Domain Types: Abstract-Domain Selection Based on Variable Usage*

Sven Apel[1], Dirk Beyer[1], Karlheinz Friedberger[1], Franco Raimondi[2], and Alexander von Rhein[1]

[1] University of Passau, Germany
[2] Middlesex University, London, UK

Abstract. The success of software model checking depends on finding an appropriate abstraction of the program to verify. The choice of the abstract domain and the analysis configuration is currently left to the user, who may not be familiar with the tradeoffs and performance details of the available abstract domains. We introduce the concept of *domain types*, which classify the program variables into types that are more fine-grained than standard declared types (e.g., 'int' and 'long') to guide the selection of an appropriate abstract domain for a model checker. Our implementation on top of an existing verification framework determines the domain type for each variable in a pre-analysis step, based on the usage of variables in the program, and then assigns each variable to an abstract domain. Based on a series of experiments on a comprehensive set of verification tasks from international verification competitions, we demonstrate that the choice of the abstract domain per variable (we consider one explicit and one symbolic domain) can substantially improve the verification in terms of performance and precision.

1 Introduction

One of the main challenges in software model checking is to automatically select, for each program variable, an abstract representation (also known as *abstract domain*) that allows to effectively prove the program correct or to identify an error path. Several abstract domains have been applied successfully to software-verification problems, with different strengths and weaknesses. Abstract domains can be based on explicit representations (e.g., hash tables for integers, memory graphs for the heap) and symbolic representations (predicates, binary decision diagrams (BDD)). For example, using an explicit-value domain [14] was efficient on many benchmarks from the recent competition on software verification [9], while using a BDD domain [15] was more efficient on event-condition-action (ECA) systems that involve only simple operations over integers in an ECA competition [30]. In the context of product-line verification, it has been shown that BDD-encodings of feature variables improve verification performance [5, 24]. The key insight is that different abstract domains are successful on different programs, and for every abstract domain, we can find programs for which the abstract domain is not successful.

* A preliminary version was published as Technical Report MIP-1303 in May 2013 [3].

So far, the choice of the abstract domain for a given verification problem (which often implies the choice of a certain verification tool as well) was left to the user. Our goal is to automate the choice of an effective abstract domain. We analyze the usage of program variables before the model checker starts the state-space exploration and assign each variable to a certain domain type. In addition to the declared type of a variable (e.g., int and char), the *domain type* represents information about the value range and the operations in which the variable is involved.

Our approach is based on the CPA verification framework, in which each abstract domain has a *precision* associated with it [11]. We use the domain types from the pre-analysis as guidance for assigning an abstract domain to each variable. In the experiments that we conducted to evaluate our approach, we use two abstract domains: an explicit-value domain and a BDD-based domain. For both domains, the precision is a set of variables that should be tracked in the domain. The precisions are initialized based on the variables' domain types. The domain assignment improves the overall verification performance, if each abstract domain tracks the kind of variables that it is suited for.

The analysis is implemented in the verification framework CPAchecker [13], which implements configurable program analysis for C programs and provides abstract domains for an explicit-value analysis and a BDD-based analysis (we do not use the predicate analysis). We evaluate our approach on six sets of verification tasks from different application domains (a total of 2 435 files) that have been used by recent international competitions on software model checking (SV-COMP 2013 [9], RERS Challenge 2012 [30]).

Our evaluation reveals that the programs in the benchmark sets contain a significant number of variables that have a much narrower domain type than the declared type of the variable. We also demonstrate that the verification performance improves if these variables are tracked using a more suitable abstract domain, compared to using a single abstract domain for all variables. All results are available on the supplementary website[1].

Example. We illustrate our approach on the example program in Fig. 1. The program contains three variables that are declared by the programmer as int. The variables are used in different ways: the variable enabled is used as a boolean; the variables a and b are numeric and used in a greater-than comparison, b is also used in a multiplication. Neither the explicit-value analysis nor the BDD-based analysis is able to efficiently verify such a program: The explicit-value domain is perfectly suited to handle variable b, because b has a concrete value, and the multiplication and the greater-than comparison can easily be computed; BDDs are known to be inefficient for multiplication [31]. The BDD domain can efficiently encode the variables enabled and a, whereas the explicit-value analysis is not good at encoding facts like a > 5.

```
int enabled, a, b;
b = 20;
if (enabled) {
  if (a > 5) {
    if (a == 0) {
      b = 0;
    }
    assert (b * b > 200);
  }
}
```

Fig. 1. Example with int variables of different domain types

[1] http://www.sosy-lab.org/projects/domaintypes/

Thus, without information about variable a, the explicit-value analysis does not know the value of variable b and cannot determine the result of the multiplication.

It has been proposed to use several abstract domains in parallel, with each domain handling all variables (e.g. [17]). If the domains are well communicating (reduced product), this could solve the verification task, but the load on each domain would be unnecessarily high, because every domain has to handle more variables than necessary.

Contributions. We make the following contributions:

- We introduced the concept of domain types and developed a pre-analysis that computes the domain types for all program variables.
- We extended an existing verification framework to use the two abstract domains 'explicit-value' and 'BDD' in parallel, while controlling the precision of each abstract domain (the variables to track) separately, based on domain types.
- We evaluate our approach on verification benchmarks from recent international software-verification competitions.

2 Background

We informally explain the concepts that we use, and provide references to the literature for details. As context, we assume to verify C programs with integer variables.

