Hardware Implementation of an Invariant Observer

Marko Stanisic

Department of Informatics

Technical University of Vienna

This dissertation is submitted for the degree of

Bachelor of Science

2014

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Marko Stanisic
2014

Abstract

The Invariant Observer is a Runtime Verification Unit monitoring a signal ϕ from a System under Test in real time and determining whether the signal was in an active state and had been active up to τ clock cycles before. If this is the case then signal ϕ is observed as being invariant in the past within the time interval $[0,\tau]$.

In their paper „Runtime Verification of Embedded Real Time Systems“, Reinbacher et al. [2] presented a hardware implementation of an Invariant Observer. In this thesis a different hardware implementation is shown, that allows for parallel execution of several instances, leading to significant performance improvements if the time required for determining whether ϕ holds, is not neglectable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview of the Bachelor Thesis

In embedded real time systems it is necessary to make efforts to verify a system design. A system design can be formalized by a mathematical specification within a properly chosen dynamic system model. One approach to system design verification is a deduction, which shows that the design implies the requirements.

In critical Real Time Systems (RTS) timing constraints have to be considered in the requirement engineering. Such Real Time Systems are modelled by states changing over time. Time constraints can be formulated as constraints on the duration of critical states. A real time logic should be able to specify that real time constraints. Generally it seems that two main classes of real time logic are present,explicit or implicit temporal logic.[1]

Explicit temporal logic makes use of explicit expressions of time variables. The time variable can be the representation of a time interval or a variable in temporal logic. Implicit temporal logic (for example MTL - Metric Temporal Logic) is using temporal operators that constrain the extent of a state. It is based on interval temporal logic and the duration concept. Implicit temporal logic can be very useful to express before/after relations between concurrent actions. For further details [1] can be a good source of information. In runtime verification a monitor evaluates executions of a **System under Test (SUT)** [2]. The evaluation is formalised from a formal specification described in temporal logic.

For ultra critical systems it is important to meet four major requirements:

1. Functionality:cannot change target's behaviour
2. Certifiability:must avoid re-certification
3. Timing:must not interfere with the target's timing
4. Swap:must not exhaust size,weight and power tolerance

A **Runtime Verification Unit (RVU)** is a verification monitor that meets these four major requirements. As part of this requirements,the RVU must be separated from SUT. In fact it is a synthesized hardware that monitors the execution of a SUT.
The topic of my thesis "Hardware Implementation of an Invariant Observer" can also be considered as a RVU, it evaluates the execution of a SUT and checks it for invariance conditions. My observer is an alternative implementation of the invariant observer INVARIANT-SYMBOL published in [2], that bypasses the problem of resource limitation and makes use of the significant advantages of a highly parallel **Field Programmable Gate Array(FPGA)** hardware implementation. The most important difference is that my observer is not bounded to a specific τ,but the observers in [2] are bounded.This feature will be explained in the next section.

In the publication "**Real-Time Runtime Verification on Chip** " [2] the concept of a RVU and the principles of that Verification Framework are described in greater detail.

A survey about the functionality of the invariant observer is given in the following sections.

1.2 The Invariant Observer

This section is a survey about the invariant observer and how it works. More details about the observer algorithm are presented in the next chapter.
The Invariant Observer acts like the temporal (invariant previously) operator $\boxdot_\tau \phi$ of the Metric Temporal Logic (MTL) and is certainly restricted to the past (ptMTL). Such a temporal operator takes an input ϕ,the calculation of a propositional formula,and evaluates if ϕ holds for the past τ execution times,including the current execution time in a discrete time setting. For example the logical consequence $e^n \models \boxdot_3 \phi$ expresses that the operator $\boxdot_3 \phi$ is true at current time n iff (if and only if) the evaluation of ϕ is true now and was also true the last $\tau = 3$ execution times. In fact the $\boxdot_\tau \phi$ is a specialization of the $\boxdot_{[0,\tau]} \phi$ ptMTL operator which restricts the range of the invariance qualification.
Figure 1.1 and Figure 1.2 show an example for such a temporal operator.

