Formal Timing Model for Hardware Components

Tomi Westerlund
Turku Centre for Computer Science
Turku, Finland
tomi.westerlund@utu.fi

Juha Plosila
University of Turku
Turku, Finland
juha.plosila@utu.fi

Abstract

The correctness of functional and non-functional properties of hardware components is ensured during development cycles conventionally by simulation. Also different description languages are needed during development phases. With Action Systems, we are able to use the same formalism from a specification down into an implementation. In this study we present timed Action Systems, an extension of Action Systems, with which a non-functional property, time, can be modelled. We show how untimed models are transformed into timed ones, and how timing characteristics of a model are analysed.

1 Introduction

Formal methods provide an environment to specify, design and verify digital hardware devices with the benefits of a rigorous mathematical basis. The Action Systems formalism [3], which we use in this study, is one of such methods. It is a framework for specification and correctness preserving development of concurrent systems and it is based on an extended version of Dijkstra's language of guarded commands [6]. Therefore, it offers a powerful stepwise design environment for digital hardware devices. It has already been successively proved to be suitable for designing both asynchronous [8] and synchronous [9] digital hardware systems. Development of action systems is done in a stepwise manner within the refinement calculus [2].

The specification of a hardware system is transformed into an implementation using correctness preserving transformations. In conventional Action Systems, only logical correctness of the system is verified during the transformations. Non-functional properties, like timing and area, of the end product are not validated.

In this study we introduce timed Action Systems, which is an extension of Action Systems. With timed Action Systems, we are able to formally specify timing of a hardware system. This gives us an opportunity to formally analyse, using precise mathematical calculations, timing during the development phases from specification down to implementation. When using informal design language, such as VHDL or Verilog, verification is based on simulation, which is always incomplete operation. We show how an untimed model of a hardware component is transformed into a timed one. We also give an example of timing analysis of a hardware component.

2 Action Systems

An *action* A is defined by

$A ::= \ abort$	(*abortion, non-termination*)
$\mid skip$	(*empty statement*)
$\mid A_1 \ [\!]\ \ldots \ [\!]\ A_n$	(*non-deterministic choice*)
$\mid A_1 \ /\!/ \ \ldots \ /\!/ \ A_n$	(*prioritised composition*)
$\mid A_1; \ldots; A_n$	(*sequential composition*)
$\mid x := e$	(*(multiple) assignment*)
$\mid p \rightarrow A$	(*guarded action*)

where A and A_i, $j = 0..n$, are actions; x is a variable or a list of variables; x_0 is a value(s) of the variable(s); e is an expression or a list of expressions; R and p are predicates.

The actions are defined using weakest precondition [6] for predicate transformers. For example:

$$\text{wp}(skip, Q) = Q,\ \ \text{wp}(x := e, Q) = Q[e/x]\ \ .$$

Actions and action composition are considered atomic, which means that only their pre- and post-states are observable, and when they are chosen for execution they cannot be interrupted by external counterparts.

The $guard\ gA$ of an action A is defined by $gA \ \widehat{=} \neg wp(A, false)$. An action is said to be enabled when its guard evaluates to true, disabled otherwise. If the action has a guard that is invariantly $true$, it is always enabled. The guard of a guarded action $b \rightarrow A$ is given as: $b \wedge gA$.

A *quantified composition* of actions is defined by $[\bullet\ 1 \leq i \leq n : A_i] \widehat{=} A_1 \bullet \ldots \bullet A_n$, where the bullet $\bullet$ denotes any of the composition operators, and n is the number of actions ($n \in \mathbb{N}$).

A *substitution* operation within an action A is denoted by $A[e'/e]$ where e refers to an element (variable, predicate, etc.) of the original action A, and e' denotes the new element, which replaces e in A. The same notation is applied to action systems as well.

2.1 Action System

An action system has a form:

```
sys  $\mathcal{N}ame$  $(g)$
  |[
   const  $c$
   var  $l$
   actions  $actions$
   init  $g, l = g0, l0$
   do  composition of $actions$ od
  ]|
```

in which we can identify three parts: *interface*, *declarations*, and *iteration*. The interface part specifies those variables, g, that are visible outside the action system boundaries, and thus accessible by other action systems. If an action system does not have any interface variables, it is *a closed action system*, otherwise it is *an open action system*. In the declarations part actions both ($actions$) and constants (c), variables (v), and their initialisations are declared. Using the items introduced in the interface and declarative parts the behaviour of the system is described in the iteration section; the **do-od** loop.

