Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 187 (2007) 91–106
www.elsevier.com/locate/entcs

Time Aware System Refinement

Tomi Westerlund [1]

Turku Centre for Computer Science, Turku, Finland
Department of Information Technology
University of Turku, Turku, Finland

Juha Plosila [2]

Department of Information Technology
University of Turku, Turku, Finland

Abstract

We propose a formal, time aware refinement of systems. The proposed timewise refinement method is a direct extension of the traditional refinement calculus of Action Systems. The adaptation provides a well-founded mathematical basis for the stepwise refinement of systems modelled with the time spiced Action Systems formalism. In the refinement of an abstract system into a more concrete one a designer must show that conditions of both functional and temporal properties are satisfied.

Keywords: Timed Action Systems, refinement, time

1 Introduction

The correctness of an implementation with respect to an initial specification is traditionally validated using simulation based methods, and in a case of unwanted mismatch between the two models a new design cycle is required. This is time consuming as well as an error prone approach to design systems. To overcome tedious design cycles formal methods with a stepwise refinement method is a promising approach. They provide tools with which a high-level system specification can be transformed with the benefits of a rigorous mathematical basis to an implementable model.

The Action Systems formalism [2], henceforward called conventional Action Systems, is such a formal method that can be used throughout the design project. The correctness of transformations are ensured using the refinement calculus framework

[1] Email: tomi.westerlund@utu.fi
[2] Email: juha.plosila@utu.fi

doi:10.1016/j.entcs.2006.08.046

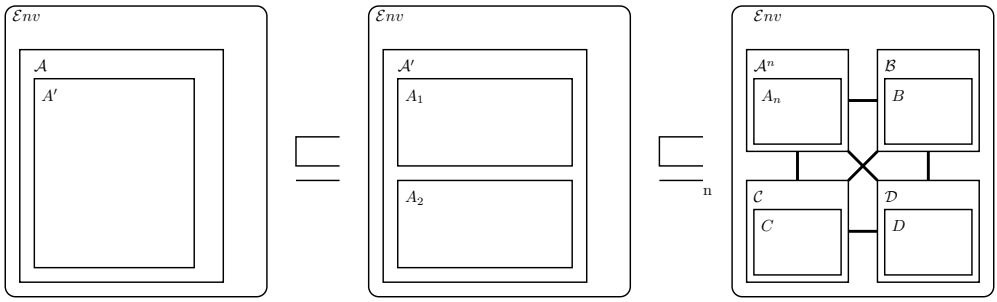


Figure 1. A example development of hardware system through several refinement steps.

[4]. Conventional Action Systems is a state based formal description language initially proposed by Back and Kurki-Suonio [1]. It is based on an extended version of a guarded command language introduced by Dijkstra [5]. It is used for specification and correctness preserving development of reactive systems. It was first tailored to a software system design but is then successfully applied also to hardware system design, both synchronous [9] and asynchronous [8].

The scope of this study is to extend the refinement calculus framework to be able to develop time spiced Action Systems, called Timed Action Systems [11], as well. That is, we present a time aware refinement methodology with which functional properties of an abstract system specification can be transformed, using standard refinement calculus, towards a more concrete one preserving its temporal characteristics (an example refinement is presented graphically in Fig. 1). The final goal in our research is to have a modelling framework for SoC/NoC (system-on-chip/network-on-chip) systems in which, within one formalism, a system can be modelled from a specification down to an implementable model. The development of a system starts from a high-level system model $\mathcal{S}$ whose functional and temporal characteristics are given in a specification. The refinement relation between system models is defined so that the total correctness is preserved: if $\mathcal{S} \sqsubseteq \mathcal{S}'$ and $p\mathcal{S}q$ then $p\mathcal{S}'q$ will also hold. After several successive refinement steps: $\mathcal{S} \sqsubseteq \mathcal{S}' \sqsubseteq \cdots \sqsubseteq \mathcal{S}^n$ we obtain an implementable specification. By transitivity, we have that $\mathcal{S} \sqsubseteq \mathcal{S}^n$ and $p\mathcal{S}^n q$ [4]. Thus, we have developed, in a stepwise manner, an implementation of the original specification.

The refinement based development method is introduced for several different specification languages some of which does not guarantee the correctness of a concrete model after several refinement steps while the others allow a stepwise development method, a chain of refinements. We shortly discuss three different refinement approaches that belongs to the latter group with our approach. In [13] is introduced an approach to refine high-level Timed MSC model into design specification. Their refinement approach resembles our trace refinement, as the environment should not distinguish between a given model and its refinement. In our approach, however, we do not consider the refinement of time constraints as in our target environment, VLSI systems, the system constraints are given by the specification, and thus are not allowed to be relaxed. However, the time constraints are allowed to be refined to meet the new, refined timed action system. In [7] is presented a refinement of action for real-time concurrent systems. They have taken the approach that the refined

action has the same duration than that of the original one, whereas we have not given any action specific restrictions because the timing behaviour is ensured by the constraints, and, furthermore, we have tried to minimise the number of additional time related proof obligations. Another real-time refinement calculus is presented in [6] for stepwise development of machine-independent real-time programs. The scope of this research is on software development. For our knowledge their approach is not applied in SoC/NoC system design, which is the interest of our research, nor it is possible to perform static timing analysis with their approach.

2 Timed Action Systems

In this section we give a short overview of conventional actions that form the formal basis for our timed formalism after which we continue to a quite elaborate review of Timed Action Systems.

