Integrating Temporal Specifications as Runtime Assertions into Parallel Debugging Tools
Project Report

Gábor Kusper, Wolfgang Schreiner
RISC-Linz

Róber Lóvas
MTA-SZTAKI

March 10, 2002

Abstract

Debugging nondeterministic parallel programs is a difficult and tedious task. We report about a project, which investigate the use of program specifications, which can serve as enhanced debugging aids in existing parallel debugging tools. For parallel programs, an adequate framework for specifying program specification is provided by temporal logic, which allows to describe the dynamic behavior of a parallel program. We developed a checker, called T(emporal)L(ogic)C(hecker) Engine, which is able to check temporal logic formulae for each state of a parallel program along one possible execution. The checker communicates with a debugger, which executes a nondeterministic parallel program. For this we use a well-defined protocol. We extended two parallel debugger such that they use this protocol. This project thus applies the idea of *model checking* temporal logic specifications to the area of *parallel and distributed debugging*, something that has to our knowledge not been investigated before.

Supported by ÖAD-WTZ Project A-32/2000 "Integrating Temporal Specifications as Runtime Assertions into Parallel Debugging Tools".

1 Introduction

Testing and debugging are two important activities of the software lifecycle, which try to improve the quality of software by addressing two important quality criteria - correctness and reliability. The correctness of a program is always expressed relative to its specification, while software reliability defines the probability of future satisfactory operation [12].

Describing the behavior of sequential programs can be performed with classical logic by a predicate (the output condition) that must hold after the execution of the program. Furthermore, the output condition can be translated (by the technique of weakest preconditions) into conditions that must hold at every step of the program. Such a condition can thus be considered as an *assertion* that must hold at a particular program step; if we restrict our attention to a particular subclass of formulas, such an assertion can be *checked* at runtime. Annotating a program by runtime assertions is a simple but very effective way of increasing the code's reliability and thus the user's confidence in a program's correct behavior. While the fact that a particular assertion holds does not prove correctness of a program, the fact that it fails exhibits an error in (at least the understanding of) it.

For parallel programs, an adequate framework for specifying program behavior is provided by temporal logic, which allows to describe the dynamic behavior of a system (program) consisting of multiple asynchronously executing components (processes) [4]. Therefore, a temporal logic formula can be considered as the *specification* of a concurrent program. An implementation of the program is thus correct if every possible program execution satisfies the formula [5] [6] [7]. This area of formal models is one of the main points of expertise at RISC-Linz [1].

In concurrent programs, however an assertion should not only be checked in one program path but in many of the possible program paths. Especially nondeterministic programs may yield different results in successive program executions, even if the same input data are provided. In fact, in the general case there will be too many paths to check the assertion in all the possible execution paths. Thus, it is not sufficient to execute the program a number of times, but instead we need a way to control the behavior of a nondeterministic program in order to select those paths that should be investigated. Furthermore, if the program state is physically distributed, we need a global snapshot of the partial states on which to perform the assertion check.

Debugging Parallel Programs Corresponding to the importance of program debugging for the program's correctness and reliability, many on-going research projects are developing tools to support users in this area [14]. Two contrary approaches are taken by MTA-SZTAKI and GUP Linz, which have both developed tools for the development of parallel message-passing programs including debuggers which allow to monitor the states of arbitrary parallel nondeterministic programs.

The P-GRADE environment [8] developed by MTA-SZTAKI is a visual programming environment that includes the distributed debugger DIWIDE [2]. In debugging mode, DIWIDE controls the execution of the target program by providing commands to create breakpoints, step-by-step execution, animation, etc. In trace mode, DIWIDE generates a trace file containing all the trace events defined by the user for later semi-online-visualization. More recently, as a third mode, *macrostepping* has been added as a new method for searching bugs coming from nondeterministic behavior which gives the possibility to test all branches of the application in a concurrent manner [9]. This means that all possible timing conditions can be tested with the help of macrostep debugging in DIWIDE by controlling the channel or port selection at every alternative input block. DIWIDE is able to force any process to receive the message from the selected port or process when it reached an alternative input (CAIALT) communication block. It is achieved by stopping every process at the next communication point and scanning the possible communicating process pairs. Each communicating process pair will continue its execution at the next macrostep and at CAIALT communication the user can choose the sender process.