The operation of an action system is started by initialisation in which the variables are set to predefined values. Then in the iteration, actions are selected for execution based on the composition operators and the enabledness of the actions. This is continued until there are no enabled actions after which the action system is temporarily aborted until some other action enables it.

Parallel Action System. Consider action systems $\mathcal{A}$ and $\mathcal{B}$ in which both action names and local variables ($l_A \cap l_B = \emptyset$) are distinct, and a communication variables are a set $g_A \cap g_B$. We require that the initialisations of the communication variables $g_A \cap g_B$ in the systems $\mathcal{A}$ and $\mathcal{B}$ are consistent with each other, so that initial values are equivalent in the systems. The functionality of the action systems $\mathcal{A}$ and $\mathcal{B}$ is specified by actions A and B, respectively. The *parallel composition* of $\mathcal{A}$ and $\mathcal{B}$, denoted $\mathcal{A} \parallel \mathcal{B}$, is the action system:

```
sys   $\mathcal{A} \parallel \mathcal{B}$   (   $g_A \cup g_B$ )
 |[
  var   $l_A \cup l_B$   actions $A, B$
  init  $(g_A \cup g_B), l_A, l_B = (g_A0 \cup g_B0), l_A0, l_B0$
  do    $A \; [] \; B$  od
 ]|
```

Thus the parallel composition combines the state spaces of the constituent action systems keeping the local variables l_A and l_C distinct. The action systems $\mathcal{A}$ and $\mathcal{B}$ do not terminate independently of each other, but termination is a global property of $\mathcal{C}$.

3 Semantics of Timed Actions

In conventional Action Systems computation does not take time, a reaction is instantaneous – and therefore atomic in any possible sense. This is due to its software tailored background. In hardware systems time has a crucial role: for example in sychronous systems the time used in operation is restricted by the clock. In this paper, we take the view that every computation consumes time.

3.1 Time

The time at which we start computation is set in the initialisation of the action system. Conventionally a system time is initialised to zero, but it is of no importance because only the relative time among events is relevant. We define the time domain $\mathbb{T}$ be dense [1, 5] ($\mathbb{R}_+$), and continuous [5] ($\forall t_1 \exists t_2.(t_1 > t_2)$).

In an alternative approach a time domain is represented as natural numbers ($\mathbb{N}$) or integers ($\mathbb{Z}$) as in [7]. However, we see that non-negative real numbers offers more freedom on designing hardware systems; and furthermore, a discrete time can be modelled in continuous time domain by taking samples at certain intervals.

3.2 Timed Behaviour

Behaviour of an action system is a sequence s of states with two components [4]: $s =< (l_1, g_1), (l_2, g_2), \ldots >$, where l_i and g_i ($i \in \mathbb{N}$) are states of local and global variables, respectively. A *timed behaviour* is a sequence t: $t =< (l_1, g_1, ct_1), (l_2, g_2, ct_2), \ldots >$ [1, 7], where ct_i denotes the time when there was change in a state, and $ct_i \leq ct_{i+1}$. Thus, we have introduced a new variable ct_i into the sequence s. The new variable carries information about the time elapsed from the start of the system.

A trace, a sequence of observable states, in the Action Systems is formed by removing all the local variables from the states in the sequence s, and then removing all the consecutive states that are identical, called stuttering states [4]. A *timed trace* is formed using the same procudere except that the time variables, which are local variables, are not removed. Thus, a timed trace [1, 7] is: $tr =< (g_1, ct_1), (g_2, ct_2), \ldots >$.

