', l'' \in L : (N, (l', l'')) \in R\} \ .$$

To also show that a program P is safe w.r.t. to a safety condition U, we add an additional operator, the *safety check*, to a CPA $\mathbb{A}$ ending up with a CPA $\mathbb{A}_U$.

Definition 2. A *safety check* safe_U implementing a safety condition $U \subseteq E$ is a total function $\mathsf{safe}_U : 2^E \rightarrow \mathbb{B}$ that fulfills

$$\forall S \subseteq E : \mathsf{safe}_U(S) \implies \bigcup_{e \in S} [\![e]\!] \cap \bigcup_{e' \in U} [\![e']\!] = \emptyset \ .$$

Remark 2. The composition of program analyses as described in [9] can easily be adapted to our extension of CPAs. As long as the composite safety condition $U_\times$ is not more restrictive than all safety conditions considered by the component CPAs, $\bigcup_{e \in U_\times} [\![e]\!] \subseteq \bigcup_{i=1,\ldots,n} \bigcup_{e' \in U_i} [\![e']\!]$, the composite safety check $\mathsf{safe}_{U_\times}$ can be implemented as a conjunction of the safety checks of the component CPAs.

The safety check $\mathsf{safe}_{U_{\mathbb{RD}}}$ for the safety condition $U_{\mathbb{RD}}$ of our example is realized purely on the abstract states. It returns true if for every state $(l, R) \in S$ it holds that the set of reaching definitions R does not contain an element $(N, (l', l''))$, such that $l', l'' \in L$.

To make use of the safety check, we adapt Algorithm 1 to use CPAs $\mathbb{A}_U$. In line 12, the result of the safety check on the abstraction as well as the abstraction itself has to be returned. The adapted algorithm guarantees that if $(true, \mathsf{reached})$ is returned, then abstraction $\mathsf{reached}$ and program P are safe w.r.t. U.

Given such a configurable analysis framework, we next develop a corresponding configurable certification framework which uses the abstraction $\mathsf{reached}$ computed by CPAs as certificates.

3. CONFIGURABLE CERTIFICATION

In a certification approach an analysis, often referred to as *certificate generator*, checks the safety of a program and – upon success – produces a certificate, a witness for the program's safety. This part is usually carried out by the producer of the program (see Fig. 1). After that, the program is shipped to the consumer who validates the certificate before program execution. Validation of a certificate must be tamper-proof, i.e., a certificate must be rejected if it is not a witness for the program's correctness, possibly due to a change of the program or the certificate. Note, if the program satisfies the property but the certificate is not a witness, the certificate is rejected. Additionally, certificate validation must be more efficient than analysis (otherwise the approach is useless).

Here, we present a *configurable* program certification approach which is – like a CPA – parametric in the abstract domain and the execution of the certificate validation algorithm. As certificates it uses a safe abstraction of program P (as computed by a CPA). For certificate validation we will provide another meta algorithm, instantiable with the parameters of a configurable certificate validator (CCV). The key contribution is now that the parameters for the CCV can, in a large number of cases, automatically be computed from a given CPA.

More precisely, the certification process works as follows. The producer (certificate generator) uses some appropriate CPA to show safety of his program P with respect to some safety condition U and initial abstract state e_0. The abstraction (set $\mathsf{reached}$) computed by the CPA and used to show safety is sent as a certificate C with the program P to the consumer. The consumer receives a possibly corrupted program P' and certificate C'. First, the consumer derives a configuration for the CCV from the CPA's configuration. With this at hand he checks the validity of the certificate C' for the program P' and checks whether it certifies safety w.r.t. safety condition U and initial abstract state e_0.[4]

In the following, we present a systematic and in many cases automatic approach to get a certificate validator CCV for an arbitrary CPA and an arbitrary safety condition U. Our approach requires that the CPA, the safety condition and the initial abstract state use the same abstract domain D. The generated validator should fulfill two objectives.

- It should be *sound*, i.e., only accept valid certificates being witnesses for the safety of program P' w.r.t. U and $[\![e_0]\!]$.
- It should be *relatively complete* regarding certificate generation, i.e., if generator and validator use the same program, initial abstract state and safety condition, then the validator must accept the certificate that the generator produced.