2.1 Conventional Actions

An *action* A is defined (for example) by:

$A ::=$ *abort*	(*abortion, non-termination*)	| *skip*	(*empty statement*)
| $\{p\}$	(*assert statement*)	| $[p]$	(*assumption statement*)
| $x := x'.R$	(*non-deterministic assignment*)	| $x := e$	(*(multiple) assignment*)
| $p \rightarrow A$	(*guarded action*)	| $A_1 \ [] \ A_2$	(*non-deterministic choice*)
| $A_1; A_2$	(*sequential composition*)	| **do** A **od**	(*iterative composition*)
| $[[\mathbf{var}\ x := x_0;\ A]]$(*block with local variables*)			

where A and A_i, $i = 1, 2$, are actions; x is a variable or a list of variables; x_o some value(s) of variable(s) x; e is an expression or a list of expressions; and p and R are predicates (boolean conditions). The variables which are assigned within the action A are called the *write variables* of A, denoted by wA. The other variables present in the action A are called the *read variables* of A, denoted by rA. The write and read variables form together the *access set* vA of A: $vA \widehat{=} wA \cup rA$.

2.1.1 Semantics of actions

The *total correctness* of an action A with respect to a precondition P and a post-condition Q is denoted PAQ and defined by: $PAQ \widehat{=} P \Rightarrow \mathbf{wp}(A, Q)$, where $\mathbf{wp}(A, Q)$ stands for the *weakest precondition* for the action A to establish the post-condition Q. We define, for example: $\mathbf{wp}(abort, Q) = false$, $\mathbf{wp}(skip, Q) = Q$, $\mathbf{wp}((A_0 \ [] \ A_1), Q) = \mathbf{wp}(A_0, Q) \wedge \mathbf{wp}(A_1, Q)$, $\mathbf{wp}(\{P\}, Q) = P \wedge Q$, $\mathbf{wp}([P], Q) = P \Rightarrow Q$, $\mathbf{wp}((A_0; A_1), Q) = \mathbf{wp}(A_0, \mathbf{wp}(A_1, Q))$, $\mathbf{wp}(P \rightarrow A, Q) = P \Rightarrow \mathbf{wp}(A, Q)$ and $\mathbf{wp}(\mathbf{do}\ A\ \mathbf{od}, Q) = (\exists k.k \geq 0 \wedge H(k))$, where $H(0) = Q \wedge \neg gA, k = 0$ and $H(k) = (gA \wedge \mathbf{wp}(A, H(k-1))) \vee H(0), k > 0$. That is, the weakest precondition of the iterative composition of actions requires that after k repetitions of A the loop terminates, that is, A becomes disabled in a state where the post-condition Q holds. If $k = 0$, A is disabled and the iteration behaves as the *skip* action.

The boolean condition gA above is the guard of the action A, defined by: $gA \mathrel{\widehat{=}} \neg \mathbf{wp}(A, wfalse)$. That is, gA is $true$ in the states, where A does not behave miraculously. In the case of a guarded action $A \mathrel{\widehat{=}} p \rightarrow B$, we have that $gA = p \wedge gB$. An action A is said to be *enabled* in states where its guard is $true$. Otherwise A is *disabled*. The action A is said to be *always enabled*, if $\mathbf{wp}(A, false) = false$ (that is, the guard gA is invariantly $true$: $gA = true$). Furthermore, if $\mathbf{wp}(A, true) = true$ holds, the action A is said to be *always terminating*. The *body* sA of the action A is defined by: $sA \mathrel{\widehat{=}} A \ [\!] \ \neg gA \rightarrow abort$.

2.1.2 Notation

A *quantified composition* of actions is defined by: $[\bullet\ 1 \leq i \leq n : A_i]$, and it is defined by: $A_1 \bullet \ldots \bullet A_n$, where the bullet $\bullet$ denotes any of the composition operators, and n is the number of actions. Furthermore, a *substitution* operation within an action A_i, denoted by $A[e'/e]$, where e refers to an element such as variables and predicates of the original action A_i and e' denotes the new element, which replaces e in A_i. The same notation is applied to action systems as well.

A prioritised (' // ') composition [10] is a composition in which the execution order of enabled actions is prioritised. We have: $A \ // \ B \mathrel{\widehat{=}} A \ [\!] \ \neg gA \rightarrow B$, where the highest priority belongs to the leftmost action in the composition; thus, the leftmost enabled action is always chosen for execution.

Example 2.1 As an example of a conventional action let us have two variables x and y that are multiplied together and the result is written onto a variable *prod*. A conventional action performing the described function is defined as: $M : prod := x * y;$, where M is a label given for the action.

2.2 Timed Action System

Let us commence the introduction of Timed Action System by first showing its form, and then introducing its elements and computation model.

A timed action system $\mathcal{A}$ has a form:

sys $\mathcal{A}$ (g;) ::
|[**constraints** C_j : (B); **delays** dA_i; **var** l;
actions $A_i[\![dA_i]\!]$: (aA_i); **init** g, l : $g0, l0$;
exec **do** `composition of timed actions` A_i **od**]|

In the above system we can identify three main sections: *interface*, *declarative* and *iteration*. The interface part declares those variables, g, that are visible outside the action system boundaries, and thus accessible by other timed action systems. If a timed 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 is introduced all the local variables l, action definitions that perform operations on local and global variables, where aA_i is any kind of the defined atomic actions generated by the syntax given previously, A_i its label and dA_i a predicate that determines its delay. The predicate is joined to a timed action between *the delay brackets* $[\![\ ]\!]$. Furthermore, constraints C_j that define conditions whose strict adherence is mandatory are

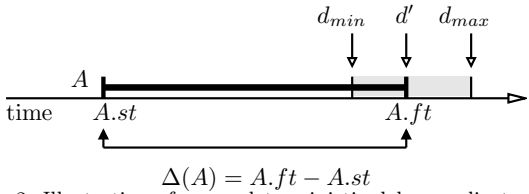


Figure 2. Illustration of a non-deterministic delay predicate dA_{nd}.

introduced in the declarative part. Finally, before the iteration section, the **do-od** loop of a timed action system, is the initialisation of both the global and local variables. The **do-od** loop defines a reactive behaviour of the system. It describes the composition of actions defined in the declarative part, that is, it defines the reactive behaviour, functionality of the system.

