DLINE JOURNALS

Progress in Computing Applications

Print ISSN: 2278 – 6465
Online ISSN: 2278 – 6473

PCA 2024; 13 (1)
https://doi.org/10.6025/pca/2024/13/1/1-6

Abstraction Techniques from Shape Analysis and Program Analysis

Bjorn Wachter
Saarland University, Im Stadtwald
Saarbrucken, Germany
{bwachter@cs.uni-sb.de}

Received: 2 October 2023
Revised: 29 December 2023
Accepted: 19 January 2024
Copyright: with Author(s)

ABSTRACT

Automatic formal validation of systems with a large or infinite number of components is challenging due to prohibitively large state space. Abstraction techniques automatically build finite approximations of an infinite-state system from which safety information about the original system can be derived. This paper explores two abstractions: shape analysis, a technique derived from program analysis. Data type reduction is a technique originating from model checking. Until recently, we did not fully understand how shape analysis relates to data type reduction. In this paper, we explain this relationship comprehensively.

Keywords: *Shape Analysis, Program Analysis, Abstraction Techniques, Data Type Reduction*

1. Introduction

We consider analysis techniques for parameterized systems such as protocols where the number of participating processes is a parameter These models are composed of processes that run in a parallel, interleaved fashion. The state of the model consists of the local states of all constituent processes. Typically one wants to verify first-order temporal properties, i.e. safety properties such as mutual exclusion and liveness properties such as lack of starvation.

Finitary abstraction techniques generate a finite state model that approximates the original infinite state model preserving certain properties. A finitary abstraction technique has typically two constituents (1) *a state abstraction function that maps states of the original model to states of the abstract model and* (2) *a method to compute transitions between abstract states, i.e. the behavior of the abstract model. The finite state model is subject to reachability analysis or to a finite-state model checker. Several finitary abstractions have been proposed, such as counter abstraction* [*PXZ*], *canonical abstraction* [*SRW02, Yah01*] *and data type reduction* [*McM00, DW03*].

In previous work [***Wac05***], *we have studied a model checking framework for*

parameterized systems based on canonical abstraction that lends ideas from data type reduction. Notably, data type reduction can be expressed in the same framework which is the topic of this work.

1.1. State Abstractions

Predicate abstraction: *Predicate abstraction approximates the state of a program by a tuple of Boolean values that record if certain properties hold or not. For example, instead of storing an integer variable x one only keeps track of whether or not $x > 0$ holds. Predicate abstraction has been successfully applied to sequential programs.*

Running Example: *To demonstrate the abstractions, we consider as an example a parameterized system in which each process p has a program counter $PC(p)$ giving the process' current control location; a control location is a member of the set* {a, b, c, d}*. The example state consists of 9 processes.*

Counter Abstraction: *Counter abstraction* $[PXZ]$ *assumes that processes are finite state, i.e. there exists a finite set* Σ *of local states. For each local state* $\sigma \in \Sigma$ *, a counter variable* C_σ *is used that records the number of processes currently in state . To obtain a finite abstract domain the counters are typically cut off at two. An abstract state is a mapping* $C : \Sigma \to \{0, 1, \geq 2\}$*. As the size of the abstract state space is exponential in the size of* Σ*, counter abstraction falls short of infinite or very large local state spaces.*