We start the presentation of the validator with the definition of a certificate.

[4]Similar to [3], our approach allows the validator to use a stronger initial abstract state e_V, $[\![e_V]\!] \subseteq [\![e_0]\!]$, or a weaker safety condition U_V, $\bigcup_{e \in U_V} [\![e]\!] \subseteq \bigcup_{e' \in U} [\![e']\!]$.

Definition 3. Let $\mathbb{A}_U$ be a CPA. A *certificate* $C_\mathbb{A}$ is a subset of the set of abstract states E defined by the abstract domain D of $\mathbb{A}_U$, $C_\mathbb{A} \subseteq E$.

For our example, the set $C_{\mathbb{RD}} = \{(l_0, \{(x, ?), (N, ?)\})\}$ is a certificate but it does not witness the safety of program `SUM` w.r.t. safety condition $U_{\mathbb{RD}}$ and initial states I, represented by abstract state $e_{0,\mathbb{RD}}$. Certificate $C_{\mathbb{RD}}$ does not consider all concrete states reachable from I. Hence, we need to know, when it is safe to accept a certificate, i.e. when a certificate is *valid*.

Definition 4. Let P be a program, e_0 an initial abstract state and U a safety condition. A *certificate* $C_\mathbb{A}$ *is valid for* P *and* U if

1. it overapproximates the set of reachable states of P, $Reach_P(\llbracket e_0 \rrbracket) \subseteq \bigcup_{e \in C_\mathbb{A}} \llbracket e \rrbracket$,
2. and it is safe w.r.t. U.

The validator's task is to determine if a possibly modified certificate $C_\mathbb{A}$[5] is a witness for the safety of program P' which is possibly unequal to P. For this, it uses a validation rule consisting of three conditions. When all three conditions are satisfied, the validity of the certificate is guaranteed. Conditions (a) and (b) ensure that $C_\mathbb{A}$ overapproximates $Reach_{P'}(\llbracket e_0 \rrbracket)$ and condition (c) ensures the safety of the certificate. We must check for certificate $C_\mathbb{A}$ if

(a) all initial states are contained, $\llbracket e_0 \rrbracket \subseteq \bigcup_{e \in C_\mathbb{A}} \llbracket e \rrbracket$,

(b) it is closed under successor computation,
$\forall g \in G, e \in C_\mathbb{A}: \quad \{c' | c \in \llbracket e \rrbracket \wedge c \xrightarrow{g} c' \in T(P')\} \subseteq \bigcup_{e' \in C_\mathbb{A}} \llbracket e' \rrbracket$,

(c) it is safe w.r.t. U.

We want to provide a systematic and mainly automatic way to get a validation rule for a certificate of an arbitrary CPA and an arbitrary safety condition. Since we do not know the CPA nor the safety condition in advance, we cannot use a CPA independent realization of (a)-(c).

Next, we explain how we realize conditions (a) to (c). The following explanation is independent of the structure of the validator and focuses on the validator's soundness, relative completeness is discussed later. For (b) we must show that the certificate is closed under successor computation. Our idea is to recompute the abstract successors using the transfer relation $\rightsquigarrow$ of the CPA and check that the certificate is closed under abstract successor computation, $\forall g \in G$, $e \in C_\mathbb{A} : (e, g, e') \in \rightsquigarrow$ implies $\llbracket e' \rrbracket \subseteq \bigcup_{e'' \in C_\mathbb{A}} \llbracket e'' \rrbracket$. Furthermore, (a) and (b) need to examine if a set of states, given by the initial abstract state or an abstract successor, is contained in the certificate. For this, both need a *coverage check* cover which checks containment for an arbitrary abstract state and which ensures that $\mathsf{cover}(e, C_\mathbb{A})$ only returns true if $\llbracket e \rrbracket \subseteq \bigcup_{e' \in C_\mathbb{A}} \llbracket e' \rrbracket$.

[5]Modification which do not lead to certificates result in a syntactic rejection.