Abstract Domains and Program Analysis. Abstraction-based software model checkers automatically extract an abstract model of the subject program and explore this model using one or more abstract domains. An abstract domain represents certain aspects of the concrete program's states that the state exploration is supposed to track [1]. Different abstract domains can track different aspects of the program state space and complement each other. For example, a *shape domain* [12, 26, 34] stores, for each tracked pointer, the shape of the pointed-to data structures on the heap. Another example is the *explicit-value domain* that, for each tracked variable, tracks the explicit value of the variable [14, 28, 29]. These two examples illustrate that abstract domains can represent different information. However, it is also possible to use different abstract domains to represent the same information in different ways. Consider a program in which the value of variable x ranges from 3 to 9. This can be stored by an *interval domain* [17] using the abstract state $\mathsf{x} \mapsto [3, 9]$, or by a *predicate domain* [7, 10, 27] using the abstract state $\mathsf{x} \geq 3 \wedge \mathsf{x} \leq 9$.

Every abstract domain consists of (1) a representation of sets of concrete states, defining the abstract states (lattice elements), (2) an operator to decide if one abstract state subsumes another abstract state (partial order), and (3) an operator that combines two abstract states into a new abstract state that represents both (join). Software verifiers use one or several abstract domains to represent the states of the program. The characteristics of the abstract domain have implications on the effectivity (low number of failures and false results) and efficiency (performance) of the program analysis.

Precision. Each abstract domain can operate at different levels of abstraction (i.e., it can be more fine-grained or more coarse-grained). The level of abstraction of an abstract domain is determined by the *abstraction precision*, which controls if the analysis is coarse or fine. For example, the precision of the shape domain could instruct the analysis which pointers to track and how large a shape can maximally grow; the precision of the

```
1 int x, y, z;
2 x = 5;
3 if (y > 1) {
4   z = 2;
5 } else {
6   z = 2 ∗ x / 5;
7 }
8 ...
```

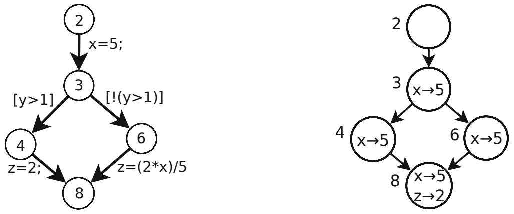

Fig. 2. Example program (left), control-flow automaton (CFA) that represents the program (middle), and abstract reachability graph (ARG, right) for the explicit-value domain. CFA edges model assume operations (e.g., [y > 1]) and assignment operations (e.g., z = 2;).

predicate domain is a set of predicates to track that can, for example, grow by adding predicates during refinement steps [23].

Next, we describe the two abstract domains that we consider in our experiments.

Explicit-Value Domain. The explicit-value domain stores explicit values for program variables. Each abstract state of this abstract domain is a map that assigns to each program variable that occurs in the precision, an integer value (or no value if an explicit value cannot be determined). For example, consider the code, the control-flow automaton (CFA), and the abstract reachability graph (ARG) in Fig. 2: the assignment of value 5 to variable x is stored in an abstract state for CFA node 3. Then, a conditional statement starts two possible execution paths, which the verifier has to explore. The explicit-value domain does not store a value for variable y, because there is no explicit value for y. After both branches of the CFA are explored, the ARG contains a 'frontier' abstract state that is the result of joining the abstract successors from both branches for CFA node 8. The explicit-value domain might suffer from a loss of information if no explicit values can be determined (e.g., for y > 1). On the one hand, this introduces imprecision and potentially false alarms. On the other hand, if values are present, all operations can be executed extremely fast. The precision controls which variables are tracked in the explicit-value domain. For the code fragment in Fig. 2, we could use a precision $\{x, z\}$ and omit y, if we knew beforehand that it is not necessary to represent variable y.

BDD Domain. The BDD domain stores information about program variables using binary decision diagrams (BDD). Each abstract state in the BDD domain is a BDD that represents a predicate over the variable values [18]. BDDs can be efficient in representing predicates and performing boolean operations. Because of this characteristic, BDDs have been used in model checking of systems with a large number of boolean variables, most prominently in hardware verification [20, 31]. Values of integer variables can be represented by BDDs using a binary encoding of the values (representing the integer values using, e.g., 32 boolean BDD variables). We can represent a variable with even fewer BDD variables if we can statically determine the set of values that the variable might hold at run time and that (non-) equality is the only arithmetical operation (nominal scale [37]). In our example, there is only one value for variable x (i.e., x = 5), and thus we need only one boolean variable for program variable x. The size of the BDD —and thus, the performance of the BDD operations— depends on the number of BDD variables; therefore, it is important to keep the number of BDD variables small.

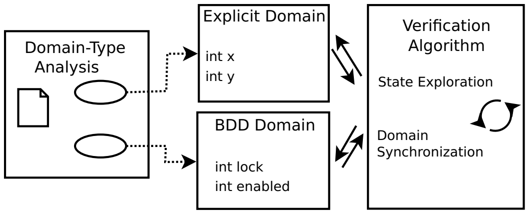

Fig. 3. A model-checking engine with two abstract domains and domain-type analysis

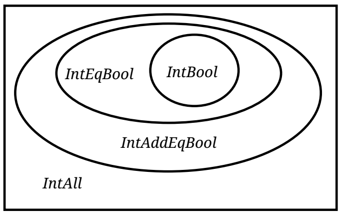

Fig. 4. Hierarchy of domain types

The abstraction precision of the BDD domain is (also) a set of program variables that an analysis should track using this abstract domain. Considering again our example of Fig. 2, if we knew beforehand that the explicit-value domain can efficiently represent variables x and z, we would not include them in the BDD precision, which would result in precision $\{y\}$ for the BDD domain, and thus we would need only BDD variables for y. Because the performance of BDD operations decreases with a growing number of variables, the BDD domain should be used only for variables that the explicit-value analysis can not efficiently track. To achieve the goal of a better assignment of program variables to abstract domains, we introduce the concept of domain types in Section 3.

3 Domain Types

The domain-type-based verification process consists of three steps: (1) The subject program is type-checked to determine the domain type for each variable (pre-analysis). (2) Each variable is mapped to an abstract domain that the analysis will use to represent information about the variable. (3) The actual verification procedure with the initialized precisions per abstract domain is started. Fig. 3 illustrates the approach of a verification engine that is based on domain types. The state-exploration algorithm uses several abstract domains to represent the state space of the program.