We chose the time domain be dense and continuous $\mathbb{T} = \mathbb{R}_{\geq 0}$, because it is a natural model for systems operating over continuous time. The elapse of time is modelled by postponing the update of the write variables. The time when the computation is commenced is set in the initialisation, but it is of no importance as only the relative ordering of timed actions is important. Let us next introduce delay predicates after which we introduce a timed action and its form in detail.

2.2.1 Delay models

As stated above, a delay of a timed action, say A, is determined by the predicate dA given in the **delay** clause located in the declarative part. In this section we introduce the two most commonly used delay predicates: a deterministic dA_n and non-deterministic dA_{nd} delay predicates. For these two predicates we use the following abbreviations $A[\![d]\!]$ and $A[\![d_{min}, d_{max}]\!]$, respectively. They are defined by:

$dA_d \mathrel{\widehat{=}} d' = d$	(*delay (deterministic)*)
$dA_{nd} \mathrel{\widehat{=}} d_{min} \leq d' \leq d_{max}$	(*delay (bounded, non-deterministic)*)

where d' is a variable of type $\mathbb{T}$ and d, d_{min} and d_{max} are numerical values of type $\mathbb{T}$. In the former predicate the delay d' obtains the value d. In the latter predicate the value is chosen non-deterministically between the given interval, see Fig. 2. That is, the exact value of the delay is not known beforehand. It is given during the evaluation of a component.

2.3 Timed action

The computation of conventional Action Systems does not take time, a reaction is instantaneous – and therefore atomic in any possible sense. Atomicity means that only pre- and post-states of actions are observable, and when they are chosen for execution they cannot be interrupted by external counterparts. This is due to its software tailored background. In modelling SoC/NoC systems it is important to know the time consumed by actions, because the operation speed is determined by the delay of those actions. Therefore, in Timed Action Systems we take the view that every computation takes time. This approach is also justified by the atomicity of actions and the fact that a state of the system can be observed after each execution of

actions. It should be observed that the complexity of a timed action is not restricted, and thus the operation time is not bounded either.

A timed action $A[\![dA]\!]$ is defined by:

$$A[\![dA]\!] \mathrel{\widehat{=}} (A_f \ [\!] \ A_k) \ /\!/ \ A_s \ /\!/ \ Pt \qquad (\textit{timed action}) \quad (1)$$

where we can identify three operational segments diveded by the prioritised compotision: *commence*, *end* and *time*. The commence segment contains *the start action* A_s whose execution initiates the operation of the timed action, and the operation is terminated in the end segment which consists of *the finish action* A_f and *the kill action* A_k. The one which will be executed depends on the enabledness of the timed action. That is, if a timed action is disabled by some other timed action when it is considered a scheduled timed action the kill action releases it for future computation. It prevents a timed action being *deadlocked*. The time is advanced in the time segment after the execution of the start action by executing *the time propagation action* Pt. A timed action whose operation is performed, but its write variables are not yet updated, is called *a scheduled timed action*. The time period during which a timed action is considered a scheduled one is determined by the predicate dA.

2.3.1 Timed Action in Detail

We shall next introduce the timed action components in detail. Thereafter, the composition of timed actions and the time propagation action will be introduced. Timed action components are defined by:

$$A_s \mathrel{\widehat{=}} \neg bA \wedge gA \rightarrow stateA := (wA, gt, gt + d'.dA) \ ; A[stateA.wA/wA]; bA := T; \qquad (\textit{timed action (start)}) \quad (2)$$

$$A_f \mathrel{\widehat{=}} bA \wedge gA \wedge (gt = stateA.ft) \rightarrow bA := F \ ; wA := stateA.wA; \qquad (\textit{timed action (finish)}) \quad (3)$$

$$A_k \mathrel{\widehat{=}} bA \wedge \neg gA \rightarrow bA := F; \qquad (\textit{timed action (kill)}) \quad (4)$$

where boolean variable bA sequences the operation into operation and write parts; gA is the guard of the timed action; $stateA$ stores *the new state* of the action. It is of type: **type** $state$: **record**$(wA; st, ft\colon \mathbb{T})$, where wA is the write variables of A, st a start time and ft a finish time. The start time is set to the global time gt and the finish time is obtained by adding the value of a delay to the global time. Observe that $stateA.ft$ actually stores the time when the write variable is scheduled to be updated by A_w assuming that it remains enabled during the delay, that is, the kill action A_k is disabled the mentioned time period.