Condition (c) states that the certificate must be safe w.r.t. to a safety condition U. Our extended version of a CPA $\mathbb{A}_U$ already provides an operation, safe_U, which checks that a set of abstract states, e.g. a certificate, is safe w.r.t. safety condition U.

These ideas now let us establish the construction of our validator. The validator is built from two components, the *Configurable Certificate Validator (CCV)* and the CCV algorithm. We start with the CCV which must be constructed per CPA. The CCV specifies the operations used in the validation rule. To this end, it provides the abstract domain, the abstract successor computation, the transfer relation of the CPA, the coverage check as well as the safety check of the CPA. The following definition describes the construction of a CCV for a given CPA.

Definition 5. Let $\mathbb{A}_U = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{safe}_U)$ be a CPA for showing safety w.r.t. safety condition U. A *Configurable Certificate Validator (CCV)* $\mathbb{C}_{\mathbb{A}_U}$ for $\mathbb{A}_U$ is a four tuple $\mathbb{C}_{\mathbb{A}_U} = (D, \rightsquigarrow, \mathsf{cover}, \mathsf{safe}_U)$ such that the coverage check, a total function $\mathsf{cover} : E \times 2^E \to \mathbb{B}$ ensures

$$\forall e \in E, S \subseteq E : \mathsf{cover}(e, S) \implies \llbracket e \rrbracket \subseteq \bigcup_{e' \in S} \llbracket e' \rrbracket \ .$$

Remark 3. A CPA already provides a sort of coverage check. It is the termination check operator stop. For now, we will therefore use stop as coverage check.

The CCV $\mathbb{C}_{\mathbb{RD}_{U_{\mathbb{RD}}}}$ for our reaching definition example is $(D_{\mathbb{RD}}, \rightsquigarrow, \mathsf{stop}_{\mathbb{RD}}, \mathsf{safe}_{U_{\mathbb{RD}}})$.

The last component to be presented is the CCV meta algorithm. It is a parametric implementation of the validation rule. The operations needed for validation, the successor computation, the coverage check and the safety check, are described by the CCV. Next to the CCV, the CCV algorithm requires the initial abstract state and the certificate. Given these three input parameters, the CCV algorithm checks the three conditions of the validation rule. Algorithm 2 shows an implementation of the CCV algorithm.

4. SOUNDNESS AND COMPLETENESS

Before introducing the technique, we stated two requirements on our approach: soundness and relative completeness. Next, we will formally state these two properties and show when they hold. First to soundness.

THEOREM 2 (SOUNDNESS OF THE VALIDATOR). *Let P' be a program, $\mathbb{A}_U$ a CPA for showing safety w.r.t. safety condition U and e_0 an initial abstract state. If the CCV algorithm executed with CCV $\mathbb{C}_{\mathbb{A}_U}$ for $\mathbb{A}_U$, e_0, program P' and some certificate $C_\mathbb{A}$ returns true, then $C_\mathbb{A}$ is a valid certificate for P' and U.*

Algorithm 2: CCV algorithm

Input: CCV
$\mathbb{C}_{\mathbb{A}_U} = ((C,(E,\top,\bot,\sqsubseteq,\sqcup),[\![\cdot]\!]),\rightsquigarrow,\mathsf{cover},\mathsf{safe}_U)$,
initial abstract state $e_0 \in E$, certificate $C_{\mathbb{A}} \subseteq E$
Output: Boolean indicator, if certificate $C_{\mathbb{A}}$ is a valid certificate

```
// Check if initial element covered by certificate
1 if ¬cover(e_0, C_A) then
2     return false
3 for each e ∈ C_A do
4     for each (e, ·, e') ∈ ⇝ do
        // Check if successor covered by certificate
5         if ¬cover(e', C_A) then
6             return false
// Check if certificate is safe w.r.t. U
7 return safe_U(C_A)
```

