Runtime Verification of Concurrent Haskell Programs

Volker Stolz

RWTH Aachen University, 52056 Aachen, Germany

Frank Huch

Christian-Albrecht-University of Kiel, 24118 Kiel, Germany

Abstract

In this article we use model checking techniques to debug Concurrent Haskell programs. LTL formulas specifying assertions or other properties are verified at runtime. If a run which falsifies a formula is detected, the debugger emits a warning and records the path leading to the violation. It is possible to dynamically add formulas at runtime, giving a degree of flexibility which is not available in static verification of source code. We give a comprehensive example of using the new techniques to detect lock-reversal in Concurrent Haskell programs and introduce a template mechanism to define LTL formulas ranging over an arbitrary set of threads or communication abstractions.

Key words: Runtime Verification, LTL Checking, Haskell

1 Introduction

Today, almost any larger software application is no longer a sequential program but an entire concurrent system made of a varying number of processes. These processes often share resources which must be adequately protected against concurrent access. Usually this is achieved by concentrating these actions in *critical sections* which are protected by semaphores. If a semaphore is taken, another process wishing to take it is suspended until the semaphore is released. Combining two or more semaphores can easily lead to situations where a deadlock might occur. If we record a trace of actions leading up to this event (or any other crash of the program), we can give the developer useful advice on how to try to debug this error. Also, we do not have to limit ourselves to recording events on semaphores. It is useful to have unique markers throughout the program detailing periodically the position and state of the current run. Although this technique mostly resembles *hello-debugging*, we will show that these traces offer an added value: We can do runtime

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

verification for a running program by means of a runtime verifier embedded into the system.

Usually one is interested in conditions which hold in specific places or throughout the entire execution of the program. These properties cannot be easily determined or verified at compile time because it is not possible to verify all possible paths of program execution. But violation of these conditions can be detected in a post-mortem analysis or at runtime, e.g. during intensive testing. Especially, we can aggregate information from one or more runs and detect behaviour which did not lead to a crash but indicates that a condition might be violated in the future or in subsequent runs. This is regularly the case if program behaviour is dependant on scheduling or external input.

The properties we are interested in are not necessarily classic model checking properties like liveness or fairness [10], but are properties of the program known by the developer to hold at certain times. Commonly, these so-called assertions are verified by additional code placed by ambitious programmers at the appropriate locations in the source and usually removed before delivering the product because they increase program size and runtime.

A temporal logic like the *linear-time logic* (LTL) [1] lends itself naturally to describing properties depending on the changing states of a program over time. Formulas must never be evaluated to `False` during program execution. Our system allows the dynamic addition of new formulas which have to hold when the system evolves.

We show how to employ the technique of specifying a formula and that verifying it at runtime can be a significant help in testing Concurrent Haskell [9] programs. Additionally, the system records the path leading up to the state where the property was violated and thus aids the developer in debugging the program.

The article is structured as follows: Section 2 gives a brief introduction to Haskell and the technique of recording traces and using them for debugging. Section 3 provides some background on LTL and describes the LTL runtime verifier. Section 4 shows a sample verification of the *lock-reversal* problem, which can in general be solved by templates, introduced in Section 5. Section 6 concludes the paper and presents future work.

2 Tracing Haskell programs

First, we remind the reader of the purely functional lazy programming language Haskell [8]. Although there is extensive work on debugging Haskell (see [13] for a comparison), these mostly focus on the difficulties on debugging the evaluation of expressions in a lazy functional language: Evaluation might be deferred until the value is required for subsequent calculations. This leads to an evaluation order which cannot be easily derived from program source code (e.g. it is not feasible to single-step through program execution as you would in C or Java). We limit ourselves to debugging the IO behaviour of Haskell

programs (sequentialised by a monad) and can omit the problems resulting from Haskell's lazy evaluation strategy.

We are especially interested in the extension Concurrent Haskell [9] which integrates concurrent lightweight threads in Haskell's IO monad. As these threads can communicate, different schedules can lead to different results. Communication can take place by means of `MVar`s (*mutable variables*). These `MVar`s also take the role of semaphores protecting critical sections which makes them especially interesting to analyse.