3.1 Classification

In many statically-typed programming languages, variables are declared to be of a certain type. The type determines which values can be stored in the variable and which operators are allowed on the variable. For the assignment of abstract domains to variables in a program analysis, more specific information on the variables are valuable, in particular, which of the operators that the static type allows are actually applied to the variable. For example, consider boolean variables in the programming language C. The language C does not provide a type 'boolean'. In C, the boolean values true and false are represented by the integer values 1 and 0, respectively. When integer variables are read, the value 0 is interpreted as false and all other values

```
int enabled;
if (enabled) {
    ...
} else {
    ...
}
```

Fig. 5. Using an integer variable as boolean in C

Syntax definition

op	::=		*program operations:*
		[expr]	*assume*
	|	x = expr;	*assignment*
expr	::=		*expressions:*
	|	val	*value*
	|	!expr	*negation*
	|	expr == expr	*equality*
	|	expr != expr	*inequality*
	|	expr + expr	*addition*
	|	expr – expr	*subtraction*
	|	expr * expr	*multiplication*
	|	expr / expr	*division*
val	::=		*values:*
		0	*zero*
	|	c	*non-zero constant*
	|	x	*variable*

Type rules for program operations

$$\frac{\mathsf{expr} \;:\; \tau}{[\,\mathsf{expr}\,] \;:\; \tau} \qquad \text{(Assume)}$$

$$\frac{\mathsf{expr} \;:\; \tau}{\mathsf{x = expr;} \;:\; \tau} \qquad \text{(Assignment)}$$

$$\frac{\begin{array}{ll} uses(\mathsf{op}_1, \mathsf{x}) & \mathsf{op}_1 \;:\; \tau_1 \\ uses(\mathsf{op}_2, \mathsf{x}) & \mathsf{op}_2 \;:\; \tau_2 \end{array}}{\mathsf{op}_1 \;:\; max(\{\tau_1, \tau_2\})} \qquad \text{(Closure)}$$

$$\frac{uses(\mathsf{op}, \mathsf{x}) \quad \mathsf{op} \;:\; \tau}{\mathsf{x} \;:\; \tau} \qquad \text{(VarUsage)}$$

Type rules for expressions

$$\frac{\mathsf{expr} \;:\; \tau}{!\,\mathsf{expr} \;:\; max(\{\tau, \mathit{IntBool}\})} \qquad \text{(NegBool)}$$

$$\frac{\mathsf{val} \;:\; \tau}{\begin{array}{l} \mathsf{val} == 0 \;:\; max(\{\tau, \mathit{IntBool}\}) \\ \mathsf{val} \,!= 0 \;:\; max(\{\tau, \mathit{IntBool}\}) \end{array}} \qquad \text{(EqBool)}$$

$$\frac{\mathsf{expr}_1 \;:\; \tau_1 \qquad \mathsf{expr}_2 \;:\; \tau_2}{\begin{array}{l} \mathsf{expr}_1 == \mathsf{expr}_2 \;:\; max(\{\tau_1, \tau_2, \mathit{IntEqBool}\}) \\ \mathsf{expr}_1 \,!= \mathsf{expr}_2 \;:\; max(\{\tau_1, \tau_2, \mathit{IntEqBool}\}) \end{array}} \qquad \text{(EqInt)}$$

$$\frac{\mathsf{expr}_1 \;:\; \tau_1 \qquad \mathsf{expr}_2 \;:\; \tau_2}{\begin{array}{l} \mathsf{expr}_1 + \mathsf{expr}_2 \;:\; max(\{\tau_1, \tau_2, \mathit{IntAddEqBool}\}) \\ \mathsf{expr}_1 - \mathsf{expr}_2 \;:\; max(\{\tau_1, \tau_2, \mathit{IntAddEqBool}\}) \end{array}} \qquad \text{(Add)}$$

$$\frac{\mathsf{expr}_1 \;:\; \tau_1 \qquad \mathsf{expr}_2 \;:\; \tau_2}{\begin{array}{l} \mathsf{expr}_1 * \mathsf{expr}_2 \;:\; max(\{\tau_1, \tau_2, \mathit{IntAll}\}) \\ \mathsf{expr}_1 \,/\, \mathsf{expr}_2 \;:\; max(\{\tau_1, \tau_2, \mathit{IntAll}\}) \end{array}} \qquad \text{(Mult)}$$

Description

Predicate $uses(\mathsf{op}, \mathsf{x})$ states that a program operation op references a variable x; function $max(\{\tau_1, \ldots, \tau_n\})$ returns the maximal type for our defined set of types and the following (transitiv) type relation: $\mathit{IntBool} < \mathit{IntEqBool} < \mathit{IntAddEqBool} < \mathit{IntAll}$; a type constraint obj : τ states that the type of obj is equal or greater than τ, where obj can be either an expression, a program operation, or a variable; note that this first proposal for typing rules is very coarse and can be significantly refined, e.g., by eliminating the closure.

Fig. 6. Syntax definition and domain-type rules; a program is represented as control-flow automaton (CFA) [10], where nodes represent control-flow locations and edges represent program operations that are executed when control flows from one control-flow location to the next; CPACHECKER supports C, we use this largely abbreviated and adjusted grammar of program operations to simplify the presentation.

are interpreted as true. Let us consider the code in Fig. 5: The expression enabled in the if condition is internally expanded to the expression enabled != 0 [2]. As described in Sect. 2, such a variable should be represented in a BDD by one boolean variable, not by 32 boolean variables. Therefore, we introduce a domain type *IntBool* that represents this more precise type. To determine whether an integer variable has actually the domain type *IntBool*, our pre-analysis inspects all occurrences of the variable in the C expressions. If a variable is found to be of domain type *IntBool*, this fact can be considered during the assignment of the abstract domain, and thus the variable can be represented by data structures that efficiently store boolean values during the verification. Fig. 4 shows the four domain types that we consider in the static pre-analysis (more domain types are of course possible, but not yet evaluated). The pre-analysis assigns every program variable to one of these domain types, from which an appropriate abstract domain can be derived.