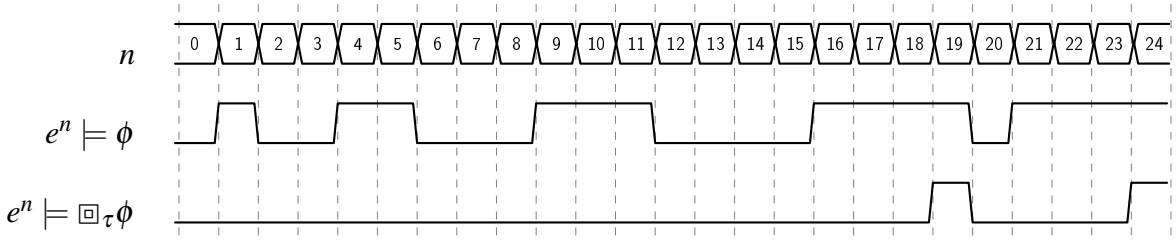


Fig. 1.1 Example for Invariant Operator $\boxdot_\tau \phi$ with τ=3

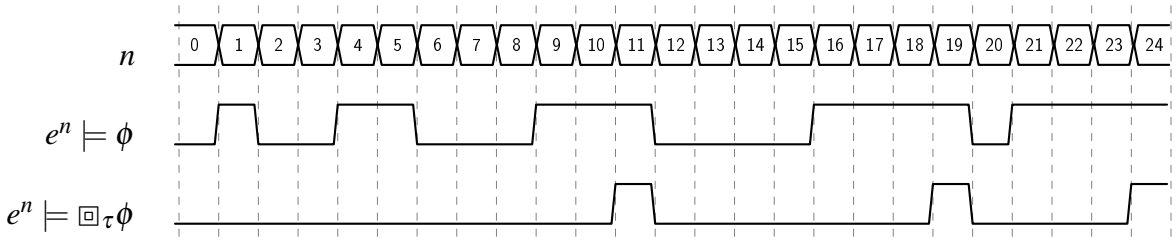


Fig. 1.2 Example for Invariant Operator $\boxdot_\tau \phi$ with τ=2

My approach of the invariant observer is based on some certain requirements. The following subsections will discuss these requirements.

1.2.1 Arbitrary calculation time of logical propositions

To introduce the first requirement we begin with the discussion of the problem that the calculation of a propositional formula ϕ could take several clock cycles (execution times). This means that an observer has to wait until the calculation of the proposition is finished. In [2] the observer needs to guarantee that it finishes evaluation of atomic propositions within a tight time bound. In our case,if we start calculation of a propositional formula ϕ at every clock cycle and the calculation itself needs y clock cycles,than we need at least y observer stages to cover finished calculations at every clock cycle. These observer stages are part of the whole observer. After y clock cycles,at every following clock cycle,a calculation of ϕ will be available. At least one observer stage will be ready,too,to evaluate a calculation ϕ at any time. In other words,we are implementing temporal pipeline stages that represent components of the invariant operator and these components together are evaluating the invariance qualification of the proposition ϕ. We don't have one observer,in fact the observer is composed from several observer stages. As a matter of fact,this solution only requires the calculation time of a proposition ϕ to be bounded by some previously known time y.

Propositional formulas can be composed from several other complex propositional subformulas or atomic propositions. In some cases a subformula is waiting for the resolution of an another subformula. In [2] this balance is achieved by the restriction of the atomic proposition class in sense of the abstract domain of logahedron.

1.2.2 Pipelined Observer Stages

Consider every finished calculation of a propositional formula ϕ as a signal value of the signal W(ϕ),every value in that signal is timely ordered just in the order it was calculated. Obviously,every represented execution time of W(ϕ) is the same as the execution time of the Observer. This means,at every execution time (clock cycle) an observer stage is evaluating a signal W(ϕ) at that time. In our case,signal W(ϕ) is apparently shifted by y clock cycles to the right. This view is encouraged by the fact,that at the beginning of the monitoring,the observer stages have to wait,until the first valid value of the signal W(ϕ) is available for evaluation of any observer stage which is duly put at disposal. The following observer stage evaluates,at execution time $n = y + 1$,the second valid value of W(ϕ),and so on and so forth. It should be considered,that the evaluation of a signal value from W(ϕ) relates to a propositional formula ϕ,which was relevant y clock cycles before. But it should also be mentioned that the signal values between execution time $n = 0$ and $n = y - 1$ are evaluated as well,but obviously with a negative result,because no calculation can be started before any

input is available.