In the IO monad, threads can create `MVar`s (`newEmptyMVar`), read values from `MVar`s (`takeMVar`) and write values to `MVar`s (`putMVar`). If a thread tries to read from an empty `MVar` or write to a full `MVar`, then it suspends until the `MVar` is filled respectively emptied by another thread. `MVar`s can be used as simple semaphores, e.g. to assure mutual exclusion, or for simple inter-thread communication. Based on `MVar`s, higher-level communication objects are built but are out of the scope of this paper.

In former work [3], we developed a tool for visualising communication between Concurrent Haskell threads called the *Concurrent Haskell Debugger* which makes it also possible to explicitly manipulate the scheduling of threads. This debugger is also able to replay recorded, concurrent executions for which a bug was found by the techniques presented in this paper.

As a simple example, we consider a small client server application implemented in Concurrent Haskell [1]:

```
server r a c = do
   req <- takeMVar r
   putMVar a (req,c)
   server r a (c+req)

client r a n = do
   putMVar r n
   (m,v) <- takeMVar a
   -- stateProp (n,m)
   client r a n

main = do
   r <- newEmptyMVar
   a <- newEmptyMVar
   forkIO (client r a 0)
   forkIO (client r a 1)
   server r a 0
```

The system consists of a server and two clients. The server provides a counter, which can be accessed and incremented by client requests. Whenever a client sends a request through the `MVar r`, the server responds through the `MVar a`. The answer contains the request (an `Int`) and the current value of the counter (`c`). Both `MVar`s are globally used for communication and synchronisation between the server and all clients. In its recursive call, the server increments

[1] The `do`-notation used in this example is syntactic sugar for Haskell's IO-Monad. A user not familiar with Haskell should just read it as a sequence of actions and '`<-`' as a (parallel) assignment. Comments start with '`--`'. The action `stateProp (n,m)` will be needed for debugging purposes and is explained below.

its state by the request.

The system is initialised in `main` by creating two empty `MVar`s and forking two client threads. Thereafter, the main thread itself switches to the server.

This example motivates our approach for run-time verification. On a first view, the program fulfils the following property:

(P) If a client sends a request, then the client receives an answer which contains the current counter state of the server.

We want to check this property by means of run-time verification. In a first step, we add atomic state propositions to the system, which identify relevant points during the execution. Therefore, we insert (uncomment) the action '`stateProp (n,m)`' after the `takeMVar` action in the client definition. During the execution of this action, the global state proposition (n, m) is valid. A necessary condition for (P) is that during the execution of the system only properties (0,0) and (1,1) occur, i.e. the answer m belongs to the corresponding request n.

Instead of developing a variety of algorithms which should be checked during the execution, we provide a general purpose checker for the powerful logic LTL. The programmer may specify properties by means of an abstract data type for LTL and can add LTL-assertions to any program point. For our example, it would be sensible to add the LTL-assertion

```
check "P" (g (Not (StateProp (1,0)) :/\: Not (StateProp (0,1))))
```

to the program. The formula expresses that in every state of the system ('`g`', globally) neither (1,0) nor (0,1) are valid state propositions. The assertion named `"P"` is activated by the execution of `check`. We will discuss the embedding of LTL in Concurrent Haskell in the next section.

Running this program for a while shows that the property is violated. The asserted LTL-property does not hold because of a race condition. In the background, we record a trace of the performed concurrent actions, which can be replayed by means of our Concurrent Haskell Debugger. This helps the user to understand the reason for the bug. The trace is produced by overloading every Concurrent Haskell function which writes a trace output into a file, similarly to [3].

In our small system, the bug results from a scheduling in which the execution of one client is interrupted by the scheduler directly after sending the request. Then, the servers answers the request. After this, the other client sends his request and reads the answer `MVar`, but obtains the response to the request of the other client. The program can be corrected by generating fresh answer `MVar`s for a request and submitting them to the server with the request.

3 Model checking LTL

Linear-time temporal logic (*LTL*) [1] is a subset of the Computation Tree Logic CTL* and extends propositional logic with operators which describe events along a computation path. The operators of LTL have the following meaning:

- "Next" ($\mathbf{X}\ \varphi$): The property φ holds in the next step
- "Eventually" ($\mathbf{F}\ \varphi$): φ will hold at some state in the future (also: "in the future","finally")
- "Globally" ($\mathbf{G}\ \varphi$): At every state on the path ϕ holds
- "Until" ($\varphi\ \mathbf{U}\ \psi$): Combines two properties in the sense that: φ has to hold until finally ψ holds.