4 Timed Actions

Consider an action system $\mathcal{A}$ with the following **do-od**–loop in which the actions are prioritised over an action Ft. The priorisation ensures that every action that is enabled will be chosen for execution before the action Ft is executed. We have:

$$\textbf{do} \; [\, [] \; 1 \leq i \leq n : A_i] \; /\!\!/ \; Ft \; \textbf{od} \; .$$

where A_i is a timed action of form $A_i \widehat{=} A_{o,i} \; [] \; A_{w,i} \; [] \; A_{r,i}$, where $A_{o,i}$ is an *operation* action, $A_{w,i}$ is a *write* action, and $A_{r,i}$ is a *release* action. The actions are defined by:

$$\begin{aligned} A_{o,i} \;&\widehat{=}\; \neg b[i] \wedge gA \rightarrow A_i[u/w]; b[i], ft[i] := T, ct + d[i] \;, \\ A_{w,i} \;&\widehat{=}\; b[i] \wedge (ct = ft[i]) \wedge gA \rightarrow w[i], b[i] := u[i], F \;, \\ A_{r,i} \;&\widehat{=}\; b[i] \wedge \neg gA_i \rightarrow b[i] := F \;, \end{aligned}$$

where i is the number of actions, $i \in \mathbb{N}$, ct denotes *a current time* inside the action system, d is *a delay* specific for a given action, and ft, called *a future time*, stores the time when the write variable will be updated. The boolean variable b sequences the operation into two parts. The action Ft is called an *update* action. It sets the future time to current time. It is defined by.

$$Ft \quad \widehat{=} \quad [\, [] \; 1 \leq i \leq n : min[i] \rightarrow ct := ft[i]] \;,$$

where n is the number of timed actions ($n \geq 1 \wedge n \in \mathbb{N}$), and the guard $min[i]$, given as:

$$min[i] \widehat{=} (ft[i] > ct) \wedge (\forall j.1 \leq j \leq n \wedge j \neq i \Rightarrow (ft[i] \leq ft[j])) \;,$$

explores the values of future times $ft[i]$. It evaluates to *true* if a future time $ft[i]$ is greater than the current time ct and no other future time $ft[j]$ is smaller than it is. Future time is calculated in the action A_o by adding the delay d to the current time ct, $ft := ct + d$.

An activity of an action is performed in the operation action. The result of the activity is postponed by a specified delay $d[i]$ after which it is written in the write action to an output variable $w[i]$. The release action is only used when an timed action is disabled during the delay, that is, it prevents a timed action being *deadlocked*. This kind of situation may arise when several timed actions are enabled and executed at the same time, and moreover, they are modelled to disable each other. The result of the described behaviour is that only the winning action (chosen non-deterministically) may proceed, whereupon other timed actions are disabled forever without the action A_r.

A shorthand notation for a timed action is presented to prevent large, complex descriptions to become opaque. It hides all the details of the timed action definition. The shorthand notation is:

$$A\llbracket d\rrbracket \quad \widehat{=} \quad (A_o \;[\,]\; A_w \;[\,]\; A_r) \;/\!/\; Ft \ ,$$

and a presence of several actions:

$$\begin{aligned} &[\,[\,] \quad 1 \le i \le n : A_i\llbracket d[i]\rrbracket]\widehat{=} \\ &\qquad [\,[\,]\, 1 \le i \le n : (A_{o,i} \;[\,]\; A_{w,i} \;[\,]\; A_{r,i})] \;/\!/\; Ft \ . \end{aligned} \tag{1}$$

Composition of Timed Action Systems. Consider timed action systems $\mathcal{A}$ and $\mathcal{B}$ with distinct local variables: $l_A \cap l_B = \emptyset$ and communication variables are a set $g_A \cap g_B$. We require that the initialisations of the communication variables $g_A \cap g_B$ in the systems $\mathcal{A}$ and $\mathcal{B}$ are consistent with each other, so that initial values are equivalent in the systems. The functionality of the action systems $\mathcal{A}$ and $\mathcal{B}$ is specified by actions $A\llbracket d_A\rrbracket$ and $B\llbracket d_B\rrbracket$, respectively. The *parallel composition* of $\mathcal{A}$ and $\mathcal{B}$, denoted $\mathcal{A} \parallel \mathcal{B}$ is:

```
**sys**   $\mathcal{A} \parallel \mathcal{B}$   $(g)$
|[
 **var**    $l_A \cup l_B$: $natural$
 **init**   $g, l_A, l_B = g0, l_A0, l_B0$
 **do**     $A\llbracket d_A\rrbracket \;[\,]\; B\llbracket d_B\rrbracket$  **od**
]|
```

where $A\llbracket d_A\rrbracket \;[\,]\; B\llbracket d_B\rrbracket$ is according to definition (1): $(A_o \;[\,]\; A_w \;[\,]\; A_r) \;[\,]\; (B_o \;[\,]\; B_w \;[\,]\; B_r) \;/\!/\; Ft$. Hence, the composition of timed action systems combines the state spaces of the constituent action systems, merges the update actions Ft by unifying local time variables and keeping the local variables distinct.

