On-the-Fly Model Checking of Program Runs for Automated Debugging

M. Frey
Institut für Informatik
Technische Universität München
Orleansstr. 34 D-81 667 Munich, Germany

B.-H. Schlingloff
Technologie–Zentrum Informatik (TZI–BISS)
Universität Bremen
Bibliothekstr. 1, D-28 359 Bremen, Germany

Abstract

In this paper, an on-the-fly algorithm is developed for model checking of temporal logic safety properties on partially ordered occurrence net structures. This algorithm is used for the automated debugging of parallel programs. During the monitoring of a program run, a state action net is constructed from the program trace. Temporal specifications are evaluated on-the-fly with respect to this net. The specifications can express e.g. that an error has occurred, or that certain control locations have been reached. When the specified condition can occur, the execution is halted. Since we use a partial order logic, specification violations can be detected even if they did not actually occur in the particular interleaving of the program run.

1 Introduction

Model checking is an automated technique which has successfully been used to verify small and medium sized finite-state systems. Because of the state explosion problem, verification of large parallel and distributed programs is often impossible. Generally, the number of reachable states is exponential in the number of state variables of the system. Especially, often the size of the input space of a parallel or distributed algorithm is too large for a complete traversal. The input space is defined by the number of input variables and their ranges. When a system has n input variables, each of which has domain D, then the size of the input space is $|D|^n$. For example, in a sorting algorithm for a field of 100 integer variables, where each integer is represented by 32 bits, the size of the input space is $(2^{32})^{100} \simeq 10^{1000}$. Even with symbolic methods, automatic verification of the unabstracted sorting algorithm is not possible.

A common practice to increase the reliability of such algorithms is testing. Several commercial tools exist for the systematic generation and execution of test cases. In these tools, the expected result of each test case usually is specified by a table. For parallel and distributed reactive systems this is not appropriate, since the correctness not only depends on the functional value, but also on the relative order of events. Recently, a number of researchers [12, 4, 13] have proposed testing based on *formal specifications*. In this approach, for each test case the result of the program run is checked whether it matches a formal description in some specification language.

The "result" of a reactive program run often is modelled as a finite or infinite sequence of events. For parallel programs, it is sometimes more appropriate to model the program behaviour as a *partial order*. Model checking can be used to determine whether a specific property formulated in an appropriate temporal logic holds for the given program run. This checking can either be done after termination or abortion of the program, or "on-the-fly" during the execution.

Several authors [3, 2] proposed to use formal specifications also for the *debugging* of parallel programs. In contrast to testing, during the debugging process it is possible to observe and modify internal variables of the system. Model checking can be used to determine the possible causes of an error. Hypotheses about the causes of an error are specified in temporal logic. If the hypotheses are not satisfied, a counter example is generated to explain why the specification fails.

A first implementation of this idea was integrated by Harter, Heimbigner and King in the debugging tool IDD [11]. Requirements on the sequence of events of handling a common communication device of a distributed system were specified in a temporal interval logic. These requirements were automatically checked during runs of the system. Goldszmidt, Yemini and Katz [10] used a linear time temporal logic for the specification of debugging assumptions. Events occurring during a run of the system are recorded together with a time stamp. Then the linear sequence of events, where the events are totally ordered by their time stamp, is used for model checking.

Both of the above approaches use a linear representation of parallel system runs. The ordering of events by time stamps introduces dependencies between events, which are not inherent in the run. For example, assume that the specification says that event A should happen before event B. If there are causally independent occurrences of A and B in a run, where by chance A's time stamp smaller than that of B, then a linear time approach will not detect the possible error of B happening before A.

Therefore, Garg and Waldecker [9] use partially ordered models to debug distributed program runs. They propose an efficient algorithm for automatic model checking of a limited temporal logic during the program run. Thus, some errors can be detected even if they do not occur in the arbitrary interleaving of independent events during the program run. Frey and Weininger [8, 5] and Frey and Oberhuber [7] extend this approach to the full partial order temporal logic of Reisig[14]. The corresponding models are special kinds of finite causal nets, and the logic describes properties of global states of these models. Since models must be finite, model checking is applied "post mortem". To debug nonterminating reactive programs, the execution is aborted after some random time interval.

In this paper, we extend the results of [5] to allow model checking of (potentially) infinite runs "on-the-fly". Thus, the debugging process can run in parallel to the system under development. Execution is halted when an error is found, and variable values can then be inspected and modified. We give conditions for formulas which can be checked during the program run, and develop an on-the-fly algorithm for the evaluation of these formulas on partial order models which are generated by the program run.

The paper is organized as follows: Section 2 gives a short overview of the basic definitions and results of [5]. In Section 3, we exhibit conditions for the monitoring process and properties for the temporal logic specifications such that the debugging process can be conducted on-the-fly. In Section 4, we describe a new model checking algorithm for this on-the-fly debugging of program runs. Finally, Section 5 summarizes the results and gives a short outlook on further work.

2 Debugging by Model Checking

Figure 1 shows an overview of our approach [5] for checking whether an execution of a parallel program satisfies a temporal logic specification.

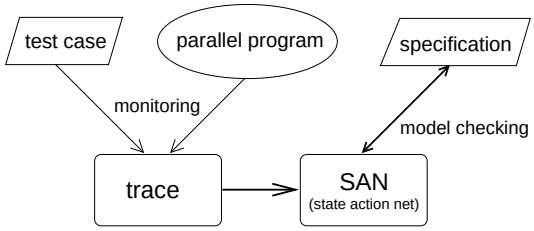

Figure 1. Specification-based debugging

We are given a parallel program, a test case of the program, and a temporal logic specification of properties for this test case. We execute the program and record a *trace* of this execution. Whenever during the test run synchronizations, communications, or accesses to variables are taking place, whenever executions of methods are starting or ending, and whenever new threads are generated or terminated, informations about these events is included in the trace. If the execution of the program is nonterminating, we abort it as soon as the trace reaches a certain length. The trace is then used to generate a special causal net called *state action net*, which is an abstract model of the program run. We then apply model checking to determine whether the specification is satisfied by the state action net. If we find that the specification is not valid, we exhibit the events which lead to the fatal situation.