The semantics of LTL is defined with respect to all paths in a given Kripke structure (a transition system with states labelled by atomic propositions (AP)). The path semantics of LTL is defined as follows:

Definition 3.1 (Path Semantics of LTL) Let AP be a set of atomic propositions. An infinite word over atomic propositions $\pi = p_0 p_1 p_2 \ldots \in (\mathcal{P}(AP))^\omega$ is called a path. A path π satisfies an LTL–formula φ ($\pi \models \varphi$) in the following cases:

$$
\begin{array}{rcll}
p_0\pi & \models & P & \text{iff } P \in p_0 \\
\pi & \models & \neg\varphi & \text{iff } \pi \not\models \varphi \\
\pi & \models & \varphi \wedge \psi & \text{iff } \pi \models \varphi \text{ and } \pi \models \psi \\
p_0\pi & \models & \mathbf{X}\ \varphi & \text{iff } \pi \models \varphi \\
p_0 p_1 \ldots & \models & \varphi\ \mathbf{U}\ \psi & \text{iff } \exists i \in \mathbb{N} : p_i p_{i+1} \ldots \models \psi \text{ and } \forall j < i : p_j p_{j+1} \ldots \models \varphi
\end{array}
$$

Further operations can be defined in terms of the above:

$$
\begin{array}{lll}
\mathbf{ff} = P \wedge \neg P & \mathbf{tt} = \neg\mathbf{ff} & \varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi) \\
\varphi \rightarrow \psi = \neg\varphi \vee \psi & \mathbf{F}\ \varphi = \mathbf{tt}\ \mathbf{U}\ \varphi & \mathbf{G}\ \varphi = \neg\mathbf{F}\ \neg\varphi
\end{array}
$$

As an example we consider the formula $P\ \mathbf{U}\ Q$. It is valid on the path $\{P\}\{P\}\{Q\}\ldots$ but it is neither valid on the path $\{P\}\{P\}\emptyset\{Q\}\ldots$ nor on the path $\{P\}\{P\}\{P\}\ldots$

From the definition of the path semantics one can easily derive the following equivalence:

$$\varphi\ \mathbf{U}\ \psi\ \sim\ \psi \vee (\varphi \wedge (\mathbf{X}\ (\varphi\ \mathbf{U}\ \psi)))$$

Using this equivalence it is easy to implement a checker that successively checks the formula for a given path.

3.1 Runtime verification

In traditional model checking, the complete state space is derived from a specification and all possible paths are checked. There are two reasons why formal verification is often not used in practice: First, for real programs the

specification has to be derived from the source code which is to be checked. This usually involves parsing the source and additional annotations, e.g. as comments. Furthermore, model checking is not applicable for large systems, because of state space explosion. In many cases, a system may even have an infinite state space and the model checking problem is undecidable.

Runtime verification does not need the state space beforehand, but simply tracks state changes in a running program. Thus, it limits itself to verifying that a formula holds along the path the program actually takes. This means that although errors could not be detected in the current run, they may still be present in the state space of the program. Various runs taking different paths may be necessary to find a path which violates the formula. However, runtime verification enables the use of well understood, formal techniques for the systematic validation of concurrent systems.

Atomic propositions in a state are set or deleted explicitly by the program where the necessary statements have either been introduced by the developer or a tool which derived them from some sort of specification.

3.2 Implementing LTL in Haskell

In this section we sketch the rough details of the LTL runtime checking engine. In Haskell, LTL formulas can simply be implemented as an algebraic data type:

```
data LTL a = StateProp (Prop a)     -- atomic proposition
           | TT                     -- True
           | FF                     -- False
           | X (LTL a)              -- NeXt
           | Not (LTL a)            -- Negation
           | U (LTL a) (LTL a)      -- Until
           | R (LTL a) (LTL a)      -- Release
           | (LTL a) :/\: (LTL a)   -- And
           | (LTL a) :\/: (LTL a)   -- Or
```

We allow arbitrary types as atomic propositions. Therefore, `LTL` is polymorphically defined over a type variable `a`. We also define disjunction and a release operator `R`. Like many model checking approaches, it is used as the dual operator to `U`. We deal with negation by pushing all negations in front of atomic propositions. Similarly to Definition 1, we handle the abbreviations **F** and **G**:

```
f phi = U TT phi                g phi = Not (f (Not phi))
```

For the checking engine, a separate thread maintains the global state of atomic propositions and LTL formulas to be checked. It accepts requests from the user's program to modify the set of valid atomic propositions. `setProp, releaseProp :: Ord`[2] `a => Prop a -> IO ()` extend respectively restrict the set of valid atomic propositions. `check :: Ord a => String -> LTL a`

[2] '`Ord a=>`' is a Haskell class constraint for the type `a`. Since atomic propositions are stored in a set, an ordering on propositions must be defined.