A contrary approach is taken by GUP Linz, who developed the record&replay tool NOPE (NOndeterministic Program Evaluator) [3] for testing and debugging of nondeterministic message passing programs within the MAD environment [10]. The operation of NOPE is divided into a record phase and a replay phase [13]. The initial record phase is used to generate partial traces containing only ordering information of critical events. During replay these trace data are used to enforce the same event ordering as occurred during record, which guarantees equivalent execution. Additionally, complete traces can be generated to serve as input for the corresponding post-mortem analysis tool [11]. In order to avoid the probe effect of monitoring program execution, the monitor instruments only receive statements and produces trace records only for wild card receive events (nondeterministic choice points). These data are sufficient to produce complete traces with exhaustive information about the execution during subsequent replay phases. Furthermore, the automatic race condition detection and manipulation mechanism of NOPE allows to evaluate other possible program runs for the supplied input. It searches the complete traces for all occurrences of wild card receives and evaluates race candidates. These are messages that are possibly accepted at a specific wild card receive. Then additional replay steps can be initiated to uncover previously hidden executions. The idea is that at each wild card receive every race candidate arrives first during one execution. NOPE automatically changes the ordering with event manipulation and starts a reply phase for each permutation. The follow–up replays deliver all possible executing that are directly connected to the original program run. The evaluation of message race candidates has to be repeated iteratively, in order to detect indirectly connected executions as well.

Both tools therefore address the same problem by different techniques. In DIWIDE, the user may interactively control the investigation of program paths by providing a selection strategy (e.g. depth-first) or by manual choice. In NOPE, a particular run of an instrumented program is recorded from which afterwards related program runs can be automatically generated.

Project Content and Expected Results The goal of this project is to generate from a temporal logic specification of a distributed program runtime assertions that can be used by the MTA-SZTAKI DIWIDE and the GUP Linz NOPE to check the correctness of particular sets of program runs.

This project thus applies the idea of *model checking* temporal logic specifications to the area of *parallel and distributed debugging*, something that has to our knowledge not been investigated before.

This framework is specified in the slide presentation [15] made by Wolfgang Schreiner and discussed in the kick-off meeting of the project.

The main results of the project are the two enhanced debugger tools providing a novel testing and debugging technology for developers of parallel and distributed programs.

2 Structure of the Checker

The checker (TLC Engine v1.1.2) is written in Java 1.1. Therefore, it usable everywhere Java 1.1 is available. The classes of the checker belong to the checker package.

The main entities of the engine are `State`, `Formula` and `StateFormula`. The class `State` represents a program state of the nondeterministic parallel program. `Formula` and `StateFormula` are interfaces. The interface `Formula` represents a temporal logic formula. The interface `StateFormula` represents a node of the state formula tree.

Details about `State`, `Formula` and `StateFormula` An instance of `State` class has a parent state (or it is the first program state) and a child state (or it is the last state). Every state may have several state formulae to be checked.

`Formula` and `StateFormula` are interfaces. For both interface we have an abstract class implementing the common methods. Those are `FormulaBase` and `InstanceBase`, respectively.

Instances of classes implementing the interface `Formula` (and inherited from `FormulaBase`) are temporal logic formulae. Every formula has an outermost operation (either future/past operator or quantifier or logical connective) and some subformulae depending on the arity of the outermost operation. We have the following formula classes. Future temporal operators are `Next`, `Always` and `Eventually`, past temporal operators are `Prev`, `Once` and `SoFar`, quantifiers are `ForAll`, `Exists` and `Let`, logical connectives are `Conjunction`, `Disjunction`, `Negation` and `Implication` and the atomic predicates are represented by `Atomic`.

Instances of classes implementing the interface `StateFormula` (and inherited from `InstanceBase`) are nodes of the state formula tree. We call such a node a state formula. Every state formula is an instance of a temporal logic formula, has a parent state formula (or it is the root of the state formula tree) and several child state formula (or it is a leaf), has a boolean3 value (true, false, unknown represented by the class `Bool3`) and belongs to a program state where it should be checked. Such classes have the name of the outermost operation plus the word `"Instance"`. For example, the class `AlwaysInstance` represents an `Always` formula instance.