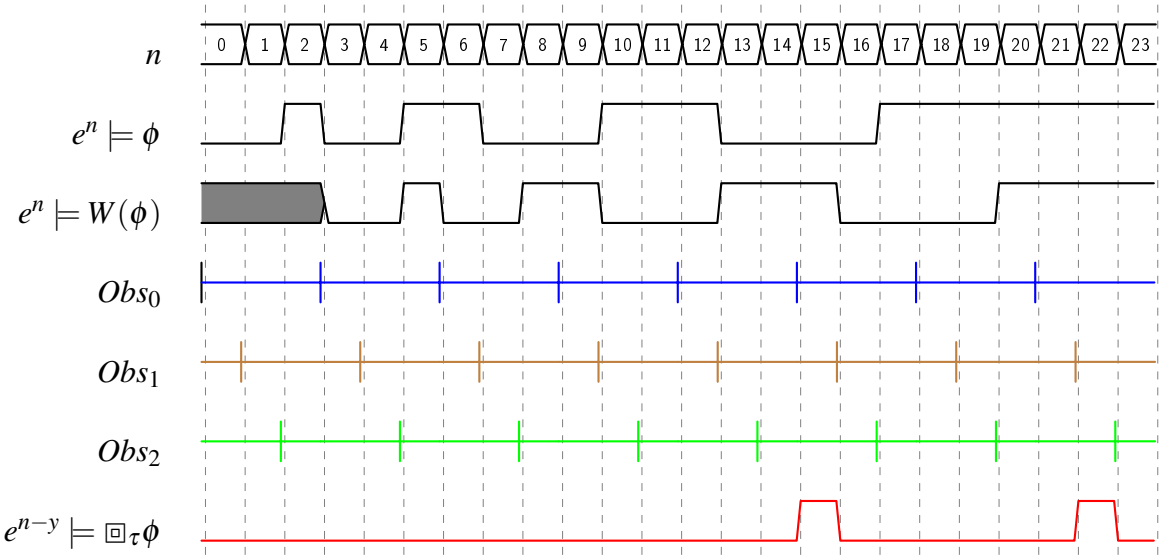


Fig. 1.3 Example for m=3 observer stages monitoring a signal W(ϕ), τ=2

The example in Figure 1.3 shows that the calculation of the first value from W(ϕ) needs y execution times. So no value is defined in the cycles before. If we take the view only from the proposition ϕ as a signal,then we calculate at every execution time n,from the latest inputs and subformulas at that time,exactly the proposition ϕ at that moment. But this holds under the assumption that the result of such a calculation is available immediately. The signal W(ϕ) shows us,what happens if calculations of proposition ϕ need some time,here we assume y=3 clock cycles. As mentioned,W(ϕ) is like signal ϕ shifted to the right. In Figure 1.3 we also see at which execution time the observer stages evaluate the signal W(ϕ). But this is only a preview about these observer stages. The algorithm about their real behaviour will be explained in the next chapter. The most important thing here is that these observers are working delayed. Every new observer stage starts its work after the observer stage before. This is important,because every clock cycle must be covered as long as the calculation of W(ϕ) takes. The last signal shows us the invariance qualification $e^{n-m} \models \boxdot_\tau \phi$ for τ=2,which holds at execution time $n = 15$ and $n = 22$, but either for one cycle only.

The invariance qualification will be resolved if we connect every observer stage in a binary “and” operation. In Figure 1.3 the result is simply calculated with:

$$\boxdot_2 \phi = Obs_0 \text{ and } Obs_1 \text{ and } Obs_2$$

Summarized, every observer stage is included in a binary ”and” operation and all these observer stages together are computing at every execution time n ,if the logical consequence $e^{n-m} \models \boxdot_{\tau}\phi$ holds. If we consider all these observer stages together as one temporal (invariance before) operator,than this operator checks at every execution time n,the invariance qualification of the proposition ϕ,m clock cycles before. Regarding Figure 1.3,it is true now if the proposition ϕ,m clock cycles before,was invariant with τ=2 .