The composition of timed actions A_i is:

`composition of timed actions` $A_i \mathrel{\widehat{=}}$ (*timed action composition*)

$[\ [\!] \ 1 \leq i \leq n\colon (A_{f,i} \ [\!] \ A_{k,i})]$ (*finish the operation of scheduled timed action(s)*)

$/\!/\ [\ ]\!]\ 1 \leq i \leq n\colon A_{s,i}]$ (*commence the operation of enabled timed action(s)*)
$/\!/\ Pt$ (*progress time*)

where n is the number of actions. Observe that time propagation action Pt is shared amongst the timed actions. It sets the global time to the nearest scheduled finish time. It is defined by:

$$Pt \widehat{=} [\ ]\!]\ 1 \leq i \leq n : min[i] \rightarrow gt := stateA_i.ft] \qquad (\textit{time propagation action})$$

where the guard $min[i]$ is given as:

$$min[i] \widehat{=} (stateA_i.ft > gt) \qquad (\textit{guard min})$$
$$\wedge\ (\forall j : 1 \leq j \leq n : j \neq i : stateA_j.ft > gt \Rightarrow stateA_i.ft \leq stateA_j.ft)$$

It explores the values of finish times $stateA_i.ft$ of scheduled timed actions. It evaluates to *true* if a finish time $stateA_i.ft$ of a timed action A_i is greater than gt (a requirement for a timed action being a scheduled timed action) and no other scheduled timed actions' finish time $stateA_j.ft$ is smaller than it is. In other words, it chooses the smallest scheduled finish time greater that the global time gt, which then becomes a new global time in Pt.

2.3.2 Weakest precondition of a timed action.

The weakest precondition defines the total correctness of an action with respect to its pre- and postcondition. The weakest precondition of a timed action divides into two parts depending on its enabledness during execution: (a) a timed action is enabled throughout its operation allowing write variables to be updated after the specified delay and (b) a timed action becomes disabled during execution preventing the update of the write variables and enabling the timed action for further executions. That is, it behaves as the *skip* action. Thus, the weakest precondition of a timed action is:

$$\mathbf{wp}((A[\![dA]\!]), Q) = \mathbf{wp}(A, Q[(gt + d'.dA)/gt]) \wedge (\neg gA \Rightarrow Q) \quad (\textit{wp of a timed action})$$

where the latter part is the weakest precondition of a *skip* action: $\mathbf{wp}(skip, Q)$.

2.4 Modelling the Behaviour of Systems

Let us have two timed action systems $\mathcal{A}$ and $\mathcal{E}nv$ whose local variables and actions are distinct and the latter is the environment of the former. Consider the parallel composition of these systems, denoted by $\mathcal{A} \parallel \mathcal{E}nv$. The parallel composition is defined to be another action system whose distinct global and local identifiers (variables and actions) consist of the identifiers of component systems and whose **exec** clause has the form: **do** $[\ ]\!]\ 1 \leq i \leq n\colon A_i]\ [\ ]\!]\ [\ ]\!]\ 1 \leq j \leq m\colon E_j]$ **od**, where A_i and E_j are actions of the systems $\mathcal{A}$ and $\mathcal{E}nv$, respectively. The actions are not allowed to have same labels. The constituent systems communicate via their shared interface variables. The definition of the parallel composition is used inversely in system

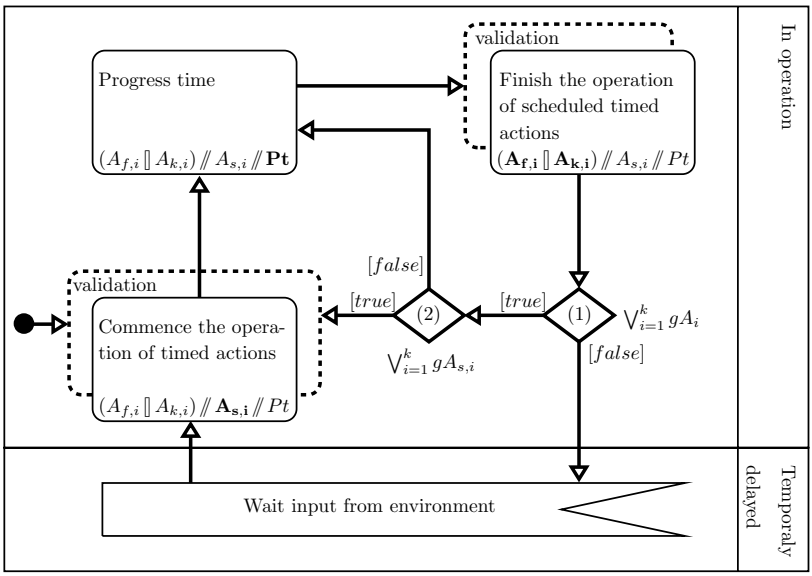


Figure 3. A computation model of a timed action system with validation states.

derivation to decompose a system description into a composition of smaller separate systems or internal subsystems. In modelling the behaviour of a system $\mathcal{A}$ and its environment $\mathcal{E}nv$, it is assumed that there is always one enabled timed action. In other words, the system must satisfy the following invariant: $I_{\mathcal{A}} \mathrel{\widehat{=}} (\vee_{i=1}^{n} gA_i) \vee (\vee_{i=1}^{m} gE)$.

Computation model (Fig. 3). The computation of a timed action system is commenced by initialising its variables (both local and global) to predefined values. In the iteration part, the **exec** section, actions are sequentially selected for execution based on the composition and enabledness of the start actions $A_{s,i}$. After all the enabled start actions are executed, the global time gt is set to nearest scheduled finish time in the time propagation action Pt. Then, either finish $A_{f,i}$ or kill $A_{k,i}$ action of those scheduled timed actions whose delay is consumed are executed. This is repeated as long as there are either enabled (1) or scheduled (2) timed actions. However, if there are no such timed actions, the timed action system is considered *temporary delayed*. The computation resumes execution when an environment enables an action via the interface variables. The dotted boxes around the commence and end states denotes the validation of design constraints that are discussed in Sect. 3.2.