The child state formulae of a state formula belong to the same program state as the state formula, see section ??. If we have a future (past) temporal logic formula instance then it has also a child state formula that belongs to the next (previous) program state. This child state formula is an instance of the class `StateFormulaNext` (`StateFormulaPrev`).

Terms Beside formulas we need terms to write temporal logic specification. We consider only integer (java `Integer` type) terms. The abstract class `Term` represents a term. A term has a name, a value (which is integer) and may have several term parameters if it a function application. The following classes are inherited from `Term`. The class `Application` represents a function application. The class `Const` represents a constant (has value but doesn't have name). The class `Variable` represent a variable (has name but doesn't have value).

The class `Context` is a hash table. It contains the values of quantified variables.

Prefetch Prefetching is a mechanism that makes it possible to work with past temporal operators.

The problem with past temporal operators is the following. If we evaluate a temporal (state) formula which has a past temporal sub-formula (it contains at least one past temporal operator) then there is a possibility that in some future state we would have to ask the debugger what *was* the value of an atomic predicate and/or a term in this state. However, we have no possibility to ask questions about the past.

We use the so-called prefetch technique to solve this problem. We "prefetch" atomic predicate and term values, i.e., evaluate all past temporal formulae (normally it would not be needed). In this way we get the value of all sub-formulae, especially we get values of atomic predicates and terms. We place those values in the value cache of this state. If we need a value in same future state from this state (because of a past temporal operator), then we just look up the value cache of this state.

Communication The class `Messages` describes messages between the debugger and the checker. We have 15 kind of messages, each kind of message identified by a key from 0 to 14. A message has such a key and a body. The body contains the parameters of the message. We can have fixed and varying size message regarding to the number of parameters of the message. The first parameter of a varying size message is the number of remaining parameters. Some kind of messages may be sent only by the checker. Those are "eval", "evalterm", "answer", "error", "warning", "okay", "failed". Some kind of messages may be sent only by the debugger. Those are "startup", "assert", "next", "done", "truth", "value", "abort". The message "protocol" may be sent by both. See section State Machine of the Protocol.

The class `Protocol` together with `Term` and `AtomicInstance` describe the protocol. The class `Protocol` describes the so-called `MainLoop`, see section State Machine of the Protocol. The class `Term` describes the so-called `EvalTermLoop` and `AtomicInstance` the `EvalLoop`. An instance of the class `Protocol` will be our server. The method `serve()` first send a "protocol" message to the debugger and if it uses the same protocol version than it starts an infinite loop (see `MainLoop`) waiting messages form the debugger.

The classes `Buffer`, `Cell`, `InThread` and `OutThread` are in charge of reading and writing the communication channels. In fact the TLC Engine use two buffered FIFOs.

The `Checker` and the `Global` class The class `Checker` is the main program. It sets some global variable, for example, the version of the checker. Than it instantiates the `Protocol` class and call the `serve()` method of it, which will communicate with the debugger.

The class `Global` is the one you should enlarge if you would like to work (e.g. visualize) with the state formula tree. It has three methods. The method `callNotifyAdd(StateFormula f)` is called whenever a new state formula added to the tree. The method `callNotifyValue(StateFormula f)` is called whenever a value of a node is changed. It means that an unknown node becomes either true or false. The method `callNotifyRetract(StateFormula f)` is called whenever a node is no more relevant to determine the vale of the root node (the specification).

Limitation of the Checker Some useful temporal operator is not implemented. For example, `LeadsTo` is not implemented. But we can express them in terms of implemented operators.

Specification that contains past temporal operator can be asserted only in the first program state, i.e., before the first next message. It is so because we (neither the debugger nor the checker) do not know what was the value of terms and atomic predicates in past program states.

There is no visualization tool for the checker.

