# Integrating Temporal Assertions into a Parallel Debugger*

Jozsef Kovacs[1], Gabor Kusper[2], Robert Lovas[1], and Wolfgang Schreiner[2]

[1] Computer and Automation Research Institute (MTA SZTAKI)
Hungarian Academy of Sciences, Budapest, Hungary
{smith,rlovas}@sztaki.hu
http://www.lpds.sztaki.hu

[2] Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
{Gabor.Kusper,Wolfgang.Schreiner}@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at

**Abstract.** We describe the use of temporal logic formulas as runtime assertions in a parallel debugging environment. The user asserts in a message passing program the expected system behavior by one or several such formulas. The debugger allows by "macro-stepping" to interactively elaborate the execution tree (i.e., the set of possible execution paths) which arises from the use of non-deterministic communication operations. In each macro-step, a temporal logic checker verifies that the once asserted temporal formulas are not violated by the current program state. Our approach thus introduces powerful runtime assertions into parallel and distributed debugging by incorporating ideas from the model checking of temporal formulas.

## 1 Introduction

We report on a system which applies ideas from the model checking of temporal formulas to the area of parallel debugging; its goal is to support the development of correct and reliable parallel programs by runtime assertions that are derived from temporal formulas which describe the expected program behavior.

The behavior of sequential programs can be described 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.

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

Assertions play an important role in the development of sequential programs, but their role in parallel programming is currently far less dominant. One reason is that (due to non-determinism) a program may exhibit for the same input different executions; furthermore, interesting properties talk about the state of the complete system (and not about the state of a single process). Another reason is that the scopes of properties are usually not defined by specific code locations but by temporal relations to other properties. These problems are difficult to overcome in production runs of parallel programs and with classical logic. We therefore turn our attention to program runs controlled by parallel debuggers and to assertions expressed in the language of temporal logic.

Since debugging parallel programs is an important and difficult task, many projects have been developing tools to support the user in this area; for a survey, see [8]. A particular challenge is the mastering of non-determinism which arises in message passing programs from the *wildcard receive* operation, i.e., a receive operation that non-deterministically accepts messages from different communication partners. The NOPE (Non-deterministic Program Evaluator) deals with this problem by generating in a record phase partial traces which contain ordering information of critical events [9, 8]. During replay these data are used to enforce the same event ordering as occurred in the recording phase. The DIWIDE debugger [7, 6] applies the technique of macro-stepping which allows to test all branches of an application in a concurrent manner; we will describe this technique in more detail in the remainder of this paper.

Temporal logic has proved as an adequate framework for describing the dynamic behavior of a system (program) consisting of multiple asynchronously executing components (processes) [10]. A temporal logic formula can be considered as the *specification* of a parallel program; in linear time temporal logic, a program is correct if every possible execution satisfies the formula. If a program itself is described in a formal framework, the technique of model checking can be applied to decide about the correctness of the temporal specification, provided that the program only exhibits a finite number of states [1]. There exist tools for the validation of concurrent system designs based on temporal logic [3] and for the generation of test cases from temporal specifications [4]. In the system presented in this paper, we use actual program runs controlled by a debugger as the universe in which a temporal formula is checked. Thus our work combines ideas from parallel debugging and from model checking.

The approach closest to our ideas is that of *pattern-oriented* parallel debugging pioneered by the program analysis tool Belvedere [5]. This approach was included and extended by the post-mortem event-based debugger Ariadne [2], and later applied in the TAU program analysis environment [11]. Ariadne matches user-specified models of intended program behavior against actual program behavior captured in event traces. Ariadne's modeling language for describing program behavior is based on communication patterns with a notation derived from regular expressions. This language is quite simple; the language of temporal logic used in our system is considerably more expressive and allows to describe the intended program behavior in much more detail.

## 2 Macrostep Debugging in DIWIDE

DIWIDE is a distributed debugger which is part of the visual parallel programming environment P-GRADE. This debugger implements the *macrostep* method which gives the user the ability to execute the application from communication point to communication point [7, 6].

A *macrostep* is the set of executed code regions between two consecutive collective breakpoints. A *collective breakpoint* is a set of local breakpoints, one for each process, that are all placed directly after communication instructions such that a macrostep contains communication instructions only as the last instructions of its regions. In the macro-step execution mode, DIWIDE generates from the current collective breakpoint the next collective breakpoint and then runs the program until the new collective breakpoint is hit. At replay, the progress of the processes is controlled by the stored collective breakpoints; the program is executed again macrostep by macrostep as in the execution phase.