The construction of a partial order model from the linear trace gives an important advantage. Only those dependencies between threads are included in the model which represent synchronizations in the program run. Thus, errors can be detected even if they did not actually occur in the particular scheduling or interleaving of the program run.

2.1 State Action Nets

State action nets (SANs) are a special kind of finite causal nets, consisting of nodes and transitions. In SANs, transitions are called *actions*. Each action represents an atomic step in the program, i.e., the execution of a single program statement or a set of synchronized program statements. SAN nodes are called *local states*. Each local state is a description of one thread of control at a particular moment in the execution of the program. It contains the identification of the thread, the name of the method executed at the preceding action, and the values of variables in this thread.

Formally, a state action net is a tuple $N = (S, A, R)$, where

- S is a nonempty and finite set of local states,
- A is a nonempty and finite set of actions, and
- $R \subset (S \times A) \cup (A \times S)$ is the causal dependency relation.

The causal dependency relation satisfies the following conditions:

- the transitive closure of R is acyclic, and
- the preset and postset of each local state contains at most one action.

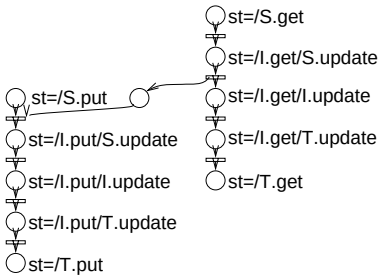

Figure 2. A state action net

Figure 2 shows an example of a state action net. At each local state the executed methods are annotated. The variable st contains all executed methods in the order in which they

have been called. The prefixes "S." and "T." indicate that the execution of a method is starting or terminating, respectively. Otherwise, the prefix is "I.".

State action nets can be generated from traces, which are recorded during a particular program run. Each traced event consists of an *event name* and a set of *parameters*. In order to relate an event to a specific thread, each event contains the identification of the thread as a parameter. Further parameters describe dependencies between pairs of events of different threads. For example, the generation of a thread can be described by two events: the event `create_thread(x.i)` for the creator, and the event `new_thread(x.i)` for the created thread. Both events contain a parameter `x.i` which uniquely identifies this generation of threads. This unique identification is done by the thread identification `x` of the creator and a unique number `i` with respect to the creator.

Other events are traced whenever shared variables are generated, deleted, read or written. Synchronizations are also described by read and write access to shared variables, together with an event describing that the read access was done during examination of a waiting condition. Furthermore, there are events which describe the entering and leaving of method bodies, and events to describe the beginning of the execution of a new statement.

SANs are generated from traces by examining the traced events and reconstructing from them causal dependencies in the program run. Sequential dependencies between actions and local states within a single thread are generated according to the order in which they occur in the trace. Dependencies between actions of different threads are generated by synchronizing the traced events of both threads according to the causal dependency within the program run. For example, each event which describes the reading of a variable value is causally dependent on the event where this value was written. Thus, in the SAN a dependency is introduced between the corresponding actions.

Since the SAN of a program run is generated by a linear traversal of the program trace, it can be generated on-the-fly, during the program run. More information about traces and the generation of SANs can be found in [7].

2.2 Temporal Logic

Our temporal logic provides means to express local properties of single threads in an execution, as well as temporal dependency operators. For the intended application in debugging, we extended classical temporal logic in three ways:

- The logic consists of two tiers: a local and a global tier.
  - The local tier describes requirements on local states. We can express that variable values satisfy specific predicates and relations within a local state, for example, that $x = 3$ or $x \leq y$. Moreover, the predicate **in.**m expresses that a methods m is executed in the action preceding the local state, and the starting state and the terminating state of the execution of m can be distinguished by the predicates **start.**m and **term.**m, respectively.
    Relational terms and predicates can be combined using the logical operators **not** and **and**.
  - The global tier has formulas of the local tier as atomic formulas. In addition to boolean operations it contains the temporal connectives **next**, **sometime** and **until**.
- To handle unknown variable values we use a three-valued interpretation. A formula containing a variable whose value is not visible in a certain state has the truth value *undefined*. The semantics of boolean operations is given by Lukasiewicz' classical three-valued interpretation of boolean algebra[15].
- To describe requirements of programs with dynamic generation of threads, we use *thread identifier variables* (TIVs) as placeholder for threads. Each atomic formula of the global tier is prefixed with a TIV. Formulas can be relativized by equality and other conditions between TIVs: **(**p **if** $cond(t,t')$**)**. The semantics of TIVs is defined by a *thread identifier variable assignment* (TIVA) which assigns a thread $\mathfrak{T}(t)$ in the program run to every TIV t.

In section 3 we introduce on-the-fly model checking of program runs with our temporal logic. Therefore, we give the formal definition of the three-valued semantics of the global tier. A *slice* is a maximal set of local states which are not causally ordered. The step-relation between slices is defined by firing of transitions. Formally, the global model of an SAN N is defined by $G_N = (\mathfrak{S}, \mathfrak{R}, \mathfrak{s}_I, \mathfrak{s}_T)$, where $\mathfrak{S}$ is the set of all slices of N, and $\mathfrak{R} = \mathfrak{S} \times \mathfrak{S}$ describes the firing of actions. Here $(l, l') \in \mathfrak{R}$, if an action a exists, which is activated in l and l' is generated from l by firing a. The initial slice is $\mathfrak{s}_I$, and the terminal slice is $\mathfrak{s}_T$. The existence of initial and terminal slice is guaranteed since the slice-graph is a finite lattice.