3 How to Write a Specification

This section contains a sample specification with rich comments. To create your own specification, please use this sample.

```
package checker;
import checker.*;

// This is a sample specification class.
// You can write your own specification by extending the interface
// Specification, rewriting getFormula and writing an empty constructor.
// In getFromula you can use getNumOfProcs, which tells you the
// number of processors.
// The method getFormula returns a temporal logic formula. In order to create
// this formula you can use the following classes (see "class name".java):
// Logic connectives:
//   Temporal operators:
//     Future temporal operators:
//       Next(formula), it is depicted by "()",
//       Always(formula), it is depicted by "[]",
//       Eventually(formula), it is depicted by "<>".
//     Past temporal operators:
//       Prev(formula), it is depicted by "(-)",
//       SoFar(formula), it is depicted by "[-]",
//       Once(formula), it is depicted by "<->".
//    Quantifiers:
//      Let(varname, term, formula),
//      Exists(varname, from_term, to_term, formula),
//      ForAll(varname, from_term, to_term, formula).
//    Connectives:
//      Conjunction(left_fomula, right_formula), it is depicted by "&&",
//      Disjunction(left_fomula, right_formula), it is depicted by "||",
//      Implication(left_fomula, right_formula), it is depicted by "=>",
//      Negation(formula), it is depicted by "!".
//    Atomic formula:
//      Atomic(predicatename, term_array).
//
// Some logic connective has term parameters, the following classes are terms:
//   Const(int32_value),
//   Variable(varname),
//   Application(functionapplicationname, term_array).
//
// To use your specification you have to send the following
```

```
// message to the checker
//   assert
//   checker.MySpec
// The checker will respond according its protocol, see the file PROTOCOL

class MySpec extends Specification
{
  // a specification have to have empty constructor
  // it is needed for instanceate it
  public MySpec() { }

  // you are asserting the formula returned by getFormula
  public Formula getFormula()
  {
    String var = "x";
    Term[] vars = new Term[1];
    vars[0] = new Variable(var);
    Const Zero = new Const(0);

    // we return the following temporal logic formula:
    // []( p() => Exists x in 0..NumOfProcs : q(x) )
    return new Always
      (new Implication
        (new Atomic("p", null),
           new Exists(var, Zero, getNumOfProcs(),
             new Atomic("q", vars))));
  }
}
```

4 State Machine of the Protocol

This section contains verbatim the file PROTOCOL from the TLC checker package. It describes the protocol (between the TLC Engine and a debugger, which executes a nondeterministic parallel program), the messages and gives an example. The description is given as a sate machine. There are seven state: `InitialState(1|2)`, `MainLoop`, `EvalLoop`, `EvalTermLoop` and `EndState`. In the `InitialState(1|2)` the checker and the debugger tell each other the protocol they use, see description of "protocol" message, and the debugger tells how many processes the parallel program have, see description of "startup" message. In the `MainLoop` the debugger may assert new assertions and tells whether a new program state or the last program state is available, see description of "assert", "next" and "done" messages. If a new program state is available (i.e., checker receives a "next" message) and there is at least one assertion to be checked then the `EvalLoop` takes place (for each assertion one `EvalLoop`). In the `EvalLoop` the checker asks the debugger what is the value of atomic predicates needed to evaluate the specification, see description of "eval" and "truth" messages. If the checker has to know also the value of a term in order to evaluate the specification then the `EvalTermLoop` takes place. The `EvalTermLoop` is similar to `EvalTerm`, the difference is here we use "evalterm" and "value" messages, see description of them. The `EndState` is the final state of the state machine. This explanation does not cover the whole protocol but gives the basic idea. The detailed description is given in the next paragraph.