Proof. Assume that the CCV algorithm executed with $\mathbb{C}_{\mathbb{A}_U}$, e_0 and $C_{\mathbb{A}}$ returns true. First, show that the certificate overapproximates the set of states reachable in P' from $[\![e_0]\!]$, $Reach_{P'}([\![e_0]\!]) \subseteq \bigcup_{e \in C_{\mathbb{A}}} [\![e]\!]$. Pick any $c \in Reach_{P'}([\![e_0]\!])$. By definition it exists a path $c_0 \stackrel{g_0}{\to} c_1 \stackrel{g_1}{\to} \cdots \stackrel{g_{n-1}}{\to} c_n$ in $T(P')$ such that $c_0 \in [\![e_0]\!]$ and $c_n = c$. Show by induction on the length of that path that $\forall 0 \leq i \leq n : c_i \in \bigcup_{e \in C_{\mathbb{A}}} [\![e]\!]$. Second, show that $C_{\mathbb{A}}$ is safe w.r.t. U. Algorithm 2 returns only true if the safety check in line 7, $\mathsf{safe}_U(C_{\mathbb{A}})$, returns true. By Definition 2 it follows that $C_{\mathbb{A}}$ is safe w.r.t. U. ☐

Hence the validator only accepts those (possibly corrupted) certificates which are valid for the program at hand and thus witness safety for it. Our certification approach is tamper-proof.

Subsequent to soundness, we discuss relative completeness of our certification approach. A relatively complete validator accepts any certificate $C_{\mathbb{A}}$ that the generator produced using the same program and initial abstract state as the validator, and that the generator successfully checked w.r.t. the safety condition. Note that all certificates $C_{\mathbb{A}}$ which these relative complete validators must accept are valid certificates. In the following, we investigate under which conditions our validator is relatively complete.

In the realization of conditions (a) and (b), we used the transfer relation of the CPA and a coverage check cover, so far implemented by the stop operator of the CPA. It is the stop operator of the CPA, which might hinder achieving completeness. To see this, we look at our running example again and define an alternative stop operation $\mathsf{stop}'_{\mathbb{RD}}(e,S) = [e = (\cdot,R) \wedge (x,(l_2,l_1)) \in R \wedge \exists e' \in S : e \sqsubseteq e']$ (only stop when the element is of a particular shape). Algorithm 1 then produces the same result for our example but $\mathsf{stop}'_{\mathbb{RD}}$ neither returns true for initial abstract state $e_{0,\mathbb{RD}}$ (an element of the certificate) nor for its abstract successor. The certificate validation would thus mistakenly return false. To rule out such cases, we need to require that cover is consistent with the partial ordering of the semi-lattice:

$$\forall S \subseteq E, e \in E : (\exists e' \in S : e \sqsubseteq e') \to \mathsf{cover}(e,S)$$

.

However, this is not sufficient yet. Again, consider another stop operator $\mathsf{stop}''_{\mathbb{RD}}(e,S) = |S| \leq 3 \wedge \exists e' \in S : e \sqsubseteq e'$ (stop depends on the size of the already constructed reach set reached). If waitlist in Algorithm 1 pops of the element with the lowest index for the location element first, it will produce the same reach set with $\mathsf{stop}''_{\mathbb{RD}}$. While constructing it (and so far having a set with size smaller 4), $\mathsf{stop}''_{\mathbb{RD}}$ returns true for element $e = (l_2, \{(x,(l_0,l_1)),(x,(l_2,l_1)),(N,?)\})$ but it does not for e and our certificate $C_{\mathbb{RD},\mathrm{r}}$ (the final valuation of reached). We have a stop operator here which is not monotonic in the increase of the reach set according to the partial ordering $\sqsubseteq$. Thus our second requirement on the stop operator (and thus for cover) is monotonicity:

$$\forall S, S' \subseteq E, e \in E \text{ s.t. } S \sqsubseteq S' : \mathsf{stop}(e,S) \to \mathsf{stop}(e,S') \ .$$

For example, the frequently used implementation of stop, $\forall e \in E, S \subseteq E : \mathsf{stop}(e,S) = \exists e' \in S : e \sqsubseteq e'$, fulfills these two conditions.

The realization of condition (c), checking if the certificate is safe w.r.t. the safety condition, does not affect relative completeness because we use the same safety check as the generator. Summarizing, to get a well-formed CCV, we need the following additional property.