Example 2.2 In **Example** 2.1 we introduced an action M. Let us place that action into a timed action system $\mathcal{M}ult$ below. The system $\mathcal{M}ult$ is operating in an environment $\mathcal{E}nv$ that updates the input variables and enables and disables the computation using a boolean variable en, and, finally, it reads the result of the computation. We have:

sys $\mathcal{M}ult$ ($x, y, prod$: $data$; en: $bool$;) ::
|[**actions** $M[\![d_{min}, d_{max}]\!]$: $(en \rightarrow prod := x * y)$;
init $en, x, y, prod := F, 0, 0, 0$;
exec **do** M **od**]|

where the non-deterministic delay defines the minimum d_{min} and maximum d_{max} operation times for the multiplication. The chosen delay type, in this example, is used to denote data-dependent delay, that is, a delay whose value depends on the values of the operands. The delay values without knowing, for example, a multiplication algorithm or a production technology are conservative ones. In this paper, however, we do not enlarge upon the algorithms nor the production technology.

3 Temporality

On showing the correctness of a trace refinement (introduced in the next section), we introduce rules to calculate delays for timed action compositions, and, furthermore, constraints with which the operation of timed action can be restricted, not only logically but also temporally.

3.1 Delay

Delay calculation rules for timed actions and their compositions are defined by:

$$\Delta(A) \mathrel{\widehat{=}} d'.dA = A.ft - A.st \qquad (\textit{action delay}) \qquad (5)$$

$$\Delta(A_1; A_2) \mathrel{\widehat{=}} \Delta(A_1) + \Delta(A_2) \qquad (\textit{sequential delay}) \qquad (6)$$

$$\Delta(p \rightarrow A) \mathrel{\widehat{=}} \Delta([p]; A) = \Delta([p]) + \Delta(A) \qquad (\textit{guarded action delay}) \qquad (7)$$

$$\Delta(A_1 \ [] \ A_2) \mathrel{\widehat{=}} \{\Delta(A_1), \Delta(A_2)\} \qquad (\textit{alternative delay}) \qquad (8)$$

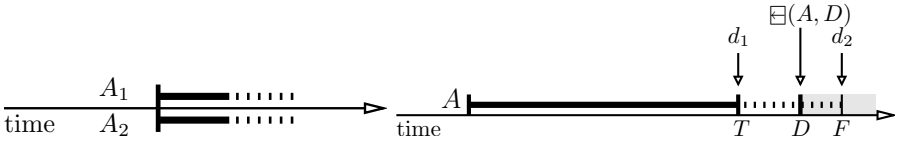
$$\Delta(A_1 \ // \ A_2) \mathrel{\widehat{=}} \{\Delta(A_1), \Delta(A_2)\} \qquad (\textit{alternative delay}) \qquad (9)$$

$$\Delta(|[\textbf{var } x := x_0; \ A]|) \mathrel{\widehat{=}} \Delta(A) \qquad (\textit{block delay}) \qquad (10)$$

$$\Delta(\textbf{do } A \textbf{ od}) \mathrel{\widehat{=}} \sum_{i=k}^{0} \Delta(A) \qquad (\textit{iterative delay}) \qquad (11)$$

where (5) defines a timed action delay. The delay is also defined using the start and finish times of a timed action; (6) sums the delays of sequentially executed timed actions; (7) defines a delay for a guarded timed action. It consists of two components based on the definition of a guarded action: the evaluation of the guard and the operation time; (8) and (9) gives a set of delays each of which reflect an alternative delay path. To extract either the best or worst case propagation delay, one may use the **Min** or **Max** functions, respectively. In critical timing path analysis one may utilise the **Max** function to observe the slowest path from input to output; (10) defines a delay for a block of timed actions; Finally, in (11) is defined the delay for the iterative composition. It equals the sum of the delays of those timed actions which are executed in k iterations. The definition is justified by the weakest precondition of the iterative composition given earlier. It states that after k selections the **do-od** loop will terminate properly.

Using the above delay calculation rules we are able to define *a static delay* for a system under design. In the static delay analysis the expected timing information of a system is computed without requiring simulations. The static delay analysis returns a set of delays each of which corresponds a possible computation path, a path between two timed actions where no loops are allowed. From the obtained set,



(a) A temporal relation **starts with**. (b) A timed action A satisfying (T) and dissatisfying (F) a deadline.

Figure 4. A graphical representation of the **starts with** temporal relation and a deadline.

it is possible to calculate, for example, the worst case delay for a system the **Max** function or the best case delay using the **Min** function.

3.2 Constraints

A constraint is an expression according to which involved timed actions are obliged to operate. Hence, violation of constraints, boolean conditions, denote a useless, unpredictable computation.

As stated above constraints are boolean conditions, for example, on the relative or measurable properties of timed actions. A relative constraint defines how timed actions interact with each other from the time point of view, for example, a relative constraint **starts with** (Fig. 4(a)) requires the start times of timed actions be equal: A_1 **meets** $A_2 \mathrel{\widehat{=}} A_1.st = A_2.st$. In text we use a symbol $\square$ to denote a constraint. For the given example we have: $\square\{A_1$ **meets** $A_2\}$. A measurable constraint, on the other hand, examines the operation time of timed actions, for example, a measurable constraint *deadline* [12] defines a time point upon which the operation of involved timed actions are obliged to finish their operation. For example, a deadline for a timed action A is: $\square\{\Delta(A) \leq D\}$. In Fig. 4(b) is denoted the correct execution ($A.ft = d_1$) by T and the false execution ($A.ft = d_2$) by F. The shaded area in Fig. 4(b) denotes a false computation area for the timed action A, that is, in that area we have $d_2 > D$. We use a symbol $\boxminus$ to denote a deadline constraint. Thus, for the given example we have: $\boxminus\{A, D\}$.