The State Machine

```
State machine of the protocol of Temporal Logic Checker Engine v1.1.2.

You can start Temporal Logic Checker Engine (from now on checker) with
the following command:
java checker.Checker

The checker communicates with a debugger, which executes a nondeterministic
parallel program to which predicate functions are (possibly dynamically) linked.

The checker and the debugger communicate by the following protocol
specified as a state machine:
State           Input Message   Output Message                  Next-State
--------------------------------------------------------------------------
InitialState                    protocol                        InitialState1
InitialState1   protocol        okay                            InitialState2
                protocol        error failed                    EndState
                not_expected_m  error failed                    EndState
                unregistered_m                                  EndState
InitialState2   startup         okay                            MainLoop
                startup         error failed                    EndState
                not_expected_m  error failed                    EndState
                unregistered_m                                  EndState
MainLoop        assert          okay                            MainLoop
                assert          error failed                    MainLoop
                next            {answer} eval                   EvalLoop
                next            {answer} evalterm               EvalTermLoop
                next            {answer} okay                   MainLoop
                next            {answer} error failed           MainLoop
                done            {answer} okay                   EndState
                abort                                           EndState
                not_expected_m  error failed                    MainLoop
                unregistered_m                                  EndState
EvalLoop        truth           {answer} eval                   EvalLoop
                truth           {answer} evalterm               EvalTermLoop
                truth           answer {answer} okay            MainLoop
                truth           answer {answer} error failed    MainLoop
                abort                                           EndState
                not_expected_m  error failed                    EvalLoop
                unregistered_m                                  EndState
EvalTermLoop    value           eval                            EvalLoop
                value           evalterm                        EvalTermLoop
                value           error evalterm                  EvalTermLoop
                abort                                           EndState
                not_expected_m  error failed                    EvalTermLoop
```

```
                unregistered_m                                  EndState
EndState

, where {} means 0 or more times.

Messages
--------
abort[]
  The debugger may send this message.
  If the checker receives this message then it aborts.

answer[arg1, arg2]
  The checker may send this message.
  It tells the truth value of an assertion in the current state.
  It is sent by the checker as a response to an assert, next or done message.
  After an assert message the checker sends one answer message.
  After a next or done message the checker sends that many answer messages
  as there were unknown assertions in the previous state.
  Parameters:
    arg1: String, name of the specification,
    arg2: {true, false, unknown}, truth value of the specification.

assert[arg1]
  The debugger may send this message.
  The debugger asks the checker to check an assertion from the current
  state on. The checker will respond with an answer message (if there was
  no error) and an okay message.
  Parameter:
    arg1: String, name of the specification to be checked.

done[]
  The debugger may send this message.
  It tells the checker the last state is available.

error[arg1]
  The checker may send this message.
  It is sent by the checker, if an error occurs.
  Parameter:
    arg1: String, explanation of the error.

eval[arg1, arg2, ...]
  The checker may send this message.
  The checker asks the debugger to evaluate a predicate described
  by the arguments. The debugger responds by a truth message.
  It is a varying size message.
  Parameters:
    arg1: 1..MaxInt32, number of parameters of this (varying size) message,
    arg2: String, name of the predicate,
    ... : Int32, parameters of the predicate.
  Note that arg1 is 1 + number of parameters of the predicate.
```

```
evalterm[arg1, arg2, ...]
  The checker may send this message.
  The checker ask the debugger to evaluate an application function
  described by the arguments. The debugger responds by a value message.
  It is a varying size message.
  Parameters:
    arg1: 1..MaxInt32, number of parameters of this (varying size) message,
    arg2: String, name of the application function,
    ... : Int32, parameters of the application function.
  Note that arg1 is 1 + number of parameters of the application function.

failed[]
  The checker may send this message.
  It is sent if the message sent by the debugger was unsuccessful.

next[]
  The debugger may send this message.
  It tells the checker there is a new state available.

okay[]
  The checker may send this message.
  It is sent if the message sent by the debugger was successful.

protocol[arg1]
  The checker and also debugger may send this message.
  The sender tells what protocol it uses.
  Parameter:
    arg1: String, the protocol which is used by the sender.

startup[arg1]
  The debugger may send this message.
  The debugger tells the startup parameters to the checker.
  Parameter:
    arg1: 0..MaxInt32, number of processes of the debugged parallel program.

truth[arg1]
  The debugger may send this message after it receives an eval message.
  The debugger tells the checker the truth value of the predicate that
  is described by the eval message sent by the checker.
  Parameter:
    arg1: {true, false, unknown}, truth value of the predicate described by the eval.

value[arg1]
  The debugger may send this message after it receives an evalterm message.
  The debugger tells the checker the truth value of the application
  function that is described by the evalterm message sent by the checker.
  Parameter:
    arg1: Int32, value of the application function described by the evalterm.
```

```
not_expected_m
  Not expected message.
  Can stay for any message that is not listed in the state in the
  Input Message column.
  For example the value message in the MainLoop state.

unregistered_m
  unregistered message.
  Can stay for any string, which doesn't correspond to a message.

Interpretation
--------------
  key[arg1,...argn] is a sequence of lines. This is represented as follows:
  key
  arg1
  ...
  argn
  separated by the "new line" character.

Other comments
--------------
- If in a state you send a message, which is not expected in this state you
  will receive an error and a failed message and the state remains the same
  except in the state InitialState1 where after a  not expected message
  the next state is EndState.
- If you send an unregistered message then the checker terminates.
- EndState means that the checker terminates.

Example
-------
gintonic:4> java checker.Checker
protocol
checker.Checker.externalProtocol 1.1.2
protocol
checker.Checker.externalProtocol 1.1.2
okay
startup
16
okay
assert
checker.MySpec
okay
next
evalterm
1
x
value
3
eval
2
```

```
p
3
truth
true
answer
checker.MySpec
unknown
okay
done
answer
checker.MySpec
true
okay
gintonic:5>
```