Definition 6. A CCV $\mathbb{C}_{\mathbb{A}_U} = ((C,\mathcal{E},[\![\cdot]\!]),\rightsquigarrow,\mathsf{cover},\mathsf{safe}_U)$ is *well-formed* if operator cover

1. is consistent with order $\sqsubseteq$, $\forall S \subseteq E_{\mathcal{E}}, e \in E_{\mathcal{E}}$: $(\exists e' \in S : e \sqsubseteq e') \implies \mathsf{cover}(e,S)$,
2. is monotonic, $\forall S, C_{\mathbb{A}} \subseteq E_{\mathcal{E}}, e \in E_{\mathcal{E}}$ s.t. $S \sqsubseteq C_{\mathbb{A}}$: $\mathsf{cover}(e,S) \implies \mathsf{cover}(e,C_{\mathbb{A}})$,
3. is consistent with stop, $\forall S \subseteq E_{\mathcal{E}}, e \in E_{\mathcal{E}}$: $\mathsf{stop}(e,S) \implies \mathsf{cover}(e,S)$.

If the operator stop of the given CPA is not consistent with order $\sqsubseteq$ or is not monotonic, the consumer must provide an own realization for cover. Note that only in this case our approach is not fully automatic.

We now show that if our validator uses a well-formed CCV $\mathbb{C}_{\mathbb{A}_U}$, it will indeed be relatively complete regarding certificate generation.

Theorem 3 (Relative completeness of validator). *Let P be a program, $\mathbb{A}_U$ a CPA for showing safety w.r.t. safety condition U and e_0 an initial abstract state. If the CPA algorithm (Algorithm 1) executed with $\mathbb{A}_U$, P and e_0 returned $(true, C_{\mathbb{A}})$ and the CCV $\mathbb{C}_{\mathbb{A}_U}$ for $\mathbb{A}_U$ is well-formed, then the CCV algorithm (Algorithm 2) executed with $\mathbb{C}_{\mathbb{A}_U}$, program $P' = P$, e_0 and $C_{\mathbb{A}}$ returns true.*

Proof Sketch. Proof by contradiction. Assume $C_{\mathbb{A}}$ is a valid certificate computed by the CPA algorithm for $\mathbb{A}_U$, P and e_0 and CCV algorithm executed with well-formed $\mathbb{C}_{\mathbb{A}_U}$,

program $P' = P$, e_0 and $C_{\mathbb{A}}$ returns false. Distinguish three cases, Algorithm 2 returns false in line 2, 6 or 7. For line 2 show that returning false contradicts the first condition of Definition 6. For line 6 note that for every abstract state $e \in C_{\mathbb{A}}$ all its abstract successors (also e') are computed by the CPA algorithm (Algorithm 1) and considered in line 10. Distinguish two cases. First, $\mathsf{stop}(e', \mathsf{reached})$ returned true. By construction of $C_{\mathbb{A}}$ it holds that $\mathsf{reached} \sqsubseteq C_{\mathbb{A}}$. Thus, returning false contradicts the second or third condition of Definition 6. Second, $\mathsf{stop}(e', \mathsf{reached})$ returned false. By construction of $C_{\mathbb{A}}$ exists $e'' \in C_{\mathbb{A}} : e' \sqsubseteq e''$. Again, returning false contradicts the first condition of Definition 6. For line 7 show that returning false contradicts CPA algorithm returning $(true, C_{\mathbb{A}})$. $\square$

Like many verification approaches our certification approach is not fully complete, i.e., although certificates generated by a CPA are accepted, this might not be the case for arbitrary valid certificates. This is due to the fact that we are working on an abstraction of the program and never carry out calculations on the concrete state space. Hence, a very precise certificate which exactly describes the concrete state space of the program might be rejected since it is not closed under abstract successor computation.

5. EXPERIMENTAL RESULTS

In our experiments we study the efficiency of our certification approach. Since the worst case execution times of the CPA and the CCV algorithm are the same, we study in particular how much faster certificate validation is than analysis. In addition, we are interested in finding out which characteristics of the analysis lead to efficient certificate validations and for which type of programs.