For a given slice l and TIVA $\mathfrak{T}$, a formula p of the global tier can be either *true* ($(l, \mathfrak{T}) \models p$), or *false* ($(l, \mathfrak{T}) \not\models p$) or *undefined* ($(l, \mathfrak{T}) \uplus p$). Formally, these three relations are defined as follows:

- $(l, \mathfrak{T}) \models t : p_l$, if a local state $s \in l$ exists where $\mathfrak{T}(t) = tid(s)$, and $s \models p_l$; $(l, \mathfrak{T}) \not\models t : p_l$, if $s \in l$ exists where $\mathfrak{T}$**(***t***)**$= tid(s)$, and $s \not\models p_l$; $(l, \mathfrak{T}) \uplus t : p_l$, otherwise.
- $(l, \mathfrak{T} \models$ **not** p if $(l, \mathfrak{T}) \not\models p$; $(l, \mathfrak{T}) \not\models$ **not** p if $(l, \mathfrak{T}) \models p$; $(l, \mathfrak{T}) \uplus$ **not** p, otherwise.
- $(l, \mathfrak{T}) \models$ **(**p **and** q**)** if $(l, \mathfrak{T}) \models p$ and $(l, \mathfrak{T}) \models q$; $(l, \mathfrak{T}) \not\models$ **(**p **and** q**)** if $(l, \mathfrak{T}) \not\models p$ or $(l, \mathfrak{T}) \not\models q$; $(l, \mathfrak{T}) \uplus$ **(**p **and** q**)**, otherwise.
- $(l, \mathfrak{T}) \models$ **next** p if a slice l' exists where $(l, l') \in \mathfrak{R}$ and $(l', \mathfrak{T}) \models p$; $(l, \mathfrak{T}) \not\models$ **next** p, otherwise, i.e., if for all slices l' with $(l, l') \in \mathfrak{R}$ either $(l', \mathfrak{T}) \not\models p$ or $(l', \mathfrak{T}) \uplus p$.
- $(l, \mathfrak{T}) \models$ **sometime** p if a slice l' exists such that $(l, l') \in \mathfrak{R}^+$ and $(l', \mathfrak{T}) \models p$; $(l, \mathfrak{T}) \not\models$ **sometime** p, otherwise.

- $(l, \mathfrak{T}) \models$ **(**p **until** q**)** if a slice l' exists where $[(l, l') \in \mathfrak{R}^+$, and $(l', \mathfrak{T}) \models q]$, and for all slices l'' where $[(l, l'') \in \mathfrak{R}^+$ and $(l'', l') \in \mathfrak{R}^+]$, either $(l'', \mathfrak{T}) \models p$, or $(l'', \mathfrak{T}) \models\!\!\!\uplus\, p$; $(l, \mathfrak{T}) \not\models$ **(**p **until** q**)**, otherwise.
- $(l, \mathfrak{T}) \models$ **(**p **if** $cond$(t,t')**)** if $(l, \mathfrak{T}) \models p$ and $cond(\mathfrak{T}(t), \mathfrak{T}(t'))$; $(l, \mathfrak{T}) \not\models$ **(**p **if** $cond$(t,t')**)** if $(l, \mathfrak{T}) \not\models p$ and $cond(\mathfrak{T}(t), \mathfrak{T}(t'))$; $(l, \mathfrak{T}) \models\!\!\!\uplus$ **(**p **if** $cond$(t,t')**)**, otherwise.

According to this definition, all formulas **next** p, **sometime** p or **(**p **until** q**)** are either true or false in any slice l and TIV $\mathfrak{T}$. A formula p is defined to be *valid* in an SAN N, if **always** p is true in the initial slice of N, where **always** p is **not sometime not** p. Since the truth value of **always** p in any slice is not *undefined*, the notion of validity in state action nets is two valued: A formula p is valid in an SAN N if for all l and $\mathfrak{T}$ of N either $(l, \mathfrak{T}) \models p$ or $(l, \mathfrak{T}) \models\!\!\!\uplus\, p$.
For example, consider the formula

(($t1$**:(in.***get* **and term.***update***) before**
$t2$**:(in.***put* **and start.***update***)) if** $t1 \neq t2$**)**

Here **(**p **before** q**)=not (not** p **until** q**)**. The formula specifies a precedence order on executions of the method *update* which has to be true if *update* is executed in parallel by several threads. It turns out that this formula is not valid in the SAN of Figure 2. In the next section, we describe an algorithm for automatically checking such properties.

2.3 Model Checking

The model checking procedure of [5] is a global and bottom-up evaluation of formulas. The checking is done by calculating all slices and TIVAs in which the formula is false. The formula is valid if the result is empty; otherwise the calculated slice and TIVA indicates where the formula is not satisfied. Since it is difficult to calculate slices and TIVAs in which subformulas of the given formula are undefined, we avoid the negation during model checking by further transformation of the input formulas. We use the dual temporal operators **or**, **all_next**, **always**, and **before** and replace negated subformulas by the respective dual formulas. For example, **not sometime** p is replaced by **always not** p.

The slices are calculated bottom-up using the syntactical structure of the negated and transformed formula. This syntactical structure can be described by the syntax tree. To avoid the calculation of slices in which subformulas are undefined, we slightly change the syntax tree into a *formula tree*. The difference between both trees is that a child node of a node labelled by **always** p, **all_next** p or **(**p **before** q**)** is labelled by the transformed formula logically equivalent to **not** p. Leaf nodes of the formula tree may contain arbitrary formulas of the local tier. Figure 3 shows the formula tree of our example formula.