Other programming languages (e.g., JAVA) provide more restrictive types than C does, such as boolean and byte, but for the purpose of assigning the best abstract domain, even more precise information is beneficial. In dynamically-typed or even untyped languages, types of variables are unknown before program execution. A static analysis of domain types can lead to considerable improvements of the verification process, because it can infer more specific domain types, and thus, choose more efficient algorithms and data structures for representing abstract states.

3.2 Pre-analysis

In the first step, a static pre-analysis computes the domain type for each program variable, according to the type system in Fig. 6. For each program operation (either ASSUME or ASSIGNMENT), the analysis determines the maximal domain type that is needed according to the expression operators that occur in the program operation. Then, it constructs the type closure over all program operations that use some common variables, to determine the maximal domain type that the program operations for a program variable require. The type of a variable x is the (maximal) domain type of program operations that use variable x. For example, the program operations x == 0, x == x + 1, and y == x * (z + x) are of the domain types *IntBool*, *IntAddEqBool*, and *IntAll*, respectively. If all program operations occur in the program, the closure includes all of them (because all use variable x), and thus the domain type of x, y, and z is *IntAll*.

The domain type of an expression is *IntBool* if all operators in the expression are negations (!) or comparisons with zero (== 0 and != 0). If an expression also contains equality tests with non-zero constant values or other variables (==, !=), then the domain type of the expression is *IntEqBool*. If an expression, in addition, contains linear arithmetic (+, −), arbitrary comparisons (==, !=, <, >, <=, >=), or bit operators (&, |, ^), then the domain type is *IntAddEqBool* [2]. Expressions that contain any other operators (e.g., multiplication, division) are of the most general domain type *IntAll*.

The four domain types are in subtype relation, as illustrated in Fig. 4. Each variable that is of type *IntBool* is also of the domain types *IntEqBool*, *IntAddEqBool*, and *IntAll*. The type system assigns the strongest (most restrictive, least) possible type that satisfies the type rules (i.e., the type system assigns domain type *IntBool* instead of *IntAddEqBool* if possible). To be able to refer to variables that are of a certain domain type and *not* of the corresponding weaker domain type (e.g., variables that are in *IntAddEqBool* and not in *IntEqBool*), we introduce four new domain types, for brevity:

$$\begin{aligned} \mathit{Bool} &= \mathit{IntBool} \\ \mathit{Eq} &= \mathit{IntEqBool} \setminus \mathit{IntBool} \\ \mathit{Add} &= \mathit{IntAddEqBool} \setminus \mathit{IntEqBool} \\ \mathit{Other} &= \mathit{IntAll} \setminus \mathit{IntAddEqBool} \end{aligned}$$

3.3 Domain Assignment

Once the domain type has been determined for each program variable, each domain type is assigned to a certain abstract domain that the analysis uses to track the variables

[2] The operators <, >, <=, >=, <<, >>, &, |, and ^ are omitted in the type rules in Fig. 6 for brevity.

of that domain type. Therefore, we define a *domain assignment* d to be a map that assigns an abstract domain to each domain type. To setup the program analysis, we add all variables of a domain type t to the abstraction precision of the abstract domain $d(t)$. In principle, every abstract domain can represent any variable, but each abstract domain has certain strengths and weaknesses. A perfect domain assignment would map each domain type to the abstract domain that is most appropriate for representing values of the variables.

It seems straightforward to assign the BDD domain to domain type $Bool$. The BDD domain can efficiently represent complex boolean combinations of variables, but is sensitive to the number of represented variables. We can also assign the BDD domain to the domain types Eq and Add. For domain type Eq, we know from the properties of the domain type that those variables only hold a limited and static set of values. Therefore, we can enumerate these values and represent them by $\log_2(n)$ BDD variables, where n is the number of values. The explicit-value domain can in principle be used for all domain types, but the more different combinations of variable assignments need to be distinguished in the analysis, the larger the state space grows, perhaps resulting in an out-of-memory exception. Moreover, the explicit-value domain is not appropriate for analyzing uninitialized variables.

In our experiments, we show that different domain assignments have significantly different performance characteristics for different sets of verification tasks. Automatically selecting an optimal domain assignment remains an open research problem. The goal of this paper is to show that the concept of domain types provides a promising technique to approach the problem.

4 Experimental Evaluation

To evaluate the domain-type-based analysis approach, we conduct a series of experiments with different configurations on a diverse set of verification tasks. The results provide evidence that the chosen domain assignment has a significant impact on effectiveness and efficiency. In particular, we address the following issues:

Domain Types. The subject systems contain a sufficient set of integer variables such that a domain-type analysis is able to classify them into more specific domain types.

Variable Partitioning. The verification performance significantly changes if variables are represented by different abstract domains, compared to representing all variables with the same abstract domain.

Advantage of Combinations. Using the BDD domain for some variables (e.g., all variables of the domain types $Bool$ and Eq) and the explicit-value domain for other variables can improve the verification performance.

4.1 Implementation

For our experiments, we extended the open verification framework CPACHECKER [13], which provides various abstract domains and supports the concept of abstraction precisions in a modular way, such that it is easy to extend and configure. The tool is applicable to an extensive set of verification benchmarks, because it participated in the competition on software verification. This makes it possible to evaluate our approach on a large set of representative programs.

Explicit-Value Domain. We use the default explicit-value domain that is already implemented in CPACHECKER [14]. It uses a hash-map to associate variables with values. This implementation is efficient in handling variables with few different values that are used in complex operations.