For the evaluation we used five different CPAs, some being combinations of basic CPAs. Our reaching definition CPA is named $\mathbb{RD}$-DF, computing reaching definitions using an ordinary data flow analysis. In addition we have a reaching definition analysis $\mathbb{RD}$-ALL computing only one set for the whole program. This gives us an example of a very coarse abstraction, in which the certificate just contains one abstract state. The second type of analysis is constant propagation $\mathbb{CP}$-DF [22], executed as data flow analysis. The last two CPAs are combined ones, the first $\mathbb{RD} + \mathbb{CP}$-DF combining reaching definition and constant propagation (computed via data flow analysis), the second combining the same, but now merging on the same location only when the concrete values for the variables are the same. This is in spirit close to explicit model checking and presents the finest abstraction, yielding the largest certificates. Note that it could be seen as an instance of trace partitioning [23] but our intention is to configure an analysis finer than data flow analyses. A simple way to configure such a finer analysis is to combine two analysis and merge only the states of one analysis. The analyses have been chosen as to span a broad range of characteristics, not because we think they are typical analyses for which a certification will frequently be needed.

We integrated our approach into CPAchecker [11] (svn v8140) using the operator stop for cover, which in all five cases gives us well-formed CCVs. Our experiments were performed on a Intel® Core™i7-2620M @ 2.70GHz running a 64 bit Ubuntu 12.04 LTS[6] with 3600 MB RAM. The installed Java version was OpenJDK 7u9. For comparability of analysis and validation we turned off the just in-time compilation off.

Table 1 shows our experimental results for all five CPAs on three classes of programs, locks, ssh (bitvectors) and ntdrivers (simplified) taken from the SV-COMP Benchmark [8]. For each CPA, the analysis time (An) (of the CPA algorithm), the validation time (Val) (of the CCV algorithm plus the time for certificate reading), and the speed up $S = \frac{\text{An}}{\text{Val}}$ are presented. The first observation gained from the experiments is that certificate validation is only in very rare cases slower than analysis. This is caused by the need for reading in the certificate, which in all cases consumes a considerable amount of time of the validation. So one conclusion is that it is worthwhile implementing more efficient ways of storing and reading certificates.

The second observation is that all types of data flow analyses are good candidates for an efficient certification, in particular when the program at hand has lots of loops or nested branching structures (which the locks programs do not have). Specifically, a higher ratio of the number of join locations to the number of program locations is better. In such cases a speedup greater 10 can be achieved. Certificate validation is faster in these cases because in contrast to the analysis the validation has to carry out no fixpoint computations (or, in terms of the CPA, no complex merges). An exception to this is $\mathbb{CP}$-DF, because the information extracted by the analysis (constant values of variables) can already be used to speedup analysis (to avoid looking at particular branches of the program because of the knowledge about values of branching conditions). In such situations, a low number of merge operations are needed in the CPA, and thus knowing the certificate gives the validation algorithm not much advantage over the analysis.

Further on, our results show that certification for very coarse analyses like $\mathbb{RD}$-ALL as well as very fine analyses like $\mathbb{RD} + \mathbb{CP}$-JS can still be efficient.

Finally, we report on our experiences with the certificate size. First, if the analysis technique is coarser the certificate is smaller. Second, certificates for analyses with less complex abstract domains are smaller. All in all, only a certificate for the coarsest analysis $\mathbb{RD}$-ALL has a size smaller than or similar to the program.

6. CONCLUSION

In this paper, we presented a configurable program certification framework based on configurable program analysis [9]. We showed that the instance of the certificate validator can in a large number of cases be automatically derived from the given program analysis instance. We proved soundness and relative completeness of our technique, for the latter assuming the program analysis to be of a particular form. Our experiments confirmed the feasibility of the approach.

Related Work. To our knowledge, there is no configurable

[6] Actually, Ubuntu was executed in the virtual machine Virtual Box version 4.2.2 r81494 running on a 64 bit Windows 7 Professional machine with 6 GB RAM.

