TOWARDS APPLYING MONA IN ABSTRACT REGULAR TREE MODEL CHECKING

Adam Rogalewicz, Doctoral Degree Programme
Dept. of Intelligent Systems, FIT, BUT
E-mail: rogalew@fit.vutbr.cz

Supervised by: Prof. Milan Češka, Dr. Tomáš Vojnar

ABSTRACT

We consider the problem of automated formal verification of modern concurrent software systems. Dealing with such systems, which involves handling unbounded dynamic instantiation, recursion, etc., naturally leads to a need of dealing with infinite state spaces. We suppose states of such systems to be encoded terms with a tree structure and we would like to use the abstract regular tree model checking method for dealing with infinite sets of states. This paper presents an ongoing research on application of abstract regular model checking in the infinite state systems verification, and possibility to use the Mona GTA library for experiments.

1 INTRODUCTION

The complexity of the modern concurrent software (e.g. control systems, operating systems) is rising, and the requirements on its correctness are also higher and higher. The through testing takes a lot of resources, and in many cases, it can not discover all possible bugs. New approaches, how to prove and guarantee correctness, have to be established. One of such approaches is the formal verification and especially the highly automated approach of model checking. Up to now a lot of work in the model checking of systems with finite state spaces was done. However, many features of the modern software (e.g., recursion, dynamic data structures) imply that the state space is not finite. Verification of such systems is not easy. There were proposed many methods, which are more or less successful on some restricted classes of such systems, and a lot of research is on the way in this area.

One of the highly promising approaches so far proposed for verification of parametric or infinite state systems is *regular model checking* [3] (other methods include, e.g., the use of the so-called cut-offs, network invariants, automated abstraction, etc.).

In the regular model checking, the system configuration is represented as a word over a finite alphabet. If the infinite set of configurations is regular, it can be described by a regular language and finite automaton (For non-regular sets can be used a regular over-approximation). The program behaviour is described by a finite transducer. Sets of initial and bad configurations are also described by finite automata.

Verification is then based on computing the set of reachable states from initial configurations by repeatedly applying the transducers on the set of initial states (or on repeatedly composing the transducers with the aim of computing the reachability relation). During the whole computation is being checked intersection with set of bad states. If the intersection is not empty, computation will be interrupted. As the problem being solved in regular model checking is in general undecidable, the method *does not necessarily terminate*. To facilitate the termination, various *acceleration methods* have been proposed. They include, e.g., widening [3], collapsing of automata states based on the history of their creation by composing transducers [2], or abstraction of automata [5].

The basic approach of regular model checking can be generalized in multiple ways including the use of more general classes of automata than finite state automata [8] or automata on more complex structures than words—e.g., on trees [4, 1]. The long-term goal of our work is to generalize the approach of abstract regular model checking, which currently belongs among the most efficient ways of regular model checking on words to trees and to apply it in verification of the modern concurrent software.

Abstract regular model checking (ARMC) [5] is a combination of regular model checking and abstractions. After each transduction, current automaton is abstracted. Abstractions are one of the successful acceleration techniques in regular model checking. The abstract regular model checking method was successfully used by A.Bouajjani et al for verification of programs with 1-selector dynamic data structures. They proposed encoding of a restricted heap into words and program statements into the finite state transducers [10]. Now we would like to generalise this approach to more general data structures. Words and finite automata is not suitable to do this, so we are interested in regular tree automata. We work on methods how to encode a heap into a tree and program statements into tree transducers. It is much more complicated than in the linear case.

For experimental purposes, it is necessary to have a suitable tool for handling tree automata and tree transducers. First, we decided to use the Timbuk library [9]. Prototype implementation of tree transducers was done, but tests with abstract regular model checking showed complexity problems. The most problematic part is determinisation of tree automata. We decided to use the guided tree automata [12] library, which is a part of the Mona project [12]. This library was written with a high interest in efficiency. Structure preserving transducers can be encoded in this library like ordinal automata, and transduction can be performed by standard automata operation.

This paper presents an ongoing research on application of abstract regular model checking in the infinite state systems verification.