In [5], model checking is done "post mortem", after generating the whole SAN of the program run. Therefore, the formula tree can be evaluated bottom-up. All slices and TIVAs of the child nodes of a node in the formula tree have to be calculated before the slices and TIVAs of a node can be calculated. To do so, we need the global structure of the net. The generation of the SAN can be done on-the-fly during the program run. For checking infinite runs on-the-fly, only part of the global structure is available. Therefore it is necessary to restrict the logic appropriately.

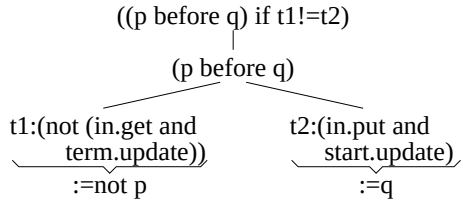

Figure 3. A formula tree

3 Safety Requirement

We want to extend the post mortem model checking approach to reactive programs with potentially infinite executions. We propose to execute the program an unlimited amount of time and to check the validity of the formula for this run on-the-fly. This means that model checking can be done simultaneously during the program run. If a slice and a TIVA is generated for the root node of the formula tree, the run is stopped and user interaction can take place. Unfortunately, not all properties can be checked in this manner.

As described in subsection 3.1, SANs can be generated on-the-fly. For our purposes, this creation has to satisfy the following two properties: Firstly, the information in local states must be completed before successors of the local state are generated. Otherwise, the valuation of formulas of the local tier may change depending on the changing values of the state. Thus, we could not guarantee that properties which are false will be remain false during all continuations of the run. Secondly, any continuation of an execution must lead to an extension of the SAN, in which all parts of the previously generated net are unchanged. Otherwise, new slices could be introduced in the old part of the net. This could lead to a change in the sequences of slices of the old part which could affect the valuation of formulas.

Both conditions on the generation of nets can be fulfilled in the generation of state action nets, if all traced variable values belong to different threads of the program. This condition is trivially satisfied in message passing programs. In shared memory programs, we can trace "local copies" of variables shared between several threads.

In addition to the conditions on SANs, we now exhibit a condition for the properties which are to be model checked on-the-fly. Only those properties can be examined which, after they are found to be false in a slice and a TIVA during part of the run, can not become true by the same slice and TIVA

in a continuation of this run. A formula p is called a *safety property*, if for every SAN N,

$$N \models p \text{ if } (\forall M < N)(\exists N' > M)\ N' \models p$$

In this definition (which originates from [1]), M is an initial subnet of N, and N' is any continuation of this subnet M. In other words, p is a safety property if for every net *not* satisfying p there is a slice and TIVA such that $(l, \mathfrak{T}) \not\models p$ and there is no extension N' such that $(l, \mathfrak{T}) \models p$ in N'. Stated differently, for every net dissatisfying p something "bad" must have happened after some finite amount of time which cannot be remedied by any future good behaviour.

For linear and branching time temporal logics, there are various syntactical characterizations of safety and liveness properties. However, these characterizations are not appropriate for our logic, since it is interpreted on partial order structures.

All formulas describing properties of finite parts of an infinite computation are safety-properties. If the length of the finite part is bounded by the property, then only those slices of a net extension have to be checked for which a succeeding sequence of slices exists which is bounded by the property. In particular, each formula of the local tier is a safety property.

If p is a safety property, then the formula **always** p, which is equivalent to **not sometime not** p, is a safety property. In general, **sometime** p is not a safety property: if in a net no slice exists in which p is satisfied, then **sometime** p is false in the initial slice. However, if an extension is constructed in which p is true, then **sometime** p is true in the initial slice. Similarly, the formula **(**p **before** q**)** is a safety property, but **(**p **until** q**)** is not a safety property.

As an example for a formula p such that neither p nor **not** p is a safety property, consider $p =$ **sometime (**q **and always** q**)**. Since **sometime** q is not a safety property, p is not a safety property. The formula **not** p **= always (not** q **or sometime not** q**)** is also not a safety property: **always sometime** q may be true in the initial slice of a net extension, because the terminal slice of the extension is the only one satisfying **not** q. In all initial subnets, **always (not** q **or sometime not** q**)** is false, because in all of its slices q is satisfied.

To describe the formulas specifying safety properties, we characterize the set of formula trees of the negated and transformed formulas. A negated and transformed formula in the formula tree is called *slice-stable*, if for every SAN N, for every slice l and for every TIVA $\mathfrak{T}$ of N

$$(l, \mathfrak{T}) \models_N p \text{ if } \forall M > N (l, \mathfrak{T}) \models_M p$$

The relation $\models_N$ is defined with respect to the SAN N; $\models_M$ is defined with respect to an extension M of N. Formulas **(**p **and** q**)**, **(**p **or** q**)**, **next** p, **all_next** p', **sometime** p and **(**p' **until** q**)** are slice-stable if p and q are slice-stable and p' specifies a property of a finite part of the SAN. If p is slice-stable, then its negation specifies a safety property. Thus, slice-stability is a sufficient criterion for the algorithm described in the next section to be applicable.

4 On-the-Fly Model Checking

In this section, we give a parallel algorithm for model checking safety properties on-the-fly. The evaluation of the specification can run in parallel with the program to be debugged, and independent subformulas can be evaluated in parallel.