Table 1: Experimental results

Source	$\mathbb{RD}$-ALL			$\mathbb{RD}$-DF			$\mathbb{CP}$-DF			$\mathbb{RD}+\mathbb{CP}$-DF			$\mathbb{RD}+\mathbb{CP}$-JS		
	An	Val	S	An	Val	S	An	Val	S	An	Val	S	An	Val	S
locks_5	0.05	0.04	1.36	0.07	0.08	0.85	0.05	0.08	0.66	0.08	0.12	0.70	5.40	2.91	1.86
locks_6	0.04	0.04	1.20	0.08	0.09	0.89	0.05	0.08	0.65	0.11	0.13	0.84	43.61	19.12	2.28
locks_7	0.05	0.04	1.44	0.10	0.10	0.96	0.07	0.09	0.77	0.11	0.15	0.76	380.93	151.77	2.51
s3_clnt_1	0.42	0.07	6.09	8.73	0.60	14.63	0.53	0.29	1.81	13.63	0.75	18.05	134.27	11.57	11.61
s3_clnt_2	0.40	0.07	6.03	7.72	0.57	13.63	0.49	0.24	2.03	11.21	0.68	16.53	645.23	42.78	15.08
s3_clnt_3	0.42	0.07	6.18	6.44	0.55	11.77	0.50	0.25	2.02	10.23	0.67	15.35	Timeout after 15 min		
s3_srvr_1	0.44	0.07	5.98	10.40	0.63	16.56	0.22	0.27	0.82	11.76	0.77	15.30	173.26	13.50	12.83
s3_srvr_2	0.45	0.07	6.52	7.95	0.61	13.00	0.25	0.27	0.92	9.81	0.79	12.48	Timeout after 15 min		
s3_srvr_3	0.45	0.07	6.42	9.98	0.60	16.69	0.21	0.27	0.78	10.91	0.76	14.40	169.42	13.35	12.70
cdaudio	4.91	0.39	12.76	45.95	4.79	9.60	8.65	6.90	1.25	21.60	9.12	2.37	37.22	29.13	1.28
diskperf	1.57	0.14	11.38	48.03	1.56	30.82	2.34	1.87	1.25	20.26	2.50	8.10	Timeout after 15 min		
floppy3	1.69	0.13	12.56	18.68	1.24	15.00	2.84	2.19	1.30	12.24	2.68	4.57	16.84	17.45	0.96
floppy4	3.30	0.24	13.95	24.46	1.92	12.75	3.78	2.92	1.30	16.23	4.11	3.95	32.82	27.41	1.20
kbfiltr1	0.41	0.06	6.77	4.47	0.41	10.83	0.69	0.61	1.13	1.68	0.79	2.12	11.00	7.00	1.57
kbfiltr2	1.16	0.10	11.62	11.42	0.77	14.80	1.40	1.25	1.12	3.61	1.53	2.36	18.58	13.74	1.35
Average			7.55			12.19			1.19			7.86			5.44

certification approach which can handle such a broad range of program analysis techniques.

There are several approaches, e.g. [24, 7, 13, 3, 4], which consider certification for static program analysis but only [24, 3, 4] use an abstraction as certificate as we do. In principle, all three re-execute one special analysis step to check if the abstraction is a fixpoint. [24, 4] certify data flow analyses on byte code level. Both keep only those elements of the abstraction which are affected by backward edges in the control flow and recompute the missing part during validation. [3] is the certification of static program analysis closest to ours. It certifies abstract interpretation for source programs, however for constraint logic programms not C, and checks safety by checking if a verification condition computed from abstraction and safety specification is valid.

For model checking even more certification approaches exist. We do not discuss approaches based on temporal logic nor [1] which is based on rewriting logic. In [26] a predicate abstraction is computed and transformed into a Boolean program, the certificate. The validator checks that the Boolean program is a valid abstraction of the program and then model checks it. Similar to us, [17, 18, 15] certify that unsafe states are unreachable. In principal, a similar type of validation rule is used there, however, always specified to the particular analysis carried out. [15] transforms the abstract model computed during model checking of a concurrent program into a verification diagram, the certificate. [17, 18] also consider C programs but they only use predicate abstraction during analysis. While in [18] the abstraction is the certificate, in [17] the certificate is a proof of the premises of their rule.