BDD Domain. We extended CPACHECKER's BDD domain [15] to use —depending on the domain type— specialized encodings of variables in the BDD. For domain type $Bool$, we use exactly one BDD variable per program variable. For variables of domain type Add, we use 32 BDD variables to represent one program variable (we omit the details of bit-precise analysis). For variables of domain type Eq, we know from the pre-analysis how many different values the variable can hold. Therefore, we can re-map the values to a new set of values with the same cardinality (nominal scale [37]), which needs considerably fewer BDD variables (compared to 32 BDD variables). We use a simple bijective map from the original constants in the program to a (smaller, successive) set of integer values encoded with BDD variables. We also encode information about equality of uninitialized Eq variables (for example, in the expression x==y). To achieve this, we reserve a value in the encoding for each of the Eq variables. In total, we use $log_2(n+m)$ BDD variables per Eq program variable, where n is the number of program constants and m is the number of Eq variables.

4.2 Experimental Setup

We performed all experiments on a Ubuntu 12.04 (64-bit) system (LINUX 3.2 as kernel and OpenJDK 1.7 as JAVA VM) with a 3.4 GHz Quad Core processor (Intel Core i7-2600). Each verification run was limited to 2 cores, 15 GB of memory, and 15 min of CPU time. We used the version of CPACHECKER that is available as revision tag cpachecker-1.2.7-hvc13. Each verification task was verified using five different configurations:

Explicit: This configuration tracks all variables with the explicit-value domain.

BDD-IntBool: This configuration uses both abstract domains [3]; all variables of domain type *IntBool* are in the precision of the BDD domain and all other variables are in the precision of the explicit-value analysis.

BDD-IntEqBool: This configuration uses both abstract domains; all variables of domain type *IntEqBool* are in the precision of the BDD domain and all other variables are in the precision of the explicit-value domain.

BDD-IntAddEqBool: This configuration uses both abstract domains; all variables of domain type *IntAddEqBool* are in the precision of the BDD domain and all other variables are in the precision of the explicit-value domain.

BDD: This configuration tracks all variables with the BDD domain.

[3] We expected that the combined configurations (*BDD-IntBool*, *BDD-IntEqBool*, and *BDD-IntAddEqBool*) would suffer from the overhead of running two abstract domains. We measured this overhead in separate experiments (running one of the domains with empty precision) and found that the impact is negligible.

4.3 Verification Tasks

We evaluate our approach on six benchmark sets that, in total, consist of 2 435 verification tasks. The benchmark sets are (number of verification tasks in parentheses):

Control Flow and Integer Variables (94)
Device Drivers Linux 64-bit (1 237)
ECA (366)
Loops (79)
SystemC (62)
Product Lines (597)

All verification tasks of the benchmark sets have been used in international competitions of software-verification tools [9, 30]; they are publicly available via the competition repository or the CPAchecker repository[4]. The SV-COMP benchmark suite is the most comprehensive and diverse suite of this kind that currently exists. It covers various application domains, such as device drivers, software product lines, and event-condition-action-systems simulation.

The following description of the systems is partly taken from the report on the first competition on software verification [8]. Unless stated otherwise, the systems are taken from the 2013 edition of the competition. The set Control Flow and Integer Variables contains, among others, verification tasks that are based on device drivers from the Windows NT kernel and verification tasks that represent the connection-handshake protocol between SSH server and clients with protocol-specific specifications. The set Device Drivers Linux 64-bit contains verification tasks that are based on device drivers from the Linux kernel. The verification tasks in the set SystemC are provided by the SyCMC project [21] and were taken (with some changes) from the SystemC distribution. The benchmark set ECA contains event-condition-action (ECA) programs, a kind of systems that is often used in sensor-actor systems. The verification tasks in our benchmark set have been used in the RERS Grey-Box Challenge 2012 [30] on verifying ECA systems. The Loops benchmark set consists of verification tasks that require the analysis of loops with non-static loop bounds. The benchmark set Product Lines models three software product lines used in feature-interaction detection [5].

Domain Types. To evaluate whether we can assign a non-trivial set of variables to specific domain types, we measured how many variables could be classified as *Bool*, *Eq* or *Add* per benchmark set. We were able to classify as *Bool*, *Eq* or *Add*, on average, 60 % for Control Flow and Integer Variables, 26 % for Device Drivers Linux 64-bit, 64 % for Loops, 52 % for Product Lines, 99 % for SystemC, and 100 % for ECA of all program variables. This confirms that there is always a set of variables that have potential for improvement by alternative domain assignments. In most benchmark sets, the domain type with the largest number of variables is *Eq*. We expect that optimizations for the domain type *Eq* pay off, especially, in the benchmark sets ECA and SystemC, because this domain type covers a large part of the variables in these sets. The benchmark set SystemC also has a high number of *Add* variables in a significant number of verification tasks, so we expect a performance difference for the different domain assignments especially for this domain type.