`-> IO ()` adds new formulas to the pool of formulas at runtime. Finally, a `step` message (produced by any concurrent action) causes the check thread to evaluate the active formulas with respect to the currently holding atomic propositions. Evaluation of a formula can have three different results which we map into the data type `Either Bool (LTL a)`[3]:

(i) `Left True`: The formula was proven and can be removed from the pool.

(ii) `Left False`: The verification failed. In case this condition does not coincide with a program crash, the user might want to proceed to check the remaining formulas.

(iii) `Right` φ: After this step, the formula reduced to neither `True` nor `False`. The new formula φ has to be checked on the remaining path.

The evaluation function called by the check thread is as follows (due to the lack of space, we restrict ourselves to basic propositions and the `until` operator):

```
checkStep :: Ord a => LTL a -> Set a -> Either Bool (LTL a)

checkStep (StateProp p) stateProps = Left (p `elem` stateProps)

checkStep u@(U phi psi) stateProps =
 let phiEval = checkStep phi stateProps in
  case checkStep psi stateProps of
   Left True  -> Left True
   Left False -> case phiEval of
                   Left False -> Left False
                   Left True  -> Right u
                   Right phi' -> Right (phi' :/\: u)
   Right psi' -> case phiEval of
                   Left False -> Right psi'
                   Left True  -> Right (psi' :\/: u)
                   Right phi' -> Right (psi' :\/: (phi' :/\: u))
```

In the last two cases of the outer case expression we build a new formula that has to be checked in the next step. From model checking we know that only a finite set of formulas can occur. To avoid multiple checking of equal formulas (resulting from nested fixed point operators, e.g., $G\ F\ \varphi$), the check thread stores the formulas to be checked in a set.

3.3 Verifying concurrent programs

As long as only one process is generating a debugging trace that we use for verification, the interpretation of the debugging trace is clear. Each new entry in the trace corresponds to a transition in the Kripke structure. What happens if

[3] The Haskell type `Either` can be used to union two arbitrary types. The underlying types are identified by two constructors: `data Either a b = Left a | Right b`.

we allow more than one process to manipulate the global state? Consider the following example: Two processes (P_1 and P_2) want to set a different proposition each, unaware of the other process (see Figure 1). Although the states of the two processes are aligned on the time axis, it is wrong to assume that they proceed in lock-step! Concurrent Haskell has an interleaving semantics, so if we join the propositions from both processes we obtain the model in Figure 2. Notice how each path resembles a possible execution order in Haskell. A left branch indicates process P_1 taking a step while a right branch corresponds to a step by process P_2. If there is only one transition leaving a state, this is the only possible step because one of the processes has already reached its final state (final propositions underlined).

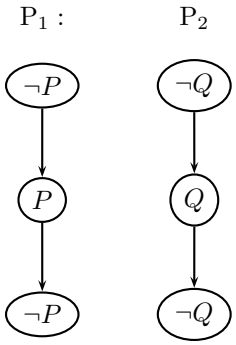

Fig. 1. Separate state spaces

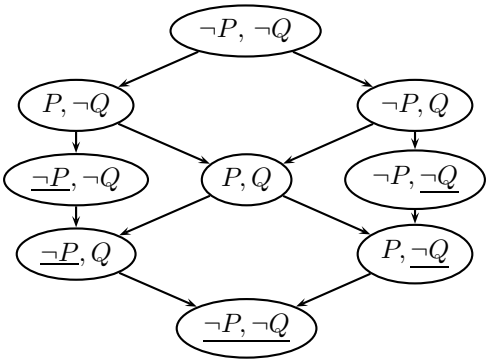

Fig. 2. Interleaved state spaces

Because of this interleaving it is clear that you should not use "Ne**X**t" explicitly in your formula but instead use "**U**ntil" or another fixed point operator: Changes in propositions relevant to one formula may affect the evaluation of others. We enforce this in the implementation by hiding the "Ne**X**t"-constructor from the programmer by Haskell's module system.