1.2.3 System Settings of the Invariant Observer Stages

This was a brief introduction about the specific behaviour of the invariant observer as a whole. We will next detail on the parameter settings. It starts with the question of how many observer stages do we actually need ? We already know that at least y are necessary. If we have to define a number of observer stages with variable ”m” then the following condition must be hold: $m \geq y$. Depending on how long the calculation of a proposition ϕ needs,a minimum of y observer stages are necessary,but upwards can be chosen arbitrarily. This shows us also the possibility to apply the Observer with his observer stages on different evaluations of system inputs despite the time they need and it works in real time. If we use only one observer stage (means immediate evaluation of ϕ),then it works as a native invariant operator.

The next interesting aspect,and maybe the most powerful argument of this design way,is the invariance parameter τ. The number of observer stages ”m” stays in no relation to the invariance parameter τ. Following conditions are possible: $m \geq \tau$ or $m < \tau$. As long as the observer is configured to cover the computation time of W(ϕ),it can be used for every arbitrary choice of τ. In [2] this setting is fixed. My implementation of an invariant observer is able to change that parameter during run-time,but this feature is not specified in the requirements,and should be treated as such.

The next chapter shows the algorithm of the observer stages and how every observer stage works. Also a representation of the software implementation will be explained in great detail.

Chapter 2

Software Implementation and Algorithm

2.1 Algorithm of the Invariant Observer Stages

The following algorithm shows the proper behaviour of an observer stage.

Algorithm 1 Pseudo Code of an Observer Stage

Require: Precondition: $m \geq y$ and clock = 0

```
1:  Initialize: count = 0
2:  if (clock mod m) = 0 then
3:      if W(φ) = 0 then
4:          /*evaluates finished calculation of φ after m clock cycles*/
5:          count = 0
6:      else
7:          /*do nothing*/
8:      end if
9:  end if
10: /*Following code executes every clock cycle*/
11: if count = τ + 1 then
12:     output = 1
13: else
14:     output = 0
15: end if
16: count = min(count + 1, τ + 1)
17: return output
```

As you can see in Algorithm 1 the algorithm is separated in two main parts. The upper part checks from the start of the observer stage, and periodically every m clock cycles, the status of the current signal value W(ϕ). If W(ϕ) has an active status (e.g. W(ϕ)=1),the

counter remains his old value otherwise the counter will be set to zero. It is important that one observer stage recognize that the invariance qualification was not satisfied at this time. If the conjunction of all observer stages is done,at any arbitrary execution time n, and at least on stage hast not an active output, than the result is false. This means that,at the execution time n,the invariance qualification is not fulfilled. The bottom part is executed at every clock cycle and increments the counter value up to the maximum range of the invariance qualification. If the counter reaches the maximum value,the respective observer stage activates his output to an active state. In fact,the counter represents the invariance qualification of length 'τ'. The term '$\tau+1$' indicates that the present value must also be involved in the invariance qualification. The counter value will be initialized with zero at the beginning of the algorithm. Hypothetical,if counter is initalized with '$\tau+1$' the ouptut is activated immediately,because of the bottom algorithm. But this is an contradiction to the assumption that W(ϕ)=0 for all execution time n before 0.

It should be mentioned that this current design does not implement or handle the calculations of the propositions ϕ, which is indicated with W(ϕ). On the other hand the observers from [2] are responsible to take the necessary inputs, calculate the atomic propositions (with ATCheckers) and immediately evaluate the ptMTL operator qualifications. These steps have to be done in a tight time bound. In our case,W(ϕ) must be updated from another entity in such a way,that at every clock cycle an observer stage must have a consistent value for evaluation. The following subsection is an overview about the Vhdl implementation of the Algorithm 1.