The tenability of a constraint is confirmed in either of the dotted boxes shown in Fig. 3 depending on the validated property. Constraints are introduced in the declarative part of a timed action systems in the **constraints** clause. Constraints are validated against *observation points* in *validation points*. The observation point of a constraint is, for example, the time point when the first referred timed action in a constraint commence or finish its operation. For the deadline defined above the observation point is the start time $A.st$. The validation point, on the other hand, is defined by the last referred timed action in the constraints. For the deadline the validation point is $A.ft$.

To give a special status for a constraint, it can be defined as an assert statement. In such a case it behaves as a *skip* statement if it holds ($B \equiv true$) but otherwise it behaves like *abort* ($B \equiv false$). In other words, if constraints are satisfied they operate as empty statements that do not change the state at all. On the other hand, if a constraint is *not* satisfied, it is a never terminating statement, which hence does not establish any postconditions causing an abnormal termination of the system.

4 Refinement

Conventional Action Systems are meant to be designed in a stepwise manner within the *refinement calculus* framework [4]. The *refinement calculus* preserves the correctness of actions during refinement procedure. In this section we concentrate on applying the well-founded refinement calculus framework for timed actions. As presented in the previous section the timed notation is a clear extension of the untimed model. Therefore, we adopt the refinement calculus of conventional actions, and extend it to the time domain.

An (atomic) action A is said to be *(correctly) refined* by action C, denoted $A \leq C$, if the following property holds:

$$\forall Q.(\mathrm{wp}(A, Q) \Rightarrow \mathrm{wp}(C, Q)) \qquad (\textit{refinement condition of a conventional action})$$

This is equivalent to the condition

$$\forall P, Q.((P\ A\ Q) \Rightarrow (P\ C\ Q)) \qquad (\textit{total correctness property})$$

which means that the *concrete* action C preserves every total correctness property of the *abstract* action A.

4.1 Data Refinement of Conventional Actions

In a *data refinement* an abstract action A on the variables a and u is refined by a concrete action C on the variables c and u using an abstraction invariant $R(a, c, u)$, which is a boolean relation between the abstract variables a and the concrete variables c. The action A is *data-refined* by the action C, denoted $A \leq_R C$, if the following condition holds:

$$\forall Q.(R \wedge \mathrm{wp}(A, Q) \Rightarrow \mathrm{wp}(C, \exists a.R \wedge Q)) \qquad (\textit{condition of data refinement})$$

holds. The predicate $\exists a.R \wedge Q$ is a boolean condition on the program variables a and c. The above definition of data-refinement can be written in terms of the guards gA, gC and bodies sA, sC of the actions A and C as follows:

$$R \wedge gC \Rightarrow gA \qquad (\textit{body}) \quad \text{(ii)}$$
$$\forall Q.(R \wedge gC \wedge \mathrm{wp}(sA, Q) \Rightarrow \mathrm{wp}(sC, \exists a.R \wedge Q)) \qquad (\textit{guard}) \quad \text{(iii)}$$

The data refinement $A \leq_R C$ replaces a with c preserving the variables u. Naturally, if we do not replace any variables and delays but maybe just add some new ones, we have $R \equiv true$, and hence R can be omitted from the refinement proof.

The above presented data refinement rule will be used also for timed actions in the next section where a trace refinement of a timed action system is introduced. We will not introduce a data refinement rule for a timed action as there is no use for it due to the chosen modelling approach presented in Sect. 2.4, and that constraints,

which have to be considered in the time aware refinement, are given in a system specification and are therefore considered system properties. For short, we denote the constraints of a timed action system, say $\mathcal{A}$, by $\Box\{\mathcal{A}\} = C_1 \wedge \ldots \wedge C_n$, where C_i, $1 \leq i \leq n$, are the constraint of the timed action system $\mathcal{A}$.

4.2 Trace Refinement

The idea in refining an open action system within some environment is to preserve the *traces*, or sequence of global states (observable behaviour), of the system in question. Consequently, such a transformation is often referred to as a trace refinement of the system. A fundamental study of the trace theory can be found in [3]. Rather than going into the details of the trace refinement theory, we apply here a commonly-used method to prove a trace refinement between two systems by means of data refinement on establishing a foundation for the trace refinement of a timed action system. We will address those points that coincide with the conventional presentation, and argue that they are suitable for timed actions as well.

Consider timed action systems $\mathcal{A}$ and $\mathcal{C}$:

sys $\mathcal{A}$ (g;) ::
|[**constraints** $\Box\{\mathcal{A}\}$;
delays dA;
var a;
actions $A[\![dA]\!]$: (aA);
init $g, a := g0, a0$;
exec **do** A **od**]|

sys $\mathcal{C}$ (g;) ::
|[**constraints** $\Box\{\mathcal{C}\}$;
delays dC, dX;
var c;
actions $C[\![dC]\!]$: (aC);
$X[\![dX]\!]$: (aX);
init $g, c := g0, c0$;
exec **do** C [] X **od**]|

where aA, aC and aX are any of the atomic actions defined earlier and $\Box$ defines the constraints posed on the systems.