3.4 Using LTL to check assertions

In Section 2 we already presented a simple application of our LTL runtime checker. In this example, we only set a single atomic proposition by means of `stateProp`. Sometimes it can be useful to set propositions for a period of time, e.g., to represent a thread being in a critical section. The function `stateProp` is defined as a sequence of `setProp, step` and `releaseProp`.

The `step` messages for the check engine thread are automatically generated by extended versions of the concurrency actions. I.e., we only consider concurrency actions of the system as steps while calculations, independently of how much time they take, are not interpreted as steps in the state space. This is permissible since a computation does not affect the concurrent system as long as the result is not used for communication.

Executing a program containing a (specified) bug and giving it enough time and a "favourable" scheduling, the runtime verifier will stop the program

as soon as the formula is falsified. Furthermore, it prints a message which formula was falsified and stores a trace which can be used by the Concurrent Haskell Debugger.

4 Example application: Lock-reversal

In this section we will discuss how to implement the following rather complex check using the LTL runtime verifier: If two locks are taken in a specific order (with no release steps in between), the debugger should warn the user if he also uses these locks in swapped order (*lock-reversal*) because in concurrent programs this would mean that two processes could deadlock when their execution is scheduled in an unfortunate order.

However, this warning might not be adequate as it is always possible to construct a program which takes the locks in reverse order and will never deadlock. This can be achieved by adding another lock, a so-called gate lock, to the program which assures mutual exclusion and prevents the system from deadlocking. An algorithm which avoids this kind of false warnings in the presence of gate locks is detailed in [11].

This particular technique of noticing inadvertent lock-reversal at runtime is employed in the development phase of the recent FreeBSD operating system kernel under the name of *witness* [6]. In [11] the same technique is used in the Java PathFinder [15], a model checker for Java applications.

We consider two concurrent processes competing repeatedly for two locks A and B. It is not possible to acquire several locks in one go, so both locks have to be obtained sequentially. If we assume that the first process tries to obtain them in the order A, B and the second one in B, A, we can see that sooner or later the system ends up in a state where process 1 holds lock A and waits for B to become available while process 2 holds B waiting on A! This situation is exactly the circular holding pattern which indicates a deadlock (cf. chapter 7 of [2]). In small programs the bug might be easy to spot. For larger applications we can conclude that it is very hard to prevent this error from simply perusing the source code. Even when testing the program the error might not occur for various runs. For example it may be possible that process 1 and 2 are never executed interleaved but rather in a sequential manner.

For debugging Concurrent Haskell programs, we provide an algebraic data type which contains the `ThreadId` of the corresponding thread so that formulas can contain per-thread expressions. Additionally, we will record the name of the `MVar` being held. We use this data type as predefined atomic propositions. They are automatically set and released by concurrency actions as follows:

- `takeMVar: setProp (ThreadProp <threadId> <mvarName>)`
- `putMVar: releaseProp (ThreadProp <threadId> <mvarName>)`

This strategy can also be applied to other concurrency functions.

By using this approach, the program will generate a trace containing the

above propositions, possibly in the order indicating the erroneous behaviour. We can detect this and warn the developer that his application has the potential to enter a deadlock under certain conditions.

In the following we give a sample LTL formula for two locks which will evaluate to `False` in case the locks are used in reverse order. Notice that we have to provide fixed `ThreadIds` and lock names in the Haskell program to detect a specific situation. In Section 5, we will see how to solve this more elegantly with templates, as it is usually impractical in a dynamic system to generate all formulas beforehand. Instead of using the algebraic data type from the assertion, we will write $holds_{(p_i,l_x)}$ in the formula if process i holds lock x. The effect of the "Globally" is to check this formula over and over again for each step the program makes:

$$\mathbf{G}\ \neg\varphi\ =\ \mathbf{G}\ \neg\ (holds_{(p_j,l_x)} \wedge \neg holds_{(p_j,l_y)} \wedge (holds_{(p_j,l_x)}\ \mathbf{U}\ holds_{(p_j,l_y)})).$$