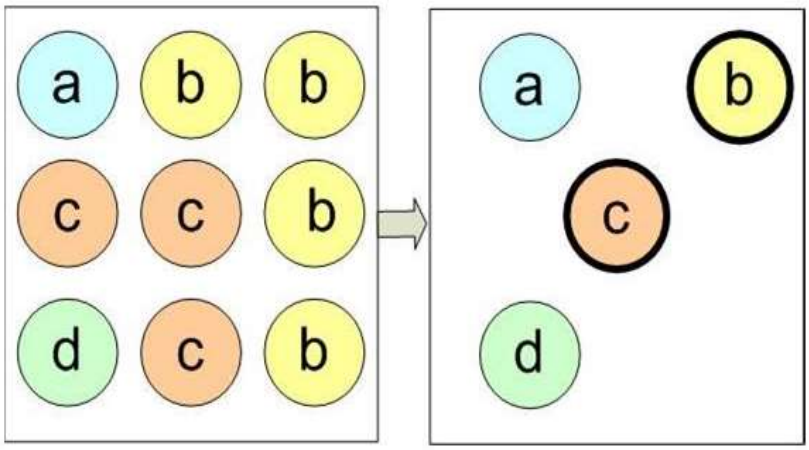

Figure 1. Counter abstraction

Figure 1 *shows, left to the arrow, a concrete state with* 9 *processes that is abstracted to the abstract state right of the arrow. The circles denote concrete processes and the letters in the circles the value of the program counter. Note that in this example the set of local states is* $\Sigma = \{a, b, c, d\}$*. The abstract state has four counters one for each element of* Σ*. We think of the non-zero counters as abstract processes, as they stand for concrete processes. We symbolize each abstract process by a circle with a thin border if the counter is one, and by a circle with a thick border if the counter has at least value* 2*.*

Canonical abstraction: *As opposed to counter abstraction, canonical abstraction is applicable to systems where the local state space is infinite. Intuitively, canonical abstraction first abstracts local state per process, then processes with the same abstract local state are collapsed to one abstract process similar to counter abstraction. Local state is abstracted to a vector in which each position encodes the truth of a predicate ranging over processes. Predicates have defining formulas that may refer to local and global state, informally stated predicates can refer to the environment of a process.*

Canonical abstraction admit predicates ranging over pairs of processes. For the sake of brevity, we omit these aspects of canonical abstraction for now.

Returning to the running example, we define two predicate: one predicate $at_a(p)$ *is true of a process if it is in control location a,* $at_a(p) \equiv PC(p) = a$*, the other predicate atb holds for a process that is is control location* b*,* $at_b \equiv PC(p) = b$*. Figure* 2 *depicts the concrete state and its canonical abstraction. Abstract processes are two-component boolean vectors where the first components stands for the truth of predicate* at_a *and the second component for* at_b*. The process in location a is mapped to the abstract process* (1, 0)*, the one of the processes in location* b *is* (0, 1) *all other processes have* (0, 0)*.*

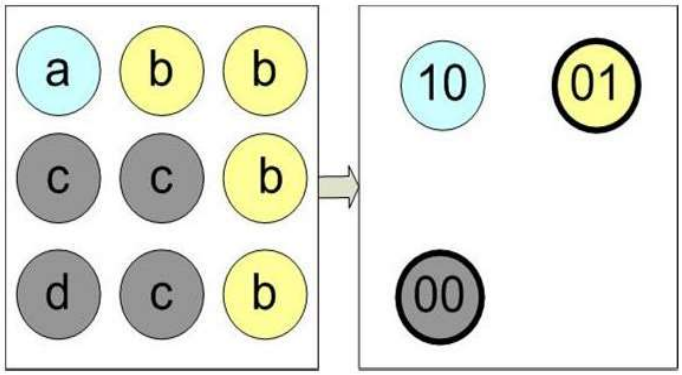

Figure 2. Canonical abstraction

1.2. The Migration Problem

The previously described abstractions are sufficient to verify and infer invariants, yet, let alone, too coarse to verify first-order properties. For example, they would not allow us to check if every process will eventually reach location b*. Consider the process in Figure* 2 *that is at control location* a*. Let us assume it moves on to location* b*. In an abstract successor state, our process would become part of the abstract process consisting of all processes being at location* b*. The example shows that in two states that each have an instance of an abstract process like* (0, 1) *these two instances may correspond to different collections of concrete processes. This is depicted in Figure* 3*. The problem is caused by canonical abstraction and counter abstraction collapsing processes to abstract processes. By a state change, a process migrates between instances of abstract processes. Abstraction takes away process identities and thus the means to track evolution of processes across transitions.*

