DiVinE – A Tool for Distributed Verification

Jiří Barnat, Luboš Brim, Ivana Černá, Pavel Moravec, Petr Rockai, and Pavel Šimecek

Department of Computer Science, Faculty of Informatics
Masaryk University, Brno, Czech Republic

Abstract. We present a tool for distributed-memory LTL model-checking and reachability analysis that emulates clusters of workstations. The tool incorporates several novel distributed-memory algorithms and provides a unique interface to use them. We describe the basic structure of the tool, discuss main architecture decisions made, and briefly explain how the tool is used.

1 Introduction

A few enumerative verification tools have been developed to support engineers in their verification needs. Despite significant improvements in model-checking techniques, their verification capabilities are in the case of real-life industrial models limited by the amount of data a single state-of-the-art computer is able to handle efficiently.

In recent years, extensive research has been conducted in parallel and distributed model-checking with the aim to push forward the frontier of enumeratively verifiable systems [6, 2, 5, 3]. Consequently, several distributed verification tools emerged. The deployment and usage of a distributed tool is significantly more demanding compared to the sequential case. It requires a cluster with properly installed message passing software and also some programming skills in the case the tool has to be compiled from its source codes. These are some of the reasons why distributed verification tools are rarely used, although their verification capabilities are undoubtedly bigger compared to sequential tools.

The goal of Distributed Verification Environment project (DiVinE) is to provide an extensible framework to support distributed verification on clusters. DiVinE offers three means to achieve this goal: First, a library of common functions (DiVinE Library) on top of which various distributed verification algorithms can be implemented. Second, a collection of state-of-the-art distributed verification algorithms incorporated into a single software product (DiVinE Tool) which is as easy to install as most sequential tools. And third, a ready-to-use cluster for users of sequential tools in case they need to run experiments using DiVinE Tool without having access to their own clusters. In this paper we report on DiVinE Tool only.
2 DiVinE Tool

DiVinE Tool is a parallel, distributed-memory enumerative model-checking tool for verification of concurrent systems. The tool can employ aggregate power of network-interconnected workstations to verify systems whose verification is beyond capabilities of sequential tools.

DiVinE modeling language is rich enough to describe systems made of synchronous and asynchronous processes communicating via shared memory and buffered or unbuffered channels. System properties can be specified with formulas of Linear Temporal Logic (LTL) and with processes describing undesired behaviour of systems under consideration (negative claim automata). Thanks to DivSPIN project [1], DiVinE Tool is also capable of verifying models written in ProMeLa.

From the algorithmic point of view, the tool is quite unique. In automata-based approach to LTL model-checking, the verification problem is reduced to problem of accepting cycle detection in the graph of Büchi automaton. Two algorithms are typically used for solving the problem: Nested Depth-First Search algorithm and Tarjan’s algorithm for decomposition of the graph into strongly connected components. Unfortunately, they both strongly rely on depth-first search postorder that is known to be difficult to be computed in parallel. Therefore, new parallel algorithms had to be designed for accepting cycle detection. These are, namely, algorithm for cycle detection using additional dependency data structure, algorithm based on negative cycles, algorithms for forward and backward elimination of trivial and non-accepting strongly connected components, algorithm for cycle detection based on breadth-first search, and algorithm based on propagation of the value of maximal accepting predecessor. Besides these, DiVinE Tool includes also algorithm for distributed state space generation and algorithm that performs sequential NestedDFS in a distributed-memory setting. More details on algorithms can be found on DiVinE project web pages [4].

DiVinE Tool can be deployed either as a complete software package to be installed on separate Linux cluster or as a small Java application to access preinstalled clusters. In the first case, basic Linux administrator skills are required to install the tool, but user is in the full control of environment settings under which distributed algorithms are to be executed and can also control the tool from a command line. In the second case, the tool can be used employing DiVinE preinstalled clusters and accessed via graphical user interface. The graphical user interface (GUI) requires properly installed Java Runtime Environment. Both versions are available on DiVinE project web page [4] together with a few models determined for initial acquaintance with the tool.

An important part of the DiVinE project is the maintenance of a public server and limited number of DiVinE dedicated clusters. For security reasons only registered users are allowed to connect to DiVinE public server. New users can be registered by following instructions given on DiVinE project web-pages.

---

1 For reviewers we prepared account CAVreviewer with password cav2006.
3 Using DiVinE Tool with GUI

The description of command line interface is beyond the scope of this paper. Therefore, we focus on controlling DiVinE Tool with GUI only. GUI is implemented as a client-server application where the server part is responsible for the control of the tool. This means the server maintains currently verified models, executes distributed algorithms, monitors cluster load, etc.

Client window is divided into three parts (see a screenshot in Figure 1). The biggest part is the main working area where models and properties are specified, and outputs of distributed algorithms are displayed. The tree structure on the left side is used to browse currently loaded models including corresponding properties and verification results. The panel at the bottom displays messages reporting changes in the status of running algorithms.

A new verification project is started with button New model. System to be verified can be written directly into the main window or imported from a local file. Having specified a model of the system the user is expected to provide properties the system should meet using button Add property. Besides distributed state space generation, the tool is capable of verifying full range of LTL formulas over state-based atomic propositions. Atomic propositions are specified using keyword #define, e.g. #define p x>3, the formula is specified using keyword #property, e.g. #property FG(p). Property specification can also be imported from a local file. The pair model-property is called a task. User can assign several distributed algorithms to be run for a given task. The number of workstations to be used can be specified for every algorithm. Algorithms are initiated with button Execute.

Each algorithm performed produces two different types of output that can be accessed with the client: the standard output and log files. While the standard

---

3
output is used to report progress in the computation and final verification results, logs are used to record multiple statistics for every computer participating in the computation. Logged values include the amount of memory currently allocated by the algorithm, number of sent and received messages, time spent in user and kernel space, size of queue of unexplored states, etc. Client displays the last logged values with refresh rate around five seconds, which allows the user to monitor the status of the computation in almost real time.

All specified models, properties and verification results are stored on the server until they are explicitly removed. Therefore, user can disconnect from the server during performance of an algorithm and reconnect later to collect verification results. It is also possible to specify and initiate new tasks during computation of other ones. Hence, several tasks can be computed in parallel.

4 Demonstration

Below we report verification time and maximal local memory consumption we measured during verification of model and (Anderson’s mutual exclusion problem). Note that we neither were able to verify this model with sequential Nested DFS algorithm in DivInE Tool nor we were able to do so with corresponding ProMeLa model in SPIN.

<table>
<thead>
<tr>
<th>Number of computers</th>
<th>10</th>
<th>15</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. Local Memory (MB)</td>
<td>—</td>
<td>600</td>
<td>523</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>—</td>
<td>2703</td>
<td>2552</td>
</tr>
</tbody>
</table>

References