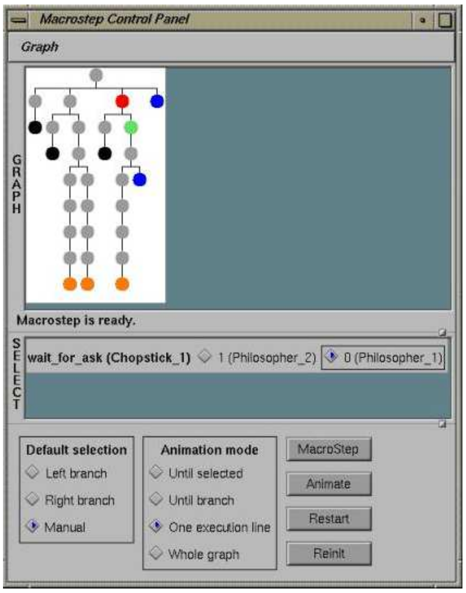


**Fig. 1.** The Macrostep Debugger Control Panel

When a communication operation in a collective breakpoint is a wildcard receive operation, this breakpoint splits macro-step execution into multiple execution paths. Each path represents one possible selection of a sender/receiver pair for all wildcard receive operations in the originating collective breakpoint.

The set of all possible execution paths can be represents by a tree whose nodes represent collective breakpoints and whose arcs represent macrosteps. The macrostep control panel of the DIWIDE debugger visualizes this tree as far as it has been already constructed and allows the user to control its further elaboration (see Figure 1). The user may select particular branches in the tree or let the system automatically traverse the tree according to some strategy. He may also set a *meta-breakpoint* in some node and let the system replay execution along the corresponding branch until the selected node is hit. The system therefore gives the user very powerful means to control the non-deterministic behavior of a parallel program in the debugging process.

## 3 Macrostep Debugging with Temporal Assertions

Before we go into technical details, we will illustrate the use of temporal formulas as runtime assertions by a simple example. Take a parallel program which consists of three processes: a *producer* process which generates a finite number of values and sends them to a *buffer* process which receives values from the producer and eventually forwards them to a *consumer* process which receives the values from the buffer and processes them. The buffer has a finite capacity; depending on its fill state (full, empty, not full and not empty) it waits for requests from one or from both of the other processes (to receive or to send a value) and answers them. Its behavior is therefore in general non-deterministic and may be investigated by the macro-step debugger as sketched in the previous section.

A fundamental property which we expect from the system is that the number of messages stored in the buffer always equals the difference of the number of messages sent by the producer and of the number of messages received by the consumer (we assume a synchronous message passing handshake). Another property is the fact that if the buffer is non-empty, it will eventually get empty. In the notation of temporal logic [10], these properties can be written as

$$\Box \mathrm{NoLostMessage} \land \Box(\neg \mathrm{BufferEmpty} \Rightarrow \Diamond \mathrm{BufferEmpty})$$

where "NoLostMessage" expresses the core of the first property and "BufferEmpty" the core of the second property. The temporal operator $\Box$ reads as "always" and the temporal operator $\Diamond$ as "eventually". This property can be asserted at the beginning of our program by a C statement

```
assert("BufferSpec");
```

where `BufferSpec` is the name of a Java class whose method `getFormula` returns an object that encodes above formula:

```
class BufferSpec extends Specification {
  public Formula getFormula() {
    return new Conjunction(
      new Always(new Atomic("NoLostMessage", null)),
      new Always(new Implication(
        new Negation(new Atomic("BufferEmpty", null)),
        new Eventually(new Atomic("BufferEmpty", null)))));
  }
}
```

When the debugger encounters the `assert` statement, it instructs the temporal logic checker (TLC) which is implemented in Java to dynamically load this class. TLC is called by the debugger after every subsequent macro-step to verify whether the state of the current collective breakpoint violates the asserted formula or not. The user can follow the checking process in a window that displays the status of the formula in every collective breakpoint: "false" means that the stated assertion has been violated by the current execution, "true" means that the assertion cannot be violated any more, "unknown" means that the assertion may be still violated in the future.

The formula `BufferSpec` refers to two atomic predicates `NoLostMessage` and `BufferEmpty` which are the names of C-functions which are located in a separate library that is dynamically loaded by the debugger. Whenever the TLC asks the debugger for the value of an atomic formula, the debugger executes the corresponding function which returns the truth value of the predicate in the current system state:

```
int NoLostMessage() {
  long number, countP, countC;
  number = getVarLongInt("number", getProcessIndex("Buffer"));
  countP = getVarLongInt("count", getProcessIndex("Producer"));
  countC = getVarLongInt("count", getProcessIndex("Consumer"));
  return number == countP-countC;
}

int BufferEmpty() {
  long number = getVarLongInt("number", getProcessIndex("Buffer"));
  return number == 0;
}
```

The atomic predicate functions can inspect the system state via an interface to the debugger. For instance, the function `getVarLongInt(`*var*`, `*proc*`)` returns the value of the program variable *var* in process *proc* as a value of type `long`. In this way, the predicate function `NoLostMessage` checks the number of messages in the buffer process with respect of the values of two counter variables in the producer process and in the consumer process.