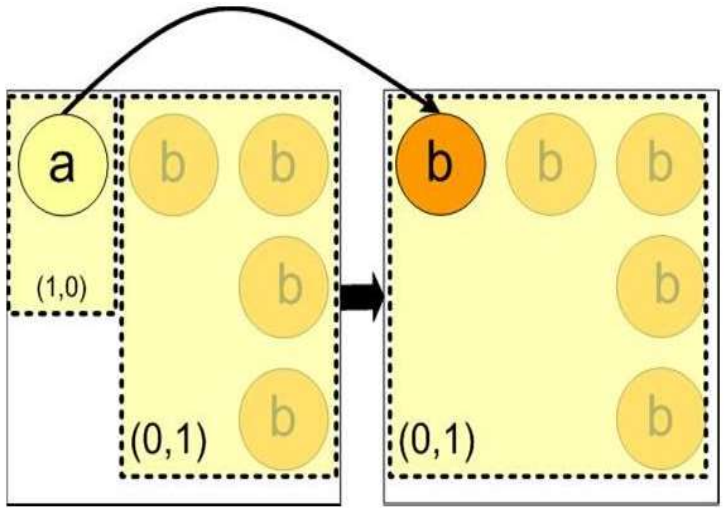

Figure 3. Process migrating between abstract processes

1.3. Abstraction for First-Order Properties

One solution to the migration problem is to reduce the first-order model checking problem to an equivalent problem in which explicit tracking of process evolution is not necessary anymore. The semantics of universal quantification is usually given inductively in terms of the semantics of the subformula without the outermost quantifier. One combines the results of evaluating that subformula under all the different possible values the quantification variable can take on. In the domain of parameterized systems, that leaves us with an infinite number of cases to check. By applying abstraction, the infinite number of cases can be reduced to a finite, tractable number of cases.

Each subproblem only requires one to show the property for a distinguished process rater than for all processes. The abstraction can be adapted such that it is centered around the distinguished process, in that it retains more information pertaining to the distinguished process and abstracts the other processes more coarsely, and further precisely models the relation of the other processes to the distinguished process.

Note that properties which involve multiple quantifiers, like mutual exclusion, can be shown in the same way. Then there is a number of distinguished processes rather than just a single distinguished process.

Variations of this idea of decomposition are present in data type reduction, and in the shape analysis for JDBC in Ramalingam et al. [YR04].

Data type reduction: *Data type reduction relies on a separation of processes into two classes : a fixed number of distinguished processes and all other processes, let us call the other processes environment processes. Data type reduction retains the distinguished processes and abstracts all environment processes into one summary abstract process. The summary abstract process mimics the behavior of all the processes it represents, it is non-deterministic and memoryless, i.e. the analysis does not compute information concerning environment processes.*

Figure 4 *shows the data type reduction of the state from the running example. The reference process is colored black. It retains its local state a. All other processes, the environment processes, are abstracted to one abstract summary process. The local state of the summary process is abstracted away as indicated by the question tag.*

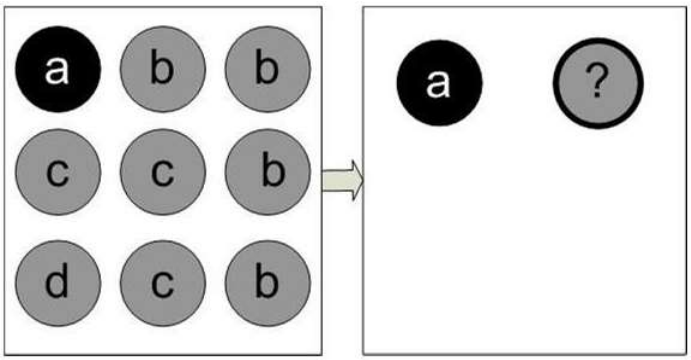

Figure 4. Data type reduction

1.4. Results

[SRW02] *characterizes canonical abstraction in the framework of three-valued logic analysis underlying shape analysis. Abstract states are compared by a partial order, named embedding. A state being embedded in another state implies that information derived from the state that is larger in the order also holds for the smaller state. A state is always embedded in its canonical abstraction. Canonical abstraction is an abstraction which retains the optimal amount of information in the abstract. Formally, it is a tight embedding. Data type reduction is coarser. A state can be embedded into its data type reduction, however, all information about the environment process is*

lost, and therefore it is not a tight embedding.