In [12] a regression verification technique for abstraction-based analyses with counterexample-guided abstraction refinement is presented. This technique may also be used for certification. The core idea is to store the granularity of abstraction, the information used to guide the abstract successor computation, and reuse it as initial granularity in the next run. In contrast to our approach, the technique is not tamper-proof and not, if program or certificate changed and the stored granularity (the certificate) is not sufficient to show safety further refinement steps will follow, the behavior intended for regression verification. Furthermore, the technique is not efficient when no refinement is applied. Then, verification and certificate checking are the same.

Future Work. In the future, we plan to work on decreasing our certificate size. Similar to e.g. [24, 2, 4], we want to remove those elements from the abstraction which are recomputed during validation. A first idea is to keep elements which are computed for program locations which join control flows. Furthermore, we will try to speed up validation by parallelization. A different idea is the extension of our approach to configurable analysis with dynamic precision adjustment [10], an extension of the CPA concept. A slightly different idea, but with the same purpose, is to develop a generic framework for the certification alternative presented in [25]. Instead of providing a certificate, the approach presented in [25] (based on a type-state analysis) uses the analysis result to transform the program into a more easy checkable one. A generic framework would need to generalize this to arbitrary types of analysis, as to get a configurable approach again.

7. REFERENCES

[1] M. Alba-Castro, M. Alpuente, and S. Escobar. Automatic certification of java source code in rewriting logic. In *FMICS*, pages 200–217, 2007.

[2] E. Albert, P. Arenas-Sánchez, G. Puebla, and M. V. Hermenegildo. Reduced certificates for abstraction-carrying code. In *ICLP*, pages 163–178, 2006.

[3] E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-carrying code. In *LPAR*, pages 380–397, 2004.

[4] W. Amme, M.-A. Möller, and P. Adler. Data flow analysis as a general concept for the transport of verifiable program annotations. *Electr. Notes Theor. Comput. Sci.*, 176(3):97–108, 2007.

[5] T. Amtoft, J. Dodds, Z. Zhang, A. W. Appel, L. Beringer, J. Hatcliff, X. Ou, and A. Cousino. A certificate infrastructure for machine-checked proofs of conditional information flow. In *POST*, pages 369–389, 2012.

[6] L. Bauer, M. A. Schneider, E. W. Felten, and A. W.

Appel. Access control on the web using proof-carrying authorization. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX (2)*, pages 117–119, 2003.

[7] F. Besson, T. P. Jensen, and D. Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci.*, 364(3):273–291, 2006.

[8] D. Beyer. Second competition on software verification - (summary of SV-COMP 2013). In *TACAS*, pages 594–609, 2013.

[9] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *CAV*, pages 504–518, 2007.

[10] D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *ASE*, pages 29–38, 2008.

[11] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, pages 184–190, 2011.

[12] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. Precision reuse for efficient regression verification. In *ESEC/SIGSOFT FSE*, pages 389–399, 2013.

[13] A. Chaieb. Proof-producing program analysis. In *ICTAC*, pages 287–301, 2006.

[14] K. Crary and S. Weirich. Resource bound certification. In *POPL*, pages 184–198, 2000.

[15] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. SLAB: a certifying model checker for infinite-state concurrent systems. In *TACAS*, pages 271–274, 2010.

[16] S. Drzevitzky, U. Kastens, and M. Platzner. Proof-carrying hardware: Towards runtime verification of reconfigurable modules. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 189–194, 2009.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV*, pages 526–538, 2002.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. Extreme model checking. In *Verification: Theory and Practice*, pages 332–358, 2003.

[19] K. Klohs and U. Kastens. Memory requirements of java bytecode verification on limited devices. *Electr. Notes Theor. Comput. Sci.*, 132(1):95–111, 2005.

[20] G. C. Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.

[21] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.

[22] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer Heidelberg, 2004.

[23] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Transaction on Programming Languages and Systems*, 29(5), 2007.

[24] E. Rose. Lightweight bytecode verification. *J. Autom. Reasoning*, 31(3-4):303–334, 2003.

[25] D. Wonisch, A. Schremmer, and H. Wehrheim. Programs from proofs - a PCC alternative. In *CAV*, pages 912–927, 2013.

[26] S. Xia and J. Hook. Certifying temporal properties for compiled C programs. In *VMCAI*, pages 161–174, 2004.