Summarizing, for using in our system temporal formulas as runtime assertions, the programmer needs to

1. annotate the program to be debugged by the assertions[1],
2. provide a Java encoding of the temporal formulas,
3. provide C functions for the atomic predicates used in the temporal formulas.

This only reflects the current state of the system; in later versions, we plan to develop a meta-language where the Java encoding and the C functions are automatically generated from a high-level specification language.

[1] If the atomic predicates in an assertion refer to program labels, the program must be also annotated by such labels.

## 4 Temporal Assertions

We are now going to sketch the formal basis of using temporal formulas as runtime assertions. Any system can be described by a tuple $\langle is, ns \rangle$ where $is$ is the set of initial states of the system and $ns$ is the next state relation of the system. A temporal formula $F$ is valid for such a system, written as $\mathbf{T}[[F]] is\, ns$, if for every (finite or infinite) state sequence $s$ induced by $\langle is, ns \rangle$, $F$ holds at position 0 of $s$. Thus it suffices to define the truth value of a temporal formula $F$ at position $i$ of $s$, written as $\mathbf{T}[[F]]s\ i$:

$$\mathbf{T}[[\Box F]]s\ i = \text{true iff } \mathbf{T}[[F]]s\ j = \text{true for all } j \text{ with } i \leq j < |s|$$
$$\mathbf{T}[[\Diamond F]]s\ i = \text{true iff } \mathbf{T}[[F]]s\ j = \text{true for some } j \text{ with } i \leq j < |s|$$

Now let us introduce a "next step" formula $\circ_v F$

$$\mathbf{T}[[\circ_v F]]s\ i = \ \text{if } i+1 = |s| \text{ then } v \text{ else } \mathbf{T}[[F]]s\ (i+1)$$

which is true, if $F$ holds in the next step, and if no such step exists, takes the truth value $v$. We then define a semantics-preserving formula translation $\mathbf{G}[[F]]$

$$\mathbf{G}[[\Box F]] = \mathbf{G}[[F]] \wedge \circ_{\text{true}} \Box F$$
$$\mathbf{G}[[\Diamond F]] = \mathbf{G}[[F]] \vee \circ_{\text{false}} \Box F$$

such that in the result $G := \mathbf{G}[[F]]$ the operators $\Box$ and $\Diamond$ are always guarded by the $\circ_v$ operator. We can therefore reduce the validity of a temporal formula $F$ in a state sequence $s$ at position $i$ to the validity of atomic formulas in state $s(i)$ and to the validity of temporal formulas in $s$ at $i+1$.

Above definition is based on state sequences, but in assertion checking we only have access to the "current" state of the system. We therefore introduce a set of state trees $T(is, ns)$ induced by $\langle is, ns \rangle$. Each node in such a tree $t$ holds a state $t_{\text{state}}$, has a link $t_{\text{prev}}$ to its predecessor node, and a set of successor nodes $t_{\text{next}}$. The roots $r$ of these trees (the nodes with $r_{\text{state}} \in is$) have $r_{\text{prev}} = \top$; the leaves $l$ of these trees (for which no state $s$ exists such that $ns(l_{\text{state}}, s)$) have $l_{\text{next}} = \{\top\}$. We can now define the semantics $\mathbf{T}[[G]]t$ of a guarded formula $G$ with respect to such a tree $t$ such that the relationship to the original semantics is preserved, i.e., $\mathbf{T}[[F]]is\ ns = \mathbf{T}[[G]]\,T(is, ns)$.

However, during assertion checking, we only have access to a part of the tree referenced by a "current" state node whose children (the nodes of the successor states) may not yet be completely (or not at all) evaluated. We represent such "partial trees" by trees that contain "unknown subtrees" denoted by $\perp$ and extend the semantics $\mathbf{T}$ on complete trees to a semantics $\mathbf{T}_3$ on partial trees using a 3-valued logic with an additional logical value $\perp$ ("unknown").

The new semantics is compatible with the original one and monotonic with respect to a partial ordering $\sqsubseteq$ of trees according to their information content: $s \sqsubseteq t \Rightarrow \mathbf{T}_3[[G]]s \sqsubseteq \mathbf{T}_3[[G]]t$. We therefore have defined the semantics of a guarded temporal formula $G$ on the set of partial state trees induced by a system. While above explanation only describes temporal operators $\Box$ and $\Diamond$ which refer to the "future" of a state, our framework also supports corresponding operators which talk about the "past". The temporal logic checker TLC described in the following section implements $\mathbf{T}_3$ to determine the validity of an assertion $G$.