References

[1] Wolfgang Schreiner. *On Engineering a Distributed Algorithm.* Helmut Gutmann (ed.), Software for Communication Technologies, Third International Austrian-Israeli Technion Symposium with Industrial Forum, April 26-27, 1999, Hagenberg, Austria, Austrian Technion Society, Vienna, pp. 110-116.

[2] P. Kacsuk and R.Lovas and J.Kovács. *Systematic Debugging of Parallel Programs in DIWIDE Based on Collective Breakpoints and Macrosteps.* 5th International Euro-Par Conference, Toulouse, France, pp.90–97, 1999.

[3] D. Kranzlmüller and J. Volkert. *NOPE: A Nondeterministic Program Evaluator.* ACPC'99 International Conference, Salzburg, Austria, pp. 490-499, February 1999.

[4] Leslie Lamport. *The Temporal Logic of Actions.* ACM Transactions on Programming Languages and Systems, 16(3):872–923, May 1994.

[5] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems – Safety.* Springer, New York, 1995.

[6] K.L. McMillan. *Symbolic Model Checking.* Kluer Academic Publishers, 1993.

[7] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[8] P. Kacsuk, G. Dozsa, R. Lovas. *The GRADE Graphical Parallel Programming Environment.* in: P. Kacsuk, J.C. Cunha and S.C. Winter, "Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments", Nova Science, Chapter 10, 2000.

[9] Kacsuk, P. *Macrostep-by Macrostep Debugging of Message Passing Parallel Programs.* in: 1998 Parallel and Distributed Computing and Systems proceedings, PDCS'98, Las Vegas, Nevada, USA, 1998, pp. 527-531.

[10] D. Kranzlmüller, S. Grabner, J. Volkert. *Debugging with the MAD Environment.* Parallel Computing, Vol. 23, No. 1-2, pp. 199-217 (Apr. 1997).

[11] D. Kranzlmüller, S. Grabner, J. Volkert. *Event Graph Visualization for Debugging Large Applications.* Proc. SPDT'96, ACM SIGMETRICS Symp. on Parallel and Distr. Tools, Philadelphia, USA, pp. 108-117 (May 1996).

[12] B.W. Boehm. *Seven Basic Principles of Software Engineering.* Journal of Systems and Software, Vol. 3, No. 1, pp. 3-24 (March 1983).

[13] Ch.E. McDowell, D.P. Helmbold. *Debugging Concurrent Programs.* ACM Computing Surveys, Vol. 21, No. 4, pp. 593-622 (December 1989).

[14] C.M. Pancake, R.H.B. Netzer. *A Bibliography of Parallel Debuggers, 1993 Edition.* Proc. of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, CA, USA (May 1993).

[15] W. Schreiner. *Temporal Logic Specifications for Parallel Debugging.* http://www.risc.uni-linz.ac.at/people/gkusper/checker/doc/spec/slides-main.pdf, Linz, Austria (2000).