2 MONA

Mona [12] is a tool which was designed for decision of WS1S and WS2S (Weak Second-order Theory of One or Two succesors) formulas validity. WS1S, is a fragment of arithmetic augmented with second-order quantification over finite sets of natural numbers. Mona is based on the relation between the logic and the finite (tree) automata theory. For each WS1S formula, there exists a corresponding finite automaton. For each WS2S formula there exists a corresponding tree automaton. Validity of a formula is equal to non-emptiness of the corresponding automata. The conjunction of formulas is related to the

intersection of automata, the formulas disjunction to the automata union, the negation to the automata complementation. Mona receives a formula, converts it into the corresponding automaton and checks for emptiness.

The implementation of Mona contains two interesting libraries. One for finite automata and one for guided tree automata (GTA). Both are based on BDDs (binary decision diagrams) with a strong emphasis to efficiency and complexity.

We are going to use the GTA library to manipulate bottom-up tree automata. (Class of GTA contains whole class of bottom-up tree automata). Bottom-up tree automaton [6] is a generalisation of finite automaton. It is a finite state machine, which starts with several heads (one at each leaf), and read the tree from leafs to the root.

2.1 TRANSDUCERS IN MONA LIBRARY

The GTA library does not provide transducers. However, as we proposed here, it allows us to encode structure preserving transducers like automata. In the GTA library, a symbol is a node is encoded like several binary numbers. When we work with a logic in Mona, each number corresponds to one variable. This can be used for transduction. Normal automata read just the first variable - the second one is equal to an arbitrary word of $\{0,1\}^*$. Transducers have in the first variable input symbols and in the second one output symbols. The transduction can thus be done in the following steps: (1) Intersection between the transducer and the original automaton. (2) Projection over the first variable. (3) Rename of the second variable in the result on the first one. The figure 1 demonstrate the transduction of one rule.

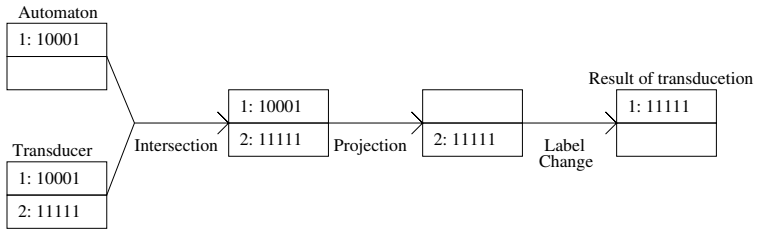

Figure 1: Transduction in GTA Mona library

2.2 ABSTRACTIONS ON TREES

The generatization of abstract regular model checking to abstract regular tree model checking required to define abstractions on trees. In [5], some abstractions on words were discussed. Abstractions are usually based on a state collapsing (instead of two or more states is left one with behaviour of all of them). We are working on their generalization on trees. So for, we have prototypely implemented finite deep trees abstraction (equal to finite length words). Two states are collapsed in case that the languages of finite deep, which are accepted by this states are exactly the same. A work on the other, more complex methods is on the way.

3 APPLICATION OF TREE LANGUAGES

We discuss here the applications of abstract regular tree model checking, which we are now developing. First possibility is to use process rewrite systems (PRS) [7] as a

formalism between programs and tree automata. Program configuration is represented in PRS by a process term, which has naturally tree structure. A set of such terms can be described as a tree automaton. A program behaviour described by a PRS can be (for some class of PRSs) described as tree transducers. Then we can use abstract regular tree model checking to verify this model. This idea was described in [11].

We are now interested in the use of the regular mode checking [5] for fully automatic verification of programs with dynamic data structures (i.e. programs with pointers) by means of abstract regular model checking as a generalization of the method applied in 1-selector data structures described in [10]. In the1-selector case, data are represented like words over a finite alphabet. A word contains several parts separated by the symbol "|". The first part describes pointer variables, which point to null, the second one undefined pointer variables. After this two parts, there are several parts, which describes list sequences. Each list sequence ends by "#" - null pointer, or "!" - undefined pointer. For example the sequence "$x|y|z///\#$" describes a configuration, where "x" is a null pointer, "y" is an undefined pointer, and "z" points to a sequence of 3 elements. This sequence ends by a null pointer.