5 Component Modelling

We show an example of transformation from untimed domain into a timed one. Then we analyse the operation of the arbiter in a timed domain by giving its execution sequence and a timed trace.

5.1 Clocked Computation

A clock is an action:

$$Clk \quad \widehat{=} \quad \neg clk \rightarrow clk := T \;[\,]\; clk \rightarrow clk := F \ .$$

It is used to sequence the operation of a system. The operation can be modelled, for instance, with two self-disabling parts: $Read$ and $Write$. The former performs the activity of the component and the latter stores the result. A synchronous system modelled using the above action is:

```
**sys**   $\mathcal{S}$   (   $din, dout :\ nat$  )
|[
 **var**    $clk, p, w, u$: $bool, bool, nat, nat$
 **actions**    $Read$:    $\neg clk \wedge \neg p \rightarrow u := f(din); p := T$
               $Write$:   $clk \wedge p \rightarrow dout := u; p := F$
                 $Clk$:   $\neg clk \rightarrow clk := T \;[\,]\; clk \rightarrow clk := F$
 **init**   $clk, b, w, din, dout = 0, 0, w_0, din_0, dout_0$
 **do**     $(Read \;[\,]\; Write) \;/\!/\; Clk$  **od**
]|
```

where f is a function and a clock changes its value after either the $Read$ or $Write$ action has disabled itself. The auxiliary variable p sequence the operation of the $Read$ and $Write$ actions. This behaviour is ensured by the prioritised composition. As in the previous example, we want to know more about the system's timing characteristics, and therefore we specify a delay for every action. The delays are d_r, d_w and d_{clk} for the actions $Read$, $Write$, and Clk, respectively.

Because we are changing a domain of a system, we need also to take in consideration that functionality is not affected by the introduction of delays. Therefore, we specify a requirement that ensures a proper operation of the system. The requirement is: $d_r, d_w < d_{clk}$. It ensures that the computation of an action must be completed before the clock signal changes its value. Now, after the transformation we have a timed synchronous system:

```
**sys**   $\mathcal{S}_t$   (   $g :\ nat$  )
|[
 **const**   $d_r, d_w, d_{clk}$: $8, 4, 10$
 **var**     $clk, w, u$: $bool, nat, nat$
 **actions**    $Read\llbracket d_r\rrbracket$:    $\neg clk \wedge \neg p \rightarrow u := f(din); p := T$
             $Write\llbracket d_w\rrbracket$:   $clk \wedge p \rightarrow dout := u; p := F$
              $Clk\llbracket d_{clk}\rrbracket$:   $\neg clk \rightarrow clk := T \;[\,]\; clk \rightarrow clk := F$
**init**    $clk, w, g = 0, w_0, g_0$
 **do**     $Read\llbracket d_r\rrbracket \;[\,]\; Write\llbracket d_w\rrbracket \;[\,]\; Clk\llbracket d_{clk}\rrbracket$  **od**
]|
```

The $Clk\llbracket d_{clk}\rrbracket$ is by definition:

$$\begin{aligned} Clk\llbracket d_{clk}\rrbracket \quad \widehat{=} \quad & (\neg b_{clk} \rightarrow (\neg clk \rightarrow clk' := T \;[\,] \\ & clk \rightarrow clk' := F); b_{clk}, ft_{clk} := T, ct + d_{clk}) \;[\,] \\ & (b_{clk} \wedge (ct = ft_{clk}) \rightarrow clk, b_{clk} := clk', F) \;[\,] \\ & (b_{clk} \wedge \neg gClk \rightarrow b_{clk} := F) \ , \end{aligned}$$

where $gClk \widehat{=} \neg clk \vee clk = T$ and therefore $gClk_r \widehat{=} b_{clk} \wedge \neg gClk = F$. Thus, the action Clk_r can be omitted. Note, that we changed the prioritised composition into a non-deterministic choice. The change was possible, because in the timed model the lapse of time does the same thing as the prioritised composition in the untimed model.