Let $R(a, c)$ be a relation between the local variables a and c, and $I(c, u)$ is an invariant of the system $\mathcal{C}$.

The abstract timed action system $\mathcal{A}$ is (trace) refined by the concrete timed action system $\mathcal{C}$, denoted by $\mathcal{A} \sqsubseteq \mathcal{C}$, if there exists relations $R(a, c)$ such that:

$R(a_0, c_0) \wedge I(a_0, c_0) = true$	(*initialisation*)	(i)
$A \leq_{R,I} C$	(*main action*)	(ii)
$skip \leq_{R,I} X$	(*auxiliary action*)	(iii)
$R \wedge I \wedge gA \Rightarrow gC \vee gX$	(*continuation condition*)	(iv)
$R \wedge I \Rightarrow \mathbf{wp}(\mathbf{do}\ X\ \mathbf{od}, true)$	(*internal convergence*)	(v)
$R \wedge \Box\{\mathcal{C}\} \Rightarrow \Box\{\mathcal{A}\}$	(*timing behaviour*)	(vi)

The *first* condition (i) says that the initialisation of the systems $\mathcal{A}$ and $\mathcal{C}$ establish the abstraction relation R. The *second* condition (ii) requires the abstract action A to be data-refined by the concrete action C using R. The *third* condition (iii), in turn, indicates that the auxiliary action X is obtained by data-refining a *skip* action. This basically means that X behaves like *skip* action with respect to the global variables u which are not allowed to be changed in the refinement. The

fourth condition (iv) requires that whenever the action A of the abstract system $\mathcal{A}$ is enabled, assuming the abstract relation R holds, there must be a enabled action in the concrete system $\mathcal{C}$ as well. The *fifth* condition (v) states that if R holds, the execution of the auxiliary action X, taken separately, must terminate at some point. Finally, the *sixth* condition, the only timing related condition, requires that all the time constraints are met in the concrete timed action system $\mathcal{C}$. That is, both the functional and temporal behaviour of the concrete system must adhere to the given constraints after the performed refinement step as otherwise the temporal characteristics of the refined system cannot be guaranteed.

In the trace refinement of a timed action system introduced above, we adopted the first five conditions (i) - (v) as such from the trace refinement of a conventional action system. This approach is justified by the fact that Timed Action Systems extends conventional Action Systems by defining a delay that determines the time after which the result is written onto write variables. The operation part, functionality, of a conventional action is not altered.

An important point in the trace refinement of a timed action system is that we have taken an approach to keep the proof obligations as simple as possible. That is, we do not pose any direct requirements on how the delays of timed actions are refined. Observe, however, that timing is confirmed in the last condition where the tenability of the temporal requirements obtained from a system specification is confirmed. In other words, the correctness of the refinement from the time point of view is ensured by showing that all the timing obligations are met in the concrete system $\mathcal{C}$.

As already stated the functionality of a timed action is refined using data refinement of conventional actions. Next we show the correctness of the data refinement of timed actions. We need to prove that:

$$A \leq_R C \Rightarrow A[\![dA]\!] \leq_R C[\![dC]\!]$$

holds.

On the proof we assume that a timed action is not disabled during its operation, in other words, the timed action remains enabled through its operation time. This justifies us to use the following timed action model $A_{dA} \widehat{=} A_o; Time; A_w$, where $A_o \mathrel{\widehat{=}} A_{o,1}; A_{o,2}$, where $A_{o,1} \mathrel{\widehat{=}} SA := (wA, gt, gt + d'.dA)$ and $A_{o,2} \mathrel{\widehat{=}} A[SA.wA/wA]$; $Time \mathrel{\widehat{=}} gt := SA.ft$ and $A_w \mathrel{\widehat{=}} wA := SA.wA$ using the following abbreviations $stateA = SA$ and $stateC = SC$ to clarify the representation. Assume $A \leq_R C$. Then:

$$
\begin{array}{ll}
 & \mathbf{wp}(A_{dA},\ Q) \\
\Leftrightarrow & \{\text{timed action model } A_{dA}\} \\
 & \mathbf{wp}(A_o; Time; A_w,\ Q) \\
\Leftrightarrow & \{\text{definition of } A_o\} \\
 & \mathbf{wp}(A_{o,1}; A_{o,2}; Time; A_w,\ Q) \\
\Leftrightarrow & \{\text{definition of } A_{o,1}, A_{o,2}, Time \text{ and } A_w\}
\end{array}
$$

$\mathbf{wp}(SA := (wA, gt, gt + d'.dA); A[SA.wA/wA]; gt := SA.ft; wA := SA.wA,\ \ Q)$
$\Rightarrow$ {monotonicity of ';', assumption $A \leq_R C$}
$\mathbf{wp}(SC := (wC, gt, gt + d'.dC); C[SC.wC/wC]; gt := SC.ft; wC := SC.wC,\ \ Q)$
$\Leftrightarrow$ {definition of $C_{o,1}, C_{o,2}, Time$ and C_w}
$\mathbf{wp}(C_o; Time[SC.ft/SA.ft]; C_w,\ \ Q)$
$\Leftrightarrow$ {timed action model $C_{dC} \mathrel{\widehat{=}} C_o; Time; C_w$}
$\mathbf{wp}(C_{dC},\ \ Q)$

Example 4.1 Consider the following timed action system $\mathcal{M}ult$ presented earlier in **Example** 2.2. Let us assume that a designer, based on thorough consideration chooses the best multiplication algorithm that fulfils the timing obligations given in the specification. The designer is not willing to decrease the abstraction level at this point of the design cycle, and therefore the designer only change timing information in a trace refinement of the system. Thus, we perform a refinement $M[\![d_{min}, d_{max}]\!] \leq M[\![d'_{min}, d'_{max}]\!]$.