[4] `http://cpachecker.sosy-lab.org/`

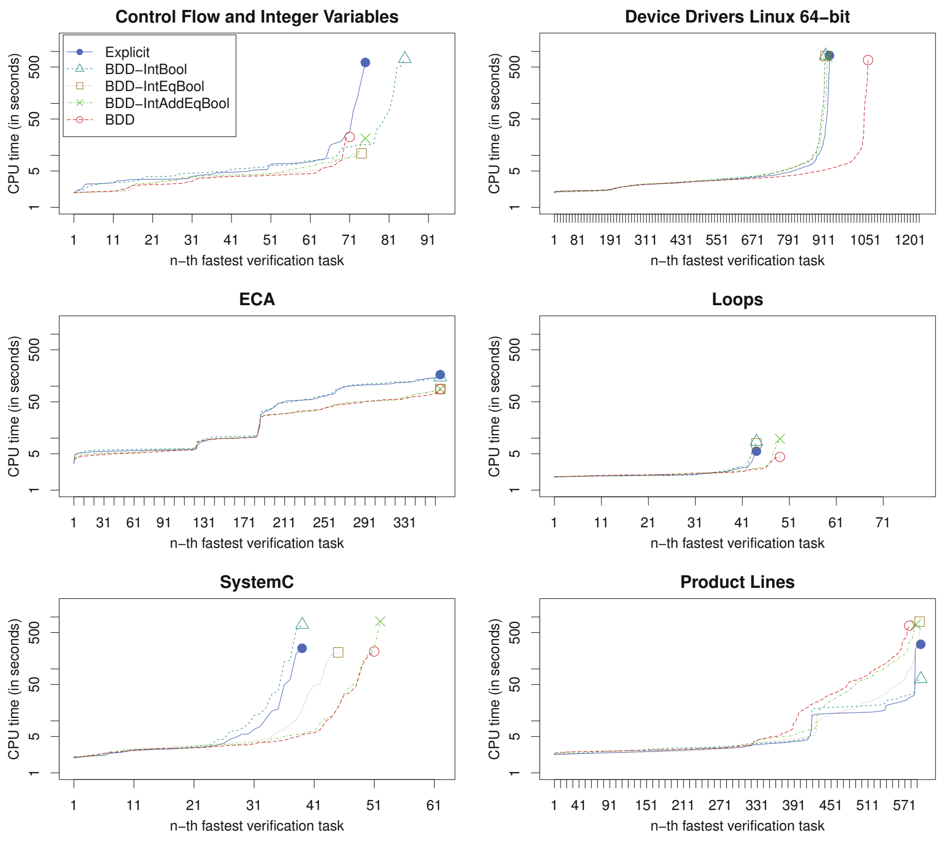

Fig. 7. The quantile plots show the performance of different configurations; each picture represents the data for one benchmark set; each data point (x, y) shows the x-th fastest verification run that needed y seconds of CPU time; the y-axes use logarithmic scales

4.4 Results

Due to the huge amount of verification results, we cannot provide the raw data of all verification runs. Instead, we discuss results aggregated by categories and configurations in Fig. 7. The diagrams show the performance of the configurations (*Explicit*, *BDD-IntBool*, *BDD-IntEqBool*, *BDD-IntAddEqBool*, and *BDD*) in quantile plots for each benchmark set. A point (x, y) in a quantile plot states that the x-th fastest verification run of the respective configuration took y seconds of CPU time. The right-most x value of a configuration indicates the total number of correctly solved verification tasks. The area below the graph is proportional to the accumulated verification time. We also provide a supplementary web page[5], where the detailed results of all verification runs (including the raw data and the log files) are available for download and as interactive plots.

[5] `http://www.sosy-lab.org/projects/domaintypes/`

Effectiveness. Figure 7 witnesses that many tasks are difficult to verify. For example, in the benchmark set LOOPS, most configurations solve only about half of the tasks correctly. Failures are caused by timeouts, out-of-memory exceptions, or limitations of the implemented abstract domains. The combined configurations often demonstrate good effectiveness results. In several benchmark sets, the configuration *BDD-IntBool* is among the configurations that can verify most files correctly (have one of the highest x values). However, there is no clear winner in terms of effectiveness, which suggests to further investigate verification based on domain types. The first plot (CONTROL FLOW AND INTEGER VARIABLES) demonstrates that using combinations of abstract domains allows solving verification tasks that are not solvable by one abstract domain alone.

Efficiency. The benchmark set CONTROL FLOW AND INTEGER VARIABLES covers a diverse set of verification tasks. Among others, it contains drivers of the WINDOWS NT kernel and SSH benchmarks. The plot (Fig. 7) shows that the configurations *BDD-IntEqBool* and *BDD-IntAddEqBool* are fast on many of the files, and that configuration *BDD-IntBool* can solve more tasks than any other configuration. This result can be explained by investigating the number of variables per domain type: the verification tasks in this category have many variables of domain types that can be efficiently handled in the BDD domain (*Bool*, *Eq*, *Add*). A certain set of verification tasks can only be solved using the configuration *BDD-IntBool*. These verification tasks illustrate a situation where two variables of types Eq and $Other$ interact in a special pattern. The variables must be handled by the same domain to verify the file. Only the configurations *Explicit* and *BDD-IntBool* track both variables in the explicit domain and compute a correct verification result. Configuration *Explicit* fails on other tasks in this set, such that its effect on these tasks cannot be seen easily in the plot.

On the benchmark set DEVICE DRIVERS LINUX 64-BIT, all configurations, except the *BDD* configuration, show identical performance. Configuration *BDD* performs so well because some of the $Other$ variables, which are ignored in configuration *BDD*, do not have an effect on the verification result. It would be interesting to combine our approach with CEGAR [23] (where such variables would be ignored in all configurations). The combination configurations perform similarly because only 26 % of all variables have been classified as $IntAddEqBool$, and therefore these tasks do not have much potential for the domain-type optimization.

For the benchmark set ECA, the configurations that encode Eq variables in BDDs are most efficient. All variables in the ECA verification tasks are of domain type Eq, and therefore the configurations that represent Eq variables with the BDD domain are performing best (*BDD-IntEqBool*, *BDD-IntAddEqBool*, and *BDD*). This indicates that tracking Eq variables with BDDs can be beneficial. The configurations *Explicit* and *BDD-IntBool* perform worse, because they represent the variables of domain type Eq using the explicit-value domain. The performance result is in line with the results of a recent paper on BDD-based software model checking [15].

In the benchmark set LOOPS, the *BDD-IntAddEqBool* and *BDD* configurations can solve a specific group of tasks that the other configurations can not solve. These tasks model a token-ring architecture with a varying number of nodes. The verification tasks each contain pairs of *Add* variables that are difficult to track with the explicit-value domain, because they are not initialized at program start. One of the variables is assigned to the other, then both are incremented (which makes them *Add*), and then the values are compared again. This unique usage profile requires to represent these variables in the BDD domain, which explains the results.