Clock signal. With the above model of the clock, we are able to model a clock wave with equal sized low $'0'$ and high $'1'$ periods, that is, the duty cycle, ratio of clock width and clock period, is 50%. We have to divide the clock action into two separate parts to be able to model other kinds of clock waves. In general we can define: $Clk0\llbracket d_{clk0}\rrbracket : \neg clk \rightarrow clk := T$ and $Clk1\llbracket d_{clk1}\rrbracket : clk \rightarrow clk := F$.

5.2 Arbiter

An arbiter is a component that takes care of selecting which one of the two or more independent modules access a shared resource. In the below system $\mathcal{S}ys$:

```
**sys**   $\mathcal{S}ys$   ( )
|[
 **init**   $l_{\mathcal{M}1}, l_{\mathcal{M}2}, ct = l_{\mathcal{M}1}0, l_{\mathcal{M}2}0, 0$
         $l_{\mathcal{A}rb}, l_{\mathcal{S}hR} = l_{\mathcal{A}rb}0, l_{\mathcal{S}hR}0$
 $\mathcal{M}1 \parallel \mathcal{M}2 \parallel \mathcal{A}rb \parallel \mathcal{S}hR$
]|
```

masters $\mathcal{M}1$ and $\mathcal{M}2$ communicate with a shared resource $\mathcal{S}hR$ through an arbiter $\mathcal{A}rb$. We assume that the time required by the masters $\mathcal{M}1$ and $\mathcal{M}2$ for their operation is d_m. The arbiter $\mathcal{A}rb$ is:

```
sys  $\mathcal{A}rb$  ( $req_{M1}, gr_{M1}, req_{M2}, gr_{M2} : bool$ )
|[
 const    $d$: 10
 actions        $M1_1$:   $req_{M1} \neg gr_{M2} \rightarrow gr_{M1} := T$
                $M1_2$:   $\neg req_{M1} \wedge gr_{M1} \rightarrow gr_{M1} := F$
                $M2_1$:   $req_{M2} \wedge \neg gr_{M1} \rightarrow gr_{M2} := T$
                $M2_2$:   $\neg req_{M2} \wedge gr_{M2} \rightarrow gr_{M2} := F$
             $M1[\![d]\!]$:   $M1_1 \; [\!] \; M1_2$
             $M2[\![d]\!]$:   $M2_1 \; [\!] \; M2_2$
init  $req_{M1}, gr_{M1} = req_{M1}0, gr_{M1}0$
      $req_{M2}, gr_{M2} = req_{M2}0, gr_{M2}0$
 do   $M1[\![d]\!] \; [\!] \; M2[\![d]\!]$  od
]|
```

and the iteration part by the definition is:

$$M1[\![d]\!] \; [\!] \; M2[\![d]\!] \quad \hat{=} \quad (M1_o \; [\!] \; M1_w \; [\!] \; M1_r) \; [\!] \; (M2_o \; [\!] \; M2_w \; [\!] \; M2_r) \;\; /\!\!/ \; Ft \;,$$

Let us consider a situation where the masters request the shared resource at the same time. After the requests have arrived, the arbiter grants either the master $\mathcal{M}1$ or $\mathcal{M}2$ non-deterministically. The winning master may proceed its activity, while the other must wait its turn. Note, that if the requests are not simultaneous, the grant is given deterministicly, the first requesting master is granted. The execution sequence is, for example:

$$\ldots, \mathcal{M}2, \mathcal{M}1, M1_{o,1}, M2_{o,1}, Ft, M1_{w,1}, M2_{r,1}, Ft, \mathcal{M}1, Ft, M1_{o,2}, Ft, M1_{w,2}, \mathcal{M}2, Ft, M2_{o,1}, Ft, M2_{w,1} \ldots \;, \quad (2)$$

where the masters simultaneously request the grant after which the actions $M1_{o,1}$ and $M2_{o,1}$ are executed. The next enabled actions, based on the definition of the timed actions, are $M1_{w,1}$ are $M2_{w,1}$. The arbiter gives the grant to the master $\mathcal{M}1$, and hence $M1_{w,1}$ will be executed and $M2_{w,1}$ becomes disabled. This leads a situation where $M2_{r,1}$ becomes enabled and executed. The former executed action reveals the winning master and the latter executed initialises the action $M2[\![d]\!]$. Now, the master $\mathcal{M}1$ may start accessing the shared resource. After the communication has ended, $\mathcal{M}1$ withdraws the request after which the arbiter gives the grant to the master $\mathcal{M}2$. The future time action Ft is executed whenever all the other actions are disabled.