For a better understanding of this formula, we consider the following program:

```
main = do                                          take m1 m2 = do
 check "lock" (g (Not                                takeMVar m1
  (phi (StateProp (ThreadProp 1 "B"))                takeMVar m2
       (StateProp (ThreadProp 1 "A")))))             ...
 mvA <- newMVar ()                                   putMVar m1 ()
 mvB <- newMVar ()                                   putMVar m2 ()
 take mvB mvA

-- helper function:
phi u v = (u :/\: (Not v)) :/\: (u `U` v)
```

When this program is executed, first the formula is passed to the verifier. The two overloaded `newMVar` invocations trigger `releaseProp` statements for the model checker because they use the instrumented `putMVar` internally. As the current state contains no properties yet, they are of no effect. When the `take` function is invoked, this will set the corresponding proposition and execute a transition in the LTL checker: After the first `takeMVar` the formula is evaluated in the current state $\{\neg holds_{(p_1,l_A)}, holds_{(p_1,l_B)}\}$ and reduced to $\neg\varphi' \wedge\ \mathbf{G}\ \neg\varphi$, where

$$\begin{aligned}\varphi' &= holds_{(p_1,l_B)}\ \mathbf{U}\ holds_{(p_1,l_A)} \\ &= holds_{(p_1,l_A)}\ \vee\ (holds_{(p_1,l_B)} \wedge\ \mathbf{X}\ \varphi').\end{aligned}$$

After stepping the formula and taking the next `MVar` we can see that the new state $\{holds_{(p_1,l_A)}, holds_{(p_1,l_B)}\}$ evaluates φ' to `True`, which in turn gets negated and invalidates the whole formula. The model checker has determined that the specified property does not hold for the current path and acts accordingly, e.g. prints a warning to the user.

5 Using formula templates

In the previous section, we used a static formula to demonstrate the mechanism. But in a dynamic system `MVar`s may be generated and used on the fly and thus make it impossible to generate all formulas beforehand. To be able to manage this problem elegantly, we introduce *templates*. Templates contain free variables which are instantiated with all properties occurring throughout the program run. This includes instantiating new formulas once a new proposition is used. Ideally, we would like to be able to specify a template for the problem above in the following form: "If I observe a process taking lock x and holding it until taking lock y, I'd like to install a formula which invalidates iff another process takes x and y in reverse order." To rephrase this in LTL:

$$\begin{aligned} & holds_{(p_i,l_x)} \wedge \neg holds_{(p_i,l_y)} \wedge (holds_{(p_i,l_x)} \ \mathbf{U} \ holds_{(p_i,l_y)}) \\ \rightarrow \mathbf{G} \ & \neg(holds_{(p_j,l_y)} \wedge \neg holds_{(p_j,l_x)} \wedge (holds_{(p_j,l_y)} \ \mathbf{U} \ holds_{(p_j,l_x)})), \ i \neq j, x \neq y \end{aligned}$$

However, implementing this in Haskell would mean devising a new syntax and writing a different LTL front end because such a formula-template cannot be easily captured with an algebraic data type: the LTL syntax would have to be extended for each new proposition.

It is far easier to use a real Haskell function instead which gets passed all currently known propositions every time a new proposition is set for the first time and decides itself which of those can be used to instantiate the formula. Because of Haskell's guarded expression syntax, this closely resembles the above syntax including the additional conditions.

It is not possible because of the type system to collect functions of different arities in a data structure as would be necessary for the implementation of the template mechanism, we use a simple trick: Since all propositions are of a single type, a template function simply takes a list of propositions as argument. The template engine will generate all possible permutations of the required length of propositions and pass them to the function. If the propositions are unsuitable for the template, the function returns `Nothing`[4]. Otherwise, an LTL formula is returned via `Just`. The following Haskell function implements the proposed template:

```
lockF :: Ord a => [Prop a] -> Maybe (LTL a)
lockF (p1@(ThreadProp i1 x1):p2@(ThreadProp i2 y1):
       p3@(ThreadProp j1 y2):p4@(ThreadProp j2 x2):_)
  | i1 == i2 && x1 == x2 && j1 == j2 && y1 == y2 &&
    i1 /= j1 && x1 /= y1 = Just
      ((phi p1 p2) --> g (Not (phi p3 p4)))
  | otherwise = Nothing
```