A more detailed treatment of the topic can be found in [Wac05] *which is also available in the proceedings and on my website* http://rw4.cs.uni-sb.de/~bwachter/thesis.pdf

2. Related Work

Originally, canonical abstraction was designed as an abstraction technique to infer invariants of heap-manipulating programs by a technique called Three-valued Logical Analysis [SRW02] *, vulgo shape analysis. The innovation of canonical abstraction for shape analysis was the generic summarization of objects, where objects were originally thought of as heap cells, and means to compute precise points-to information between abstract heap cells. This precision allows it to automatically prove partial correctness of heap-manipulating programs.*

In [MYRS05]*, a comparison of canonical abstraction and predicate abstraction in the domain of list-manipulating programs is given. They pointed out that in principle every finitary abstraction can be expressed with predicate abstraction. However, the number of predicates needed for the encoding can be prohibitively high so that specialized abstractions can be better.*

Yahav discovered [Yah01] *that the algorithms from shape analysis can be generalized to parameterized protocols and Java programs. First, an abstract finite-state transition system is produced that simulates the (infinite) transition system induced by the original system. Then* LTL *properties are checked on the obtained transition system.*

The obtained transition systems could be used to infer invariants, such as mutual exclusion, however they did not allow checking first-order temporal properties, as it suffers from the Migration Problem described in Section 1.2*. In a subsequent paper, Yahav gave a more powerful method that is able to check properties formulated in a richer logic, termed* ETL [YRSW03]*. The idea was to explicitly store the evolution of processes in state transitions.*

For the sake of higher efficiency and precision, later work aimed at adapting the abstraction to the particular first-order property to be checked. Ramalingam et al. describe a framework for typestate checking for Java programs [YR04]*, i.e. a method for checking invariants. In the context of concurrent systems,* [Wac05] *gave a more general model checking framework for first-order temporal properties of concurrent systems based on canonical abstraction and decomposition.*

Acknowledgments

This work was partly supported by the German Research Council (DFG) *as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems"* (SFB/TR 14 AVACS)*. See* www.avacs.org *for more information.*

References

[1] Clarke, Edmund, Talupur, Muralidhar., Veith, Helmut. (2006). *Environment Abstraction for Parameterized Verification. In VMCAI*, pages 126–141.

[2] Damm, Werner., Westphal, Bernd. (2003). Live and Let Die: LSC-based verification of UML-models. In Formal Methods for Components and Objects, FMCO 2002, volume 2852 of *Lecture Notes in Computer Science*, pages 99–135. Springer.

[3] McMillan, Kenneth L. (2000). *A methodology for hardware verification using compositional model checking. Sci. Comput. Program.*, 37(1-3):279–309.

[4] Manevich, Roman., Yahav, Eran., Ramalingam, G., Sagiv, Mooly. (jan 2005). Predicate abstraction and canonical abstraction for singly-linked lists. In RadhiaCousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation,* VMCAI 2005, Lecture Notes in Computer Science. Springer.

[5] Pnueli, Amir., Xu, Jessie., Zuck, Lenore. Liveness with (0, 1, ?)-counter abstraction.

[6] Sagiv, Mooly., Reps, Thomas., Wilhelm, Reinhard. (2002). Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems.

[7] Wachter, Bjorn. (2005). Checking universally quantified temporal properties with three-valued analysis. Master's thesis, Universit¨at des Saarlandes, March.

[8] Yahav, E. (2001). Verifying safety properties of concurrent Java programs using 3-valued logic. ACM SIGPLAN Notices, 36(3):27–40, March.

[9] Yahav, E., Ramalingam, G. (2004). Verifying safety properties using separation and heterogeneous abstractions. In: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, pages 25–34. ACM Press.

[10] Yahav, E., Reps, Thomas., Sagiv, Mooly., Wilhelm, Reinhard. (2003). Verifying Temporal Heap Properties Specified via Evolution Logic. In European Symposium on Programming, volume 2618 of Lecture Notes in Computer Science, pages 204 – 222. Springer-Verlag.