In principle, the bottom-up model checking procedure described in section 2.3 could be applied. Whenever a new subnet is generated during the construction of the SAN from a trace, we could evaluate the formula tree from scratch on this subnet. However, with this procedure, for each net extension and each node of the formula tree all slices and TIVAs calculated in the previous net extension would have to be recalculated. This is not necessary: we can reuse the information from the previous subnet. For each extension and formula-tree node we have to deal only with those slices, which arise from the addition of further local states to the net.

For each node and each new slice we have to determine whether the respective subformula is false in the slice. In contrast to the post-mortem evaluation of the SAN, slices and TIVAs of all children of a node can be calculated at the same time. Conceptually, all nodes of the formula tree can be examined independently and, on a multiprocessor system, in parallel. This can be done by starting a thread for each node which calculates the slices of this node. To avoid recalculation, for each node we use a queue of slices and TIVAs, in which they are inserted in a consecutive order.

If all nodes are processed independently, it is necessary to synchronize the evaluation. Version numbers are assigned to each newly generated net extension. For each node, evaluation of the slices of a new net extension can begin when all child nodes have finished the calculation of these slices.

The following method describes the synchronization protocol for each node:

```
thread calc_slices(n)
⌜version := 0;
 while (not san.terminated and
     version<san.last_version)
 ⌜await(version < san.version);
  await(version < n.childs.version);
  calc_operation(n,version+1);
 ⌞⌞version++;
```

For each node n a new thread is generated executing the method *calc_slices*. The object *san* contains the state action net, with a boolean variable *terminated* which indicates when the program run is interrupted or terminated. In this case the variable *last_version* contains the final version number of the

program run. The variable *version* contains the version number of the last generated net extension.

The first statement of the loop waits for the generation of a new net extension. The second statements enforces waiting until all child nodes have finished calculating the slices of the new net extension. Then, the method specified by the logical operator in the node is called. Additional slices and TIVAs of the net extension defined by the value of *version*$+1$ are calculated. If a slice and TIVA falsify the subformula of a node, then they are inserted in the corresponding queue. Finally, *version* is incremented to signal the parent node that the slices of the net extension are calculated for the node n.

In the following we describe the methods which are called by $calc\_slices$ to calculate the new value of the queue of a node. For space reasons, cases **sometime** p, **always** p, **(**p **before** q**)**, and **(**p **if** $cond(t,t')$**)**, which are similar to other cases, are omitted.

4.1 Nodes labelled with atomic formulas

Leaf nodes of the formula tree are labelled with atomic formulas. When the method *calc_slices* is called for a leaf node, there are no waiting conditions for the child nodes. After a new net extension is generated, the method *calc_leaf* is called with node and version number of the net extension as parameters.

```
calc_leaf(n, version)
⌜for all s ∈ san.new_states[version] do
  for all s' ∈ satisfying_states do
  ⌜if (s || s') then
   ⌜sl := states2slices({s, s'}, san.all_states[version]);
    for all l ∈ sl do
      n.queue[version] := n.queue[version]∪
  ⌞       {(l, make_tiva(s'.tid, n.var))};
   if check_ll(s) then
   ⌜new_sat_states := new_sat_states ∪ {s}
    sl := states2slices({s}, san.all_states[version]);
    for all l ∈ sl do
      n.queue[version] := n.queue[version]∪
  ⌞       {(l, make_tiva(s.tid, n.var))};
  ⌞satisfying_states := new_sat_states;
⌞
```

New slices and TIVAs can arise from two different sources. Firstly, new slices have to be built if the extension contains a state which is concurrent (not causally ordered) to a local state of a previous extension. Secondly, if the extension contains a state in which the local formula of the node is satisfied, all slices with this state have to be added. Therefore, the leaf node contains a variable *satisfying_states* which contains all states of previous extensions in which the formula of the node is satisfied. The variable *san.new_states* is an array of sets of states containing the new states of all extensions. The variable *san.all_states* is also such an array of sets of states containing all local states of a version. The method *states2slices* calculates all slices which must contain the states of the first parameter, and all other states of the slices are states of the second parameter. The method *make_tiva* generates the TIVA which assigns to the first parameter a TIV of the second parameter. n*.var* is the set of TIVs of the node n, and s'*.tid* is the thread identification of the local state s'. Method *check_ll*(s) checks whether in the local state of the parameter the formula of the node is satisfied.

4.2 Nodes labelled with (p and q)

Slices of a node labelled with **(**p **and** q**)** are calculated by an intersection of slice sets of both nodes. This intersection has to take care of the TIVA associated with a slice.

```
simple_and(var_p, var_q, sl_p, sl_q, sl_res)
⌜for all (l, 𝔗) ∈ sl_p do
  for all (l', 𝔗') ∈ sl_q ) do
   if l = l' and comp(𝔗, 𝔗', var_p ∪ var_q) then
   ⌜𝔗" = merge(𝔗, var_p, 𝔗', var_q);
   ⌞sl_res := sl_res ∪ (l, 𝔗");
⌞
```

The parameters of *simple_and* are the set of variables and the set of slices and TIVAs of both child nodes. To calculate the intersection, it has to be checked whether the TIVA of the slice of one child node and the TIVA of the slice of the other child node can be merged together. This is similar to the calculation of a most general unifier in automated theorem proving. We have to check that no TIV exists to which both TIVAs assign a different thread identification. This check is done by the method *comp*. The merging of TIVAs is done by the method *merge*.