[4] The predefined Haskell data type `Maybe` is defined as follows:
`data Maybe a = Nothing | Just a`

```
lockF _ = Nothing
```

We use pattern matching to extract four arguments from the list of parameters and discard the rest through the "_"-placeholder. If the arguments are not four `ThreadProps`, the second definition of `lockF` tells the template engine that the propositions were unsuitable for instantiation by returning `Nothing`. If the template received four `ThreadProps`, the actual content must be tested against the additional constraints: The guard tests if we are really instantiating the pattern with two different threads and two different locks like we required in the mathematical specification above. It is not possible to use the same variable multiple times in pattern matching expression like it is e.g. in Prolog, so we have to use fresh variables for each thread and each lock and explicitly test in the guard those variables which have to coincide.

After the tests succeed, we use the saved expressions in `p1...p4` to build the formula. Notice that this template can be instantiated for the first time only after at least four (different) propositions fulfilling the necessary requirements have been set by the program. At this point, we can add the instantiated formula to the formulas already present and begin checking it, starting from the current state.

We observe that without further modifications, for example ordering on the arguments, the template might be instantiated too often, leading to redundant formulas, e.g. if the formula is commutative. To optimise the generation of permutations when a new proposition is set, we only generate *new* permutations by maintaining a list of all already used propositions.

5.1 Partial instantiation

For some formulas, including the one above, it is even possible that either the formula evaluates to `True` or `False` without requiring all arguments or that the last action leading to the last required proposition is indeed the proposition which would have invalidated the formula. But as these are temporal formulas and we do not instantiate a template until we have a sufficient number of arguments, we might miss to do a *partial instantiation*: For the template above, we could instantiate the left hand side of the implication as soon as we have the first two appropriate arguments. This partial formula could then be evaluated and stepped in time. In the present case the left hand side of the implication might evaluate to `False`, thus rendering the remainder of the formula redundant.

However, once again an actual implementation of doing partial instantiation and evaluation leads to even more complex bookkeeping: By projecting this feature onto Haskell's partial application, we can implement this as a function returning `Nothing` for unsuitable propositions like before, and either a new, partially instantiated formula requiring less arguments or a complete formula otherwise.

```
newtype TemplateF a = TF  -- new data type for templates
```

```
  ([Prop a] -> Maybe (Either (TemplateF a) (LTL a)))

lockF :: Ord a => [Prop a] ->
         Maybe (Either (TemplateF a) (LTL a))
lockF (p1@(ThreadProp i1 x1):p2@(ThreadProp i2 y1):_)
  | i1 == i2 && x1 /= y1 = (Just . Left . TF)
      (\ xs -> lockF2 (p1:p2:xs)) -- return anonymous function
  | otherwise = Nothing           --  with partial application
lockF _ = Nothing

lockF2 :: Ord a => [Prop a] ->
          Maybe (Either (TemplateF a) (LTL a))
lockF2 (p1@(ThreadProp i1 x1):p2@(ThreadProp i2 y1):
        p3@(ThreadProp j1 y2):p4@(ThreadProp j2 x2):_)
  | x1 == x2 && j1 == j2 && y1 == y2 && i1 /= j1 =
       (Just . Right) ((phi p1 p2) --> g (Not (phi p3 p4)))
  | otherwise = Nothing
lockF2 _ = Nothing
```

Partial instantiation allows us to avoid testing a huge amount of argument combinations because only a fraction of arguments for the first two positions is valid. Just for those the second set of arguments has to be generated, providing an enormous benefit for this example. In general, the worst-case behaviours is equal to the previous behaviour of plain templates.

The system additionally offers the ability to record the entire trace of the program (or a configurable trailing part thereof) and check new formulas against this trace. Of course the amount of memory or storage required for keeping this trail might be prohibitive for most programs. Stepping partially instantiated templates will be supported in a future version.

6 Conclusion, Related and Future work

We presented a debugging framework for Concurrent Haskell that records way points in program execution and allows the developer to trace a particular path through the program by the Concurrent Haskell Debugger to draw conclusions about erroneous behaviour. Conditions which have to hold throughout program execution or at specific times can be expressed as formulas in the linear-time temporal logic LTL. The debugging trace provides the state transition annotations which are checked by the LTL model checking engine against the a set of formulas. In addition to specific debugging modules the developer can use explicit annotations to add information to the debugging trace. The debugger supports printing the tail of the trace from the point where it started proving the particular formula which turned `False`. Furthermore, we extend our previous work [14] with a powerful template mechanism which allows dynamic instantiation of formulas when the exact propositions are not known

beforehand, e.g. because they are parameterised by `ThreadIds` or `MVars`.