## 5 Checking Temporal Assertions with TLC in DIWIDE

The temporal logic checker TLC interacts with the DIWIDE debugger by a specified protocol. This protocol operates in a sequence of rounds that correspond to the states of an execution sequence. In each round,

1. TLC may receive from DIWIDE a new (additional) temporal formula whose validity is to be checked in the subsequent state,
2. TLC may ask DIWIDE questions about the truth values of atomic formulas in the current state (see Figure 2),
3. TLC announces its knowledge about the truth of the set of temporal formulas it has received up to now (true, false, unknown).

When a round has ended, DIWIDE informs TLC about the beginning of a new round (when a new state in the current execution sequence is available).

TLC evaluates in each round the truth of all formulas with respect to the round in which the corresponding formula has been submitted by the external partner. If a formula refers to the future, the result will be frequently "unknown". However, if more and more rounds are performed, the added knowledge may let the knowledge about such a formula change to "true" (the formula cannot be falsified any more after the current round) respectively "false" (the formula has been falsified by the current round). If a formula has been falsified, the corresponding assertion has been violated.

TLC does not repeatedly evaluate (sub)formulas whose final value ("true" or "false") is already known. Such results are cached such that only those formulas are re-evaluated whose values are not yet known but are required to determine the value of the overall formula. To support temporal "past operators", TLC prefetches in each round the values of all atomic predicates whose results may be required in the future to evaluate a "past formula". Whenever such a formula is submitted, TLC records the atomic predicates in the scope of such operators in order to start prefetching the corresponding values.

## 6 Future Work

With TLC and DIWIDE it is possible to assert temporal formulas and have their validity checked in (manually or automatically) selected runs of a parallel program. However, we do not yet provide an adequate graphical user interface which allows the programmer in an intuitive way to determine the fundamental reason why an assertion has failed respectively is not yet satisfied. Second, we are still lacking a high-level specification language from which the low level encodings of temporal formulas (as Java objects) and atomic predicates (as C functions) are automatically generated. Third, and most important, we need to evaluate by larger program examples with interesting properties to which extent the use of temporal assertions actually helps to improve the understanding of program behaviors and detect errors in them. In any case, the presented system will serve as a good starting point for these investigations on the usefulness of extending a parallel debugger with model checking capabilities.

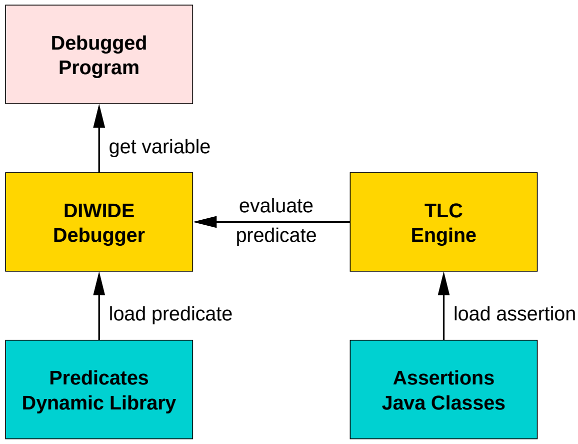


**Fig. 2.** Integrating TLC with DIWIDE

## References


1. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
2. J. Cuny et al. The Ariadne Debugger: Scalable Application of Event-Based Abstraction. *SIGPLAN Notices*, 28(12):85–95, December 1993.
3. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification, 7th International SPIN Workshop*, volume 1885 of *LNCS*, pages 323–330, Stanford, CA, August 30 - September 1, 2000. Springer.
4. J. Hakansson. Automated Generation of Test Scripts from Temporal Logic Specifications. Master's thesis, Uppsala University, Sweden, 2000.
5. A. Hough and J. Cuny. Initial Experiences with a Pattern-Oriented Parallel Debugger. *SIGPLAN Notices*, 24(1):195–205, January 1988.
6. P. Kacsuk. Systematic Macrostep-by Macrostep Debugging of Message Passing Parallel Programs. *Future Generation Computer Systems*, 16(6):609–624, 2000.
7. P. Kacsuk, R. Lovas, and J. Kovács. Systematic Debugging of Parallel Programs in DIWIDE Based on Collective Breakpoints and Macrosteps. In P. Amestoy et al., editors, *5th Euro-Par Conference*, volume 1685 of *Lecture Notes in Computer Science*, pages 90–97, Toulouse, France, August 31 – September 3, 1999. Springer.
8. D. Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, Johannes Kepler University, September 2000.
9. D. Kranzlmüller and J. Volkert. NOPE: A Nondeterministic Program Evaluator. In *Parallel Computation, 4th International ACPC Conference*, volume 1557 of *LNCS*, pages 490–499, Salzburg, Austria, February 16–18, 1999. Springer.
10. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems — Specification*. Springer, Berlin, 1992.
11. S. Shende et al. Event- and State-based Debugging in TAU: A Prototype. In *ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 21–30, Philadelphia, PA, May 1996.