2.2 VHDL Implementation of the Algorithm

In Appendix A,there is an implementation in VHDL which follows the meaning of Algorithm 1. We will discuss the different process entities,and the relations to the Algorithm 1. Finally,we get an overview about the improvements which are significant for a faster design. This vhdl design of an observer stage has following inputs and outputs:

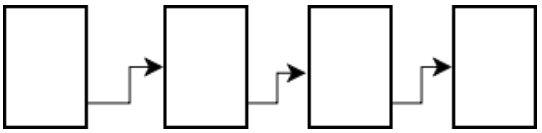
(a) inputs:

- **invariance_tau** is a signal variable which gets the value for τ
- **enable_in** signal activates the observer stage
- **signal_phi** is the signal state of W(ϕ)

(b) output:

- **enable_out** is a signal whose state is set to active after the activation of the current observer stage,but delayed for exact one clock cycle.
- **output signal** is simply the output state of the observer stage.

The Observer Stage is separated in a synchronous and asynchronous design,in sense of a Moore State Machine. The process labelled with "sync" represents the synchronous part of that design,in which on every clock cycle the states of the registers are changed. The synchronous process only works if combined signal **enable_logic** is activated,which indicates us a specific behaviour of the observer stage. The observer stages are connected in cascade to each other. The first stage activates the next observer stage over the signal **enable_out** after his activation over **enable_input**. If an activate state of **enable_input** is recognized in the current clock cycle,only then in the next clock cycle the **enable_out** signal will be active.



This ensures that the observers are working delayed in the meaning of time. The asynchronous part works immediately after every change of the system state. Signal **inc_tau** increments the input signal **invariance_tau**,which represents $\tau + 1$ at every execution time. The process entity **comb_cycle** is an internal clock counter which only counts up and down between values 0 and m. The signal **cycle** is significant to check the condition "(clock mod m) = 0" from the Algorithm 1. The process entity **comb_logic** implements the real part of the algorithm. Signal **count_p** will be incremented in each clock cycle until it reaches $\tau + 1$. The counter should be initialised with 0 (as indicated in the algorithm),but is incremented immediately every clock cycle,this also happens in the clock cycle where **count_p** should be reseted. To show this fact we have the signal **count** initialised with 1. But this signal is only for reasoning and will be reduced by a syntheses tool. The synchronous design is in a way cumbersome,but this is because of the way how the states changes. If the counter should be incremented after evaluation of some conditions,this doesn't mean an immediate change of the counter. To overcome this handicap the counter is initialised with 2,this means we check at every clock cycle if counter count_p will reach the maximum at the next clock cycle. This enables us to change things on time. In **comb_cycle**,**count_p** will be reseted only if signal $W(\phi)$ was evaluated as not active,according to the algorithm. One Case is observed in the first if branch,if the clock passed m cycles or not. If a reset condition happens

or **enable_in** is not active,than signal **enable_logic** combines these two cases in one signal which leads the if query inside of **comb_cycle** to the last branch,in case of **enable_logic** is not active. The other parts of **comb_cycle** are straightforward,if you compare it with the algorithm.In case the counter **count_p** reaches the maximum, the **output** of the observer stage is activated.

Some points about the improvements made in that design,but it can also be seen as guidelines for further design cases. It is very important that no Latches are built by the syntheses tool,so every if branch must contain the same changes on the same signals. A further point is to reduce the number of if branches to a minimum. If branches inside of an If branch extends signal paths and reduce the maximum clock design of the whole design.

In the next chapter we get an overview of the Hardware Realisation of the current design,which shows us a more visual view on that.

Chapter 3

Hardware Impelementation

Chapter 4

Experiments and Testing

References

[1] A.P. Ravn, H. Rischel, and K.M. Hansen. Specifying and verifying requirements of real-time systems. *Software Engineering, IEEE Transactions on*, 19(1):41–55, Jan 1993. ISSN 0098-5589. doi: 10.1109/32.210306.

[2] Thomas Reinbacher, Matthias Függer, and Jörg Brauer. Real-time runtime verification on chip. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification*, volume 7687 of *Lecture Notes in Computer Science*, pages 110–125. Springer Berlin Heidelberg, 2013.

Appendix A

Observer Stage Source Implementation in VHDL