Several other approaches to runtime verification exist, especially for Java. Java PathFinder [11] is an instrumentation of the Java Virtual Machine which is able to collect data about problematic traces and run a traditional model checking algorithm. Java PathExplorer [5] picks up the idea of runtime verification and uses the Maude rewriting logic tool [12] to implement e.g. Future or Past time LTL. Both approaches still require several components while our Haskell library is self-contained. Java PathExplorer also offers so called error pattern analysis by user-implemented algorithms. Compaq's `Visual Threads` [7] debugger for POSIX threads applications allows scripted, rule-based verification. The Temporal Rover [4] checks time-dependent specifications and handles assertions embedded in comments by source-to-source transformation.

As future work, we intend to integrate the information from LTL checking in the graphical Concurrent Haskell Debugger to provide a comprehensive debugging facility for Concurrent Haskell. The effect of runtime verification on performance and memory requirements have yet to be examined for larger examples. We expect to profit from the inherent sharing of sub-formulas because of our choice of a lazy functional programming language. More information on the library for LTL runtime verification can be found at
http://www-i2.informatik.rwth-aachen.de/~stolz/Haskell/.

References

[1] A.Pnueli. The Temporal Logics of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[2] A.Silberschatz and P.B. Galvin. *Operating System Concepts*. Addison-Wesley, 4th edition, 1994.

[3] T. Böttcher and F. Huch. A Debugger for Concurrent Haskell. In *Proceedings of the 14th International Workshop on the Implementation of Functional Languages*, Madrid, Spain, September 2002.

[4] D. Drusinsky. The Temporal Rover and the ATG Rover. In W.Visser, K.Havelund, G.Brat, and S.Park, editors, *SPIN Model Checking and Software Verification (7th International SPIN Workshop)*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330, Stanford, CA, USA, August/September 2000. Springer.

[5] K. Havelund and G. Roşu. Java PathExplorer - A Runtime Verification Tool. In *Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01*, Montreal, Canada, June 2001.

[6] J.H.Baldwin. Locking in the Multithreaded FreeBSD Kernel. In Samuel J. Leffler, editor, *Proceedings of BSDCon 2002, February 11-14, 2002, San Francisco, California, USA*. USENIX, 2002.

[7] J.J.Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In W.Visser, K.Havelund, G.Brat, and S.Park, editors, *SPIN Model Checking and Software Verification (7th International SPIN Workshop)*, volume 1885 of *Lecture Notes in Computer Science*, Stanford, CA, USA, August/September 2000. Springer.

[8] S. Peyton Jones et al. Haskell 98 Report. Technical report, `http://www.haskell.org/`, 1998.

[9] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 1996.

[10] E.M.Clarke Jr., O.Grumberg, and D.A.Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.

[11] K.Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In W.Visser, K.Havelund, G.Brat, and S.Park, editors, *SPIN Model Checking and Software Verification (7th International SPIN Workshop)*, volume 1885 of *Lecture Notes in Computer Science*, Stanford, CA, USA, August/September 2000. Springer.

[12] M.Clavel, F.J.Duran, S.Eker, P.Lincoln, N.Marti-Oliet, J.Meseguer, and J.F.Quesada. The Maude system. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications, RTA'99*, volume 1631 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer.

[13] O.Chitil, C.Runciman, and M.Wallace. Freja, Hat and Hood — A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In M. Mohnen and P. Koopman, editors, *Proceedings of the 13th International Workshop on Implementation of Functional Languages (IFL 2000)*, volume 2011 of *Lecture Notes in Computer Science*, 2001.

[14] V. Stolz and F. Huch. Runtime Verification of Concurrent Haskell (work in progress). *Proceedings of the 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP'03)*, Technical Report DSIC-II/13/03, Universidad Politécnica de Valencia, Spain, 2003.

[15] W.Visser, K.Havelund, G.Brat, and S.Park. Model Checking Programs. In *Proc. of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000.