The timed trace, the observable behaviour of the arbiter is formed by collecting the states of the global variable and the state of the time variable. The timed trace of the sequence 2 is:

$$\begin{aligned} tr_{Arb} =& (\neg req_{M1}, \neg gr_{M1}, \neg req_{M2}, \neg gr_{M2}, 0), \\ & (req_{M1}, \neg gr_{M1}, req_{M2}, \neg gr_{M2}, d_m), && \text{(a)} \\ & (req_{M1}, gr_{M1}, req_{M2}, \neg gr_{M2}, d + d_m), && \text{(b)} \\ & (\neg req_{M1}, gr_{M1}, req_{M2}, \neg gr_{M2}, d + 2d_m), && \text{(c)} \\ & (\neg req_{M1}, \neg gr_{M1}, req_{M2}, \neg gr_{M2}, 2d + 2d_m), && \text{(d)} \\ & (\neg req_{M1}, \neg gr_{M1}, req_{M2}, gr_{M2}, 3d + 2d_m), \ldots && \text{(e)} \end{aligned}$$

By analysing the timed trace above, we can conclude that the best case delay for a master to be granted is d, and if a master must wait the shared resource, the worst case delay is $3d + d_m$. Based on the timed trace we have depicted a timing diagram (Figure 1). We also included the boolean variable b into the timing diagram to address the inner execution sequences of the timed actions.

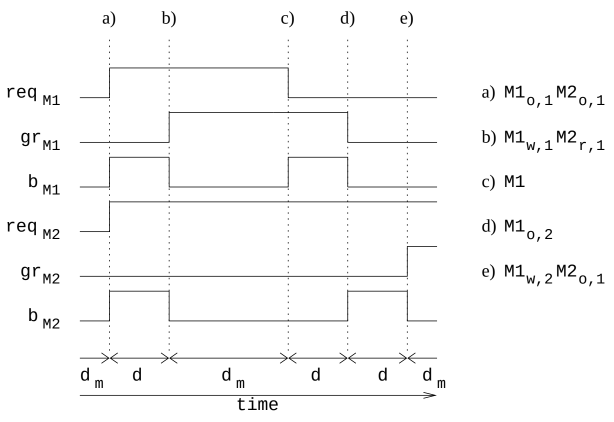

Figure 1: The timing diagram of the sequence (2)

6 Conclusions

In this paper we introduced Timed Action Systems with which it is possible to model hardware components in a time domain. We showed what is required in transformation of an untimed model to a timed one. In a transformation, not only the delays of individual actions, but also requirements that ensure the correct functionality of the system after the transformation must be specified.

The experiences of this study showed the usability of the defined Action System extension, and pointed out the direction for further studies. These include for example: a creation of a framework for modelling intellectual property (IP) blocks and development of a powerful formal timing verification method.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] R.-J. Back. *On the correctness of refinement steps in program development*. PhD thesis, University of Helsinki, 1978.

[3] R.-J. Back and K. Sere. From modular systems to action systems. In *Proc. of Formal Methods Europe '94*, Spain, October 1994. Lecture notes in computer science, Springer-Verlag.

[4] R.-J. Back and J. von Wright. Trace refinement of action systems. In *International Conference on Concurrency Theory*, pages 367–384, 1994.

[5] J. Davies and S. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, February 1995.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.

[7] C. J. Fidge and A. J. Wellings. An action-based formal model for concurrent real-time systems. *Formal Aspects of Computing*, 9(2):175–207, 1997.

[8] J. Plosila. *Self-Timed Circuit Design - The Action System Approach*. PhD thesis, University of Turku, 1999.

[9] T. Seceleanu. *Systematic Design of Synchronous Digital Circuits*. PhD thesis, Turku Center for Computer Science, 2001.