For an on-the-fly generation of the slices of nodes labelled with **(**p **and** q**)** it is not sufficient to calculate the intersection of the new slices of one extension. One of the child nodes could be the node of a temporal operator. A new slice can be generated for a new extension even if the slice itself did occur in prior extensions. As an example, consider the formula **sometime** p: If a slice of the new extension satisfies p, all slices which are predecessors of it satisfy **sometime** p. Some of these slices can also be slices of prior extensions. This leads to the following method for an on-the-fly calculation:

```
calc_and(n, version)
⌜n.child_p.slice_set := n.child_p.slice_set∪
     n.child_p.queue[version];
 n.child_q.slice_set := n.child_q.slice_set∪
     n.child_q.queue[version];
 simple_and(n.child_p.var, n.child_q.var,
     n.child_p.queue[version], n.child_q.slice_set, sl);
 simple_and(n.child_p.var, n.child_q.var,
     n.child_p.slice_set, n.child_q.queue[version], sl1);
⌞n.queue[version] := sl ∪ sl1;
```

The method consists of two symmetrical parts, where each part calculates the intersection between the new slices of one node and all slices of the other node. It can be easily generalized to deal with arbitrary finite conjunctions.

4.3 Nodes labelled with (p or q)

The slices and TIVAs of a node labelled with **(p or q)** are the union of the slices and TIVAs of the respective child nodes. To calculate this union, TIVAs have to be generated which assign TIVs to each variable of the respective subformula. The union is calculated by the following method:

$simple\_or(var_p, var_q, tiva_p, tiva_q, sl_p, sl_q, sl_{res})$
for all $(l, \mathfrak{T}) \in sl_q$ **do**
  for all $\mathfrak{T}' \in tiva_p$ **do**
    if $comp(\mathfrak{T}, \mathfrak{T}', var_p \cap var_q)$ **then**
      $\mathfrak{T}" = merge(\mathfrak{T}, var_p, \mathfrak{T}', var_q)$;
      $sl_{res} := sl_{res} \cup (l, \mathfrak{T}")$;
for all $(l, \mathfrak{T}) \in sl_p$ **do**
  for all $\mathfrak{T}' \in tiva_q$ **do**
    if $comp(\mathfrak{T}, \mathfrak{T}', var_p \cap var_q)$ **then**
      $\mathfrak{T}" = merge(\mathfrak{T}, var_p, \mathfrak{T}', var_q)$;
      $sl_{res} := sl_{res} \cup (l, \mathfrak{T}")$;

The parameters $tiva_p$ and $tiva_q$ contain the possible TIVAs for the formula p and q of both child nodes. For on-the-fly model checking the following sources for new slices and TIVAs have to be taken into account: First, new identifications for thread identifiers can occur in the new net extension. TIVAs of prior extensions in one child nodes have to be merged with TIVAs containing at most one new thread identification. Second, new slices of the new extension in the child nodes may exist. For these slices the union has to be calculated. The following method takes both sources into account:

$calc\_or(n, version)$
$tiva_p = generate\_tiva(san.new\_tiv[version],$
    $san.all\_tiv[version-1], n.child_p.var)$;
$tiva_q = generate\_tiva(san.new\_tiv[version],$
    $san.all\_tiv[version-1], n.child_q.var)$;
$simple\_or(tiva_p, tiva_q, n.child_p.slice\_set,$
    $n.child_q.slice\_set, sl)$;
$tiva_p = generate\_tiva(empty, san.all\_tiv[version],$
    $n.child_p.var)$;
$tiva_q = generate\_tiva(empty, san.all\_tiv[version],$
    $n.child_q.var)$;
$simple\_or(tiva_p, tiva_q, n.child_p.queue[version],$
    $n.child_q.queue[version], sl1)$;
$n.queue[version] := sl \cup sl1$;
$n.child_p.slice\_set := n.child_p.slice\_set \cup$
    $n.child_p.queue[version]$;
$n.child_q.slice\_set := n.child_q.slice\_set \cup$
    $n.child_q.queue[version]$;

The variable *san.new_tiv* is an array of sets of thread identifications which contains for each net extension the new thread identifications. Likewise, *san.all_tiv* is an array of sets of thread identifications which contains all thread identifications of each net extension. The first part of the method calculates the slices and TIVAs induced by new thread identifications. The second part of the method calculated the slices and TIVAs induced by new slices calculated for the child nodes.

4.4 Nodes labelled with next p

The slices of a node labelled with **next** p are calculated by generating the predecessors (w.r.t. $\mathfrak{R}$) of the slices of the child node. For on-the-fly model checking, new slices of the child node are the only source of new slices in the parent node. This holds since the TIVA of the predecessor is the same as that of the slice of the child node. Furthermore, new extensions cannot insert new actions which are predecessors of a local state in a slice from a prior extension. Assuming that the predecessors of a slice are calculated by the method *pred_slices*, on-the-fly model checking is performed by the following method:

$calc\_next(n, version)$
for all $(l, \mathfrak{T}) \in n.child.queue[version]$ **do**
  for all $l' \in pred\_slices(l)$ **do**
    $n.queue[version] := n.queue[version] \cup (l', \mathfrak{T})$;

4.5 Nodes labelled with all_next p

To calculate the slices and TIVAs for **all_next** p, the complement of the predecessors of the slices of the child node satisfying **not** p has to be calculated.

$simple\_all\_next(sl_{all}, tiva_{all}, sl, sl_{res})$
for all $(l, \mathfrak{T}) \in sl_{all} \times tiva_{all}$ **do**
  $C := true$;
  for all $(l', \mathfrak{T}') \in sl$ **do**
    if $\mathfrak{T}' = \mathfrak{T}$ **then**
      for all $l'' \in pred\_slices(l')$ **do**
        if $l = l'$ **then** $C := false$;
  if C **then** $sl_{res} := sl_{res} \cup (l, \mathfrak{T})$

The parameters sl_{all} and $tiva_{all}$ contain all slices and all TIVAs. *sl* contains the slices and TIVAs of the child node. When calculating the difference on-the-fly, it can not be calculated for the whole extension because of temporal relations of temporal operators. For example, assume that the subformula to be checked is **all_next** p, where p is an atomic formula. In this case only those slices can be examined, which contain no local state which is also a local state of the terminating slice of the net extension. Otherwise, the next extension could contain a successor slice l of a slice in the difference of the actual net extension w.r.t. $\mathfrak{R}$ such that p is satisfied in l. Therefore, the method *calc_all_next* has an additional parameter *length* which depends on the formula tree and describes how many successor slices which do not contain any

local state together with the terminating slice, a slice has to have to be included in the result.