The benchmark set SYSTEMC shows that the configurations *BDD-IntAddEqBool* and *BDD-IntEqBool* can verify a considerable number of tasks more than the other combination configuration and configuration *Explicit*. This is easy to understand: the tasks contain many *IntEqBool* (avg. 93 %) and *IntAddEqBool* (avg. 99 %) variables. This result shows that it can be extremely efficient to track such variables with BDDs. The good performance of configuration *BDD* shows that the non-*IntAddEqBool* variables can be ignored during verification.

The configuration *BDD-IntBool* performs well on the verification tasks in benchmark set PRODUCT LINES. The benchmark set has been used for research projects on product-line verification [4, 5], from which we know that these files contain many variables of type *Bool* and *Eq*. Some of the files that are most difficult to verify contain *Bool* variables that guide the control flow and are critical for the verification process. Therefore, it is no surprise that the *BDD-IntBool* configuration performs best on these tasks.

4.5 Discussion

Our experimental study has shown that the performance of the combined configurations (*BDD-IntBool*, *BDD-IntEqBool*, and *BDD-IntAddEqBool*) depends heavily on the domain types of the variables in the program. If the verification tasks contain variables of domain type *IntAddEqBool*, then representing these variables with the BDD domain can significantly improve the performance.

The experiments have also shown that configuration *BDD* exhibits a good performance on many verification tasks, even though it cannot track variables of domain type *Other*. This means that variables of domain type *Other* are ignored during verification, and still the verification result is correct. But, in the interest of soundness and reliable results, we are more interested in configurations without obvious ‘blind spots’.

Let us briefly re-visit —based on the experimental results— the issues that we listed at the beginning of the section. The first issue concerning the domain types has already been discussed (Sect. 4.3). Concerning the variable–domain mapping, our experiments confirm that analyzing variables of different domain types with different abstract domains can make a huge difference, in terms of effectiveness and efficiency. Combined configurations sometimes outperform the single-domain configurations (only explicit-value domain or only BDD domain) on several benchmark sets. The configuration *BDD* performs well on most benchmark sets, in particular on the DEVICE DRIVERS LINUX 64-BIT tasks. However, it is apparent that including the support of the explicit analysis for *Other* variables is critical to obtain reliable verification results. Overall, it might be beneficial to use the BDD domain for variables of domain type *IntAddEqBool*, and the explicit-value domain for the *Others*. This is confirmed by the performance of configurations *BDD-IntEqBool* and *BDD-IntAddEqBool*.

5 Related Work

We infer domain types for program variables according to their usage in program operations. This principle is also used by the type- and memory-safety analysis of C programs with *liquid types* [33]. There, a static program analysis is used to determine, for each variable, a predicate that restricts the possible values of the variable (the *liquid type*). In a second step, each usage of the variable is checked for type safety, or if it could lead to an unsafe memory access. In contrast to domain types, *liquid types* use a predicate for each variable. *Liquid types* are fine-grained, domain types are coarse-grained in comparison, but the granularity is flexible in both approaches. Our type checker for domain types does not depend on an SMT solver, which is an advantage in terms of computational complexity.

Roles of variables are used to analyze programs submitted by students [16]. Program slicing and data-flow analysis is applied to determine the role of each variable (e.g., *constant* or *loop index*). The role is then compared to the role that the students have assigned to the variables. Variable roles are also used to understand COBOL programs [38, 39], to understand novice-level programs [35], and to classify programs into categories [25]. These works on variable roles fall into the area of automated program comprehension. The rather strong behavioral variable types might be interesting to extend our work.

JAVA PATHFINDER [40] has an extension that combines the standard explicit analysis with a BDD-based analysis for boolean variables [5, 32]. In that approach, the variables that are to be tracked by BDDs were manually selected, based on domain knowledge. Our new approach handles a broader set of domain types and categorizes them automatically.

BEBOP [6], a model checker for boolean programs, encodes all program variables (only booleans, in this case) in BDDs, and uses explicit-state exploration for the program counter. Our domain-type analysis would correctly classify all variables as *Bool* and encode them with BDDs; thus, we subsume this approach. A similar strategy was followed by others [22].

A hybrid approach combining explicit and BDD-based representations analyzes the program variables with BDDs and the states of the property automaton explicitly [36]. In our setting, this translates to encoding all program variables in BDDs, because the property automaton runs separately and explicitly in parallel in CPACHECKER. This case can be represented in our general framework as configuration *BDD*.

The two symbolic domains BDDs and Presburger formulas have been previously used as representation for boolean and integer variables [19]. The approach was evaluated on two systems, a control software for a nuclear reactor's cooling system and a simplified transport-protocol specification. In contrast to our work, this work is not based on a separate analysis to determine domain types of variables, but includes the type analysis in the actual model-checking process. By performing the domain-type analysis in advance, we avoid overhead during the model-checking process.

6 Conclusion

We introduced the concept of *domain types*, which makes it possible to assign variables to certain abstract domains based on their usage in program operations. We define a static pre-analysis that maps each variable of type 'integer' to one of four more specific domain types, which reflect the usage of variables in the program.

We performed many experiments with two abstract domains, to demonstrate that the domain assignment based on domain types has a significant impact on the effectiveness and efficiency of the verification process. We considered five domain assignments: one for each considered abstract domain that tracks all program variables in one single abstract domain, without considering the different domain types, and three with different assignments of the variables to the two abstract domains according to the domain type.