To describe a list sharing, back pointers, etc, markers were introduced. A finite number of marker pairs (each pair contains the *from marker* m_f, and the *to marker* m_t) is defined. The *from marker* can be placed just at the end of a list sequence (instead of "#", or "!"). The *to marker* can be placed at any position in a list sequence. Each marker pair can occur at most once in a word describing a program state. For example, the list in Figure 2 can be described as a word "$||x//m_t//\#|y//m_f$".

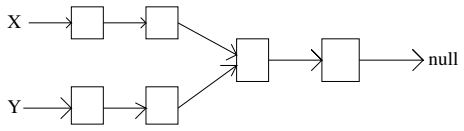

Figure 2: A list with sharing

The encoding has not a canonical form, and so one list can be encoded in different ways. The list from fig 2 can also be described like the word "$||m_t n_t//\#|x//n_f|y//m_f$". So during the computation, it is necessary to count with all possible encodings of the set. This method was successfully tested on a series of examples (e.g. list reversal, insertsort, ...).

Now we are working on encoding configuration of programs with more selectors into tree automata and its behaviour into tree transducers. There is a possibility to use markers, as in the case of words. In the case of words, it is enough to have a finite number of markers (equal to the number of pointer variables). In the case of trees, there is an unbounded number of leaves. It causes, that we are able to describe just a very limited set of program configurations (e.g. doubly linked list can not be described by markers). An other possibility is to use some kind of pushdown tree automata, which allow us to have an unbounded number of markers (a marker is a sequence of letters). Unfortunately, even the class of pushdown word automata is not closed under intersection, and some other interesting properties are not decidable. It is necessary to propose some other mechanisms.

Currently, we are working on two approaches. The first approach use special symbols called *pointer descriptors*. They are placed in the tree, and described pointer direction relatively to the tree shape. The second one encode program configuration into two machines.

One bottom-up tree automaton, which describes data without extra pointers, and one finite state automaton, which describes extra pointers. This two automata are connected by a finite number of states (a final state of the tree automaton is an initial state of the finite automaton).

4 CONCLUSION

We are going to use regular tree model checking method for automatic verification of programs. This methods on words was successfully used for verification of programs with dynamic data structures with 1-selector. We are now going to generalise this approach to data structures with more selectors. It is not easy to do it at words, so we choosed regular tree languages and automata. For experiments is necessary to have a tool for handling with tree automata and transducers. We choosed the Mona GTA library, which allows us to handle bottom-up tree automata and structure preserving transducers.

ACKNOWLEDGEMENT

The work was supported by the Grant Agency of the Czech Republic under the contracts 102/04/0780 and 102/03/D211.

REFERENCES

[1] Abdulla, P., Jonsson, B., Mahata, P., d'Orso, J.: Regular Tree Model Checking. In Proc. CAV'02, LNCS 2404, 2002.

[2] Abdulla, P., d'Orso, J., Jonsson, B., Nilsson, M.: Algorithmic Improvements in Regular Model Checking. In Proc. CAV'03, LNCS 2725, 2003.

[3] Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular Model Checking. In Proc. CAV'00, LNCS 1855, 2000.

[4] Bouajjani, A., Touili, T.: Extrapolating Tree Transformations. In Proc. CAV'02, LNCS 2404, 2002.

[5] Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract Regular Model Checking, In Proc. CAV'04, LNCS 3114, 2004.

[6] Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications, http://www.grappa.univ-lille3.fr/tata.

[7] Esparza, J.: Grammars as processes. In Formal and Natural Computing, LNCS 2300, 2002.

[8] Fisman, D., Pnueli, A.: Beyond Regular Model Checking. In Proc. FSTTCS'01, LNCS 2245, 2001.

[9] Genet, T.: Timbuk, a tree automata library, http://www.irisa.fr/lande/genet/timbuk.

[10] Bouajjani, A., Habermehl, P., Moro, P., Vojnar, T.: Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking In Proc. of ACAS'05, to appear in LNCS, April 2005.

[11] Rogalewicz, A., Vojnar, T.: Tree Automata In Modelling And Verification Of Concurrent Programs In Proc. of ASIS 2004, MARQ, 2004.

[12] Klarlund, N., Muller, A., Schwartzbach, M. I.: The MONA Project http://www.brics.dk/mona/