Two sources for new slices and TIVAs exist. Firstly, extending the net can lead to slices which do not satisfy the above *length*-condition on slices for prior extension but do satisfy the condition for the new extension. Secondly, new thread identifications can lead to new TIVAs for which all slices of prior extension are slices and TIVAs of the node. Both sources are calculated as follows:

$calc\_all\_next(n, version, length)$
$\ulcorner n.child.slice\_set := n.child.slice\_set \cup$
$\quad n.child.queue[version];$
$sl := generate\_slices(san.final\_slice[version-1],$
$\quad san.final\_slice[version], length);$
$tiva = generate\_tiva(empty, san.all\_tiv[version],$
$\quad n.var);$
$simple\_all\_next(sl, tiva, n.child.slice\_set, sl1);$
$sl := generate\_slices(\mathfrak{s}_I, san.final\_slice[version],$
$\quad length);$
$tiva = generate\_tiva(san.new\_tiv[version],$
$\quad san.all\_tiv[version-1], n.var);$
$simple\_all\_next(sl, tiva, n.child.slice\_set, sl2);$
$\llcorner n.queue[version] := sl1 \cup sl2;$

4.6 Nodes labelled with (p **until** q)

There is only one source of new slices for a node labelled **(**p **until** q**)**: The slices l and TIVAs $\mathfrak{T}$ of a new extension of the child node labelled with q can lead to new slices which are a subset of the predecessors of l w.r.t. $\mathfrak{R}^+$.

The subset is given by all slices l' and TIVAs $\mathfrak{T}'$ where $\mathfrak{T}'$ is a TIVA for all TIVs of the node and it is an extension of $\mathfrak{T}$. Furthermore, no slice l'' exists between l and l' such that $(l'', \mathfrak{T}')$ is an element of the child node labelled with **not** p. To check this condition, it is not necessary to know the slices and TIVAs of the child node labelled with **not** p of further net extensions, because only those slices of the child node which are predecessors of l are necessary. New predecessors of l cannot be generated in further net extensions because of the conditions on state action nets stated in section 3. The following method calculates the slices and TIVAs of a node labelled with **(**p **until** q**)** on-the-fly:

$calc\_until(n, version, length)$
$\ulcorner sl_{all} := gen\_slices(l_I, san.final\_slice[version], length);$
$tiva_{all} := generate\_tiva(empty, san.all\_tiv[version],$
$\quad n.var);$
$n.child_p.slice\_set := n.child_p.slice\_set \cup$
$\quad n.child_p.queue[version];$
for all $(l, \mathfrak{T}) \in n.child_q.queue[version]$ **do**
 for all (l',$\mathfrak{T}'$) $\in sl_{all} \times tiva_{all}$ **do**
  if $comp(\mathfrak{T}, \mathfrak{T}', n.child_q.var)$ and $(l', l) \in \mathfrak{R}^+$
  then
  $\ulcorner C := true;$
 for all $(l", \mathfrak{T}") \in n.child_p.slice\_set$ **do**
  if $comp(\mathfrak{T}', \mathfrak{T}", n.child_p.var)$ and
   $(l', l") \in \mathfrak{R}^+$ and $(l'', l) \in \mathfrak{R}^+$
  then $C := false;$
 if C an $l \in sl_{all}$ **then**
 $\llcorner \llcorner n.queue[version] := n.queue[version] \cup (l', \mathfrak{T}');$

4.7 Results

On-the-fly model checking allows to check potentially infinite executions of concurrent and distributed programs. Monitoring of traces, generation of state action nets and model checking can be done simultaneously in a pipelined execution order. If a slice and a TIVA is generated in the root node of the formula tree, the specification is not valid for this run. In this case, the program run is stopped, and the user has several choices:

- variable values, control locations and generated slices can be inspected to find the cause of an error,
- values and locations can interactively be changed by assigning new values to them, and
- the program run can be resumed with the same specification and history, to reach the same condition again, or
- the specification can be changed, the history purged and the execution restarted from the current state.

If no slice and TIVA falsifying the formula is found, the program could run an unlimited amount of time. However, for each program event all slices containing this event have to be stored. Thus, an unlimited amount of memory would be needed for this purpose. Therefore, the program can only be run for a limited number of steps. If no error occurs within this time, the program has to be stopped and the history must be purged.

A typical application scenario is as follows. Assume that the user wants to debug a parallel sorting algorithm. There is one thread T_s executing a method *split* which splits the input into even and odd numbers, and two threads T_e and T_o executing a method *sort* to sort the even and odd numbers, respectively. To distinguish both threads *sort* has a boolean parameter is_e which has the value true for T_e and the value false for T_o during the execution of *sort*. When T_e and T_o are finished, the resulting sequences are merged by T_m. To control that the splitting is done correctly, the user would specify

($t1$:((**start.***sort* **and** *is_e*) → even(*input*)) **and**
 $t2$:((**start.***sort* **and not** *is_e*) → odd(*input*))).