A key insight is that the concept of domain types is a simple yet powerful technique to create verification tools that implement a better choice for the domain assignment. State-of-the-art is to use either one single abstract domain, or a fixed combination of abstract domains that adjust precisions via CEGAR or otherwise dynamically, during the verification run. Our benchmark set contains a significant number of variables for which we can determine different, narrower domain types. The domain type *IntEqBool* (and even more its subtype *IntBool*) dramatically decreases the size of the internal BDD representation of the variable assignments, and thus can lead to a significant improvement in verification efficiency. Overall, our experiments show that performance can be improved substantially if the variables are tracked in an abstract domain that is suitable for the domain type of the variable. Not only the performance is improved: combinations of abstract domains make it possible to solve verification problems that are not solvable using one abstract domain alone.

Acknowledgements. S. Apel and A. von Rhein have been supported by the DFG grants AP 206/2, AP 206/4, and AP 206/5.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
2. American National Standards Institute. ANSI/ISO/ IEC 9899-1999: Programming Languages — C. American National Standards Institute, 1430 Broadway, New York, USA (1999)
3. Apel, S., Beyer, D., Friedberger, K., Raimondi, F., von Rhein, A.: Domain types: Selecting abstractions based on variable usage. Technical Report MIP-1303, University of Passau (2013), `http://arxiv.org/abs/1305.6640`
4. Apel, S., Speidel, H., Wendler, P., von Rhein, A., Beyer, D.: Detection of feature interactions using feature-aware verification. In: Proc. ASE, pp. 372–375. IEEE (2011)
5. Apel, S., von Rhein, A., Wendler, P., Größlinger, A.: Strategies for product-line verification: Case studies and experiments. In: Proc. ICSE, pp. 482–491. IEEE (2013)
6. Ball, T., Rajamani, S.: Bebop: A symbolic model checker for boolean programs. In: Proc. SPIN, pp. 113–130 (2000)
7. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: Proc. POPL, pp. 1–3. ACM (2002)
8. Beyer, D.: Competition on software verification (SV-COMP). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012)
9. Beyer, D.: Second competition on software verification. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 594–609. Springer, Heidelberg (2013)

10. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007)
11. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE, pp. 29–38. IEEE (2008)
12. Beyer, D., Henzinger, T.A., Théoduloz, G., Zufferey, D.: Shape refinement through explicit heap analysis. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 263–277. Springer, Heidelberg (2010)
13. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
14. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Cortellessa, V., Varró, D. (eds.) FASE 2013 (ETAPS 2013). LNCS, vol. 7793, pp. 146–162. Springer, Heidelberg (2013)
15. Beyer, D., Stahlbauer, A.: BDD-Based Software Model Checking with CPACHECKER. In: Kučera, A., Henzinger, T.A., Nešetřil, J., Vojnar, T., Antoš, D. (eds.) MEMICS 2012. LNCS, vol. 7721, pp. 1–11. Springer, Heidelberg (2013)
16. Bishop, C., Johnson, C.G.: Assessing roles of variables by program analysis. In: Proc. CSEIT, pp. 131–136. TUCS (2005)
17. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. PLDI, pp. 196–207. ACM (2003)
18. Bryant, R.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys 24(3), 293–318 (1992)
19. Bultan, T., Gerber, R., League, C.: Composite model-checking: Verification with type-specific symbolic representations. ACM TOSEM 9(1), 3–50 (2000)
20. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Proc. LICS, pp. 428–439. IEEE (1990)
21. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: A software model checking approach. In: Proc. FMCAD, pp. 51–59. IEEE (2010)
22. Cimatti, A., Roveri, M., Bertoli, P.G.: Searching powerset automata by combining explicit-state and symbolic model checking. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 313–327. Springer, Heidelberg (2001)
23. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
24. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A.: Symbolic model checking of software product lines. In: Proc. ICSE, pp. 321–330. ACM (2011)
25. Demyanova, Y., Veith, H., Zuleger, F.: On the concept of variable roles and its use in software analysis. Technical Report abs/1305.6745, ArXiv (2013)
26. Dudka, K., Müller, P., Peringer, P., Vojnar, T.: Predator: A verification tool for programs with dynamic linked data structures. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 545–548. Springer, Heidelberg (2012)
27. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
28. Havelund, K., Pressburger, T.: Model checking Java programs using Java PATHFINDER. Int. J. Softw. Tools Technol. Transfer 2(4), 366–381 (2000)
29. Holzmann, G.J.: The SPIN model checker. IEEE Trans. Softw. Eng. 23(5), 279–295 (1997)
30. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D.: The RERS grey-box challenge 2012: Analysis of event-condition-action systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 608–614. Springer, Heidelberg (2012)
31. McMillan, K.L.: The SMV system. Technical Report CMU-CS-92-131, CMU (1992)

32. von Rhein, A., Apel, S., Raimondi, F.: Introducing binary decision diagrams in the explicit-state verification of Java code. In: JavaPathfinder Workshop (2011), `http://www.infosun.fim.uni-passau.de/cl/publications/docs/JPF2011.pdf`
33. Rondon, P., Bakst, A., Kawaguchi, M., Jhala, R.: CSolve: Verifying C with liquid types. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 744–750. Springer, Heidelberg (2012)
34. Sagiv, M., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM TOPLAS 24(3), 217–298 (2002)
35. Sajaniemi, J.: An empirical analysis of roles of variables in novice-level procedural programs. In: Proc. HCC, pp. 37–39. IEEE (2002)
36. Sebastiani, R., Tonetta, S., Vardi, M.Y.: Symbolic systems, explicit properties: On hybrid approaches for LTL symbolic model checking. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 350–363. Springer, Heidelberg (2005)
37. Stevens, S.S.: On the theory of scales of measurement. Science 103(2684), 677–680 (1946)
38. van Deursen, A., Moonen, L.: Type inference for COBOL systems. In: Proc. WCRE, pp. 220–230. IEEE (1998)
39. van Deursen, A., Moonen, L.: Understanding COBOL systems using inferred types. In: Proc. IWPC, pp. 74–81. IEEE (1999)
40. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. J. ASE 10(2), 203–232 (2003)