Thus, whenever e.g. the thread for even numbers gets an odd input the execution is stopped. The cause of this error can then be found by examination of the sequence of actions leading to this situation. The following formula assures that T_e must be terminated before T_m is started:

(t:(**start.***split*)→ (t':(**term**.*sort* **and** *is_e*)
 before t'':(**start**.*merge*))).

If the synchronization between T_e and T_m is faulty, the program is stopped on the preliminary start of the merging process. To debug the merging, the user would halt the program when both separate sorting processes are finished:

not($t1$:(**term**.*sort* **and** *is_e*)
and ($t2$:**term**.*sort* **and not** *is_e*)).

The thread for merging can be advanced a single step with the specification

(t:(**start.***merge*)→ **all_next** t:(**term.***merge*)).

The whole program can be advanced n steps with the following formula:

(t:(**start.***split*)→ **all_next**^{n} t':(**term.***merge*)).

Since all of these specification formulas follow a fixed scheme, we are currently investigating the possibility of standard templates for them.

5 Conclusion and Further Work

We have presented an efficient algorithm for on-the-fly model checking of partial order structures. With this algorithm, parallel program runs can be automatically debugged. The execution is constantly monitored with respect to the specification. The use of a high-level temporal logic language allows a flexible use of the debugger. Invariance conditions can be supervised, concurrent simultaneous or alternative breakpoints can be set, and specific steps can be executed by appropriate formulas.

When the run stops at a specified breakpoint, the user can examine whether an error has occurred. At the moment, this check has to be performed by a manual inspection of slices. It would be nice to have a "replay" facility which visualizes the traced run and allows to step backward and forward within this partial order.

Another useful extension could be the integration of assignments in the specification language. With such an extension, variable values could be automatically changed in dependence on certain temporal logic conditions. On one hand, this would allow to modify the behaviour of the program without changing and recompiling the code. On the other hand, input values and test cases could be specified by logical formulas. This would unify and simplify the user interface of the debugger.

We are currently implementing our algorithm for the parallel programming language ParMod-C [16]. One of the main limiting factors is the size of the slice sets for each node in the formula tree. Therefore, we need a good representation for large sets of slices. In [6], a symbolic representation for such sets is developed. However, some changes are necessary to adopt this representation to the present case. Finally, we want to include algorithms proposed by Garg and Waldecker [9] to improve the efficiency of our method.

References

[1] B. Alpern and F. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, oct 1985.

[2] F. Baiardi, N. D. Francesco, and G. Vaglini. Development of a Debugger for a Concurrent Language. *IEEE Trans. on Software Engineering*, 12(4):547–553, April 1986.

[3] P. Bates and J. Wileden. High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach. *Journal of Systems and Software*, 3(4):255–264, 1983.

[4] R. Carver and K. Tai. Test sequence generation from formal specifications of distributed programs. In *Proc. 15th Int. Conf. on Distributed Computing Systems (ICDCS'95), Vancouver, Canada, 30. May–2. June 1995*, pages 360–367, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[5] M. Frey. Debugging parallel programs using temporal logic specifications. In I. Jelly, I. Gorton, and P. Croll, editors, *Software Engineering for parallel and Distributed Systems*, pages 122–133, London, March 1996. IFIP, Chapman & Hall.

[6] M. Frey. Using slice sets for model checking causal nets. In *Proc. of the 5th Int. Workshop on Verification In New Orientations (VINO'96), Rottach-Egern, 16.–19. May 1996*, pages 14–26. Institut für Informatik, TU München, Bericht TUM-I9720, 1997.

[7] M. Frey and M. Oberhuber. Testing and debugging parallel and distributed programs with temporal logic specifications. In *2nd Int. Workshop on Software Engineering for Parallel and Distributed Systems, Boston, MA, 17.–18. May 1997*, pages 62–72, Los Alamitos, CA, 1997. IEEE Coputer Society.

[8] M. Frey and A. Weininger. A temporal logic language for debugging parallel programs. In *Proc. 20th EUROMICRO Conference, Liverpool, England*, pages 170–178. Euromicro, IEEE, September 1994.

[9] V. Garg and B. Waldecker. Detection of weak unstable predicates in distributed systems. *IEEE Trans. on Parallel and Distributed Systems*, 5(3):229–307, March 1994.

[10] G. Goldszmidt, S. Yemini, and S. Katz. High-level language debugging for concurrent programs. *ACM Trans. on Computer Systems*, 8(4):311–336, Nov. 1990.

[11] P. Harter, D. Heimbigner, and R. King. Idd: An Interactive Distributed Debugger. In *5th Int. Conf. on Distributed Computing Systems, Denver, Colorado*, pages 498–506, 1985.

[12] M. Hennessy. Concurrent testing of processes. *Acta Informatica*, 32:509–543, 1995.

[13] J. Peleska. Test automation for safty-critical reactive systems. Technical report, 3rd Winter School on Formal and Applied Computer Science, 1.–12. July 1996, University of Cape Town, Department of Mathematics and Applied Mathemathics and Department of Computer Science, Kapstadt, Südafrika, 1996.

[14] W. Reisig. Temporal logic and causality in concurrent systems. In F. Vogt, editor, *Proc. Int. Conf. on Concurrency (Concurrency 88), Hamburg, 18.–19. October 1988*, LNCS 335, pages 121–139, 1988. Springer.

[15] N. Rescher and A. Urquhart. *Temporal Logic*. Springer, Wien-New York, 1971.

[16] A. Weininger, T. Schnekenburger, and M. Friedrich. Parallel and Distributed Programming with ParMod-C. In H. Zima, editor, *Parallel Computing, First Int. ACPC Conf., Salzburg, Austria*, LNCS 591, pages 115–126, 1991. Springer.