On showing the correctness of this refinement step we are only required to validate the first and last refinement requirements.

(i) *Initialisation.* The initialisation of the action system $\mathcal{M}ult$ does not contradict with the initialisation of the action system $\mathcal{M}ult'$.

(ii) *Timing behaviour.* The maximum allowed operation time for the timed action M is D: $\boxminus\{M, D\}$. We require that $d'_{min}, d'_{max} \in [d_{min}, d_{max}]$. Thus, the time constraint is satisfied, because based on the requirement it would not hold in the abstract timed action system $\mathcal{M}ult$.

Thus, we have performed a trace refinement $\mathcal{M}ult \sqsubseteq \mathcal{M}ult'$.

Example 4.2 Consider a timed action system $\mathcal{M}ult'$ presented in the previous example. The designer decompose the timed action M into two separate timed actions by introducing a new intermediate variable t and a boolean variable b that sequence to operation of the new timed actions. After the refinement we have a timed action system $\mathcal{M}ult''$ whose operation is given by: **do** $M_1 \,[\!]\, M_2$ **od**, where two timed actions M_1 and M_2 are as follows: $M_1[\![d^1_{min}, d^1_{max}]\!] \mathrel{\widehat{=}} \neg b \rightarrow u := w; t := x * y; b := T;$ and $M_2[\![d_2]\!] \mathrel{\widehat{=}} b \rightarrow prod, b := t, F;$.

On showing the correctness of this refinement we need to prove that all the condition of timed trace refinement are satisfied. We have:

(i) *Initialisation.* The initialisation of the action system $\mathcal{M}ult''$ does not contradict with the initialisation of the action system $\mathcal{M}ult'$.

(ii) *Main action.* Our goal is to prove that $M \leq_I M_2$, where I is an invariant of form: $I \mathrel{\widehat{=}} b \Rightarrow t = prod$. We have:

Guard: $I \wedge gM_2 \Rightarrow gM$
$\Leftrightarrow I \wedge b \Rightarrow true$
$\Leftrightarrow true$

$$
\begin{aligned}
\textit{Body:}\ \ & I \wedge gM_2 \wedge \mathrm{wp}(sA, Q) \Rightarrow \mathrm{wp}(sM_2, I \wedge Q) \\
& \Leftrightarrow \{\text{weakest preconditions of } sM \text{ and } sM_2\} \\
& \quad I \wedge b \wedge Q \Rightarrow I[F/b] \wedge Q[t/prod] \\
& \Leftrightarrow \{\text{the invariant } I \widehat{=} b \Rightarrow t = prod\} \\
& \quad (b \Rightarrow t = prod) \wedge b \wedge Q \Rightarrow (b \Rightarrow t = prod)[F/b] \wedge Q[u/w] \\
& \Leftrightarrow \{\text{logic}\} \\
& \quad (b \Rightarrow t = prod) \wedge b \wedge Q \Rightarrow T \wedge Q[t/prod] \\
& \Leftrightarrow \{\text{logic}\} \\
& \quad b \wedge t = prod \wedge Q \Rightarrow Q[t/prod] \\
& \Leftarrow b \wedge Q \Rightarrow Q[t/prod] \wedge t = prod \\
& \Leftrightarrow b \wedge Q \Rightarrow Q \\
& \Leftrightarrow true
\end{aligned}
$$

Thus, we have shown that $M \leq_I M_2$ holds.

(iii) *Auxiliary action.* Because the auxiliary action M_1 does not modify any interface variables, it behaves like *skip* with respect to this kind of variables:

(iv) *Continuation condition.* There is always either of the new timed actions M_1 or M_2 enabled when the original timed action A is enabled.

(v) *Internal convergence.* Holds trivially as the new auxiliary action M_1 disables itself.

(vi) *Timing behaviour.* The maximum allowed operation time for the timed action M is D: $\boxminus\{M, D\}$. Although we decompose the action into two parts the scope of the deadline remains unchanged, that is, the observation point is the time point at when M_1 commence its operation and the validation point is the time point at when M_2 finish its operation. In calculating the delay for the composition we use the delay calculation rule (6), although the composition suggests rule (8). This is justified by the interaction of the refined timed actions: M_1 enables M_2. By requiring that $d^1_{max} + d_2 \leq d$ the deadline is satisfied.

Thus, we have performed a trace refinement $\mathcal{M}' \sqsubseteq \mathcal{M}''$.

The keep the constraints up-to-date they are changed to meet the new action definitions. In our example the deadline thus becomes $\boxminus\{(M_1; M_2), D\}$ due to the fact that the former timed action enables the latter. We do not see this as a refinement, because the constraint itself is not changed, that is, the scope of a constraint remains the same as pointed earlier.

5 Conclusions

In this paper we introduced a method to develop, in a stepwise manner, an abstract system towards a more concrete one. One of the advantages of the defined refinement rules is that it is a clear extension to existing refinement rules for conventional action systems, and therefore it is easily adopted. We showed, using a simple example, how a timed action system is stepwisely refined into a form where the operation is

sequenced by a new auxiliary boolean variable.

In this paper we did not considered scalability of our timed formalism, as the scope was the decomposition of an atomic construct in the time domain. However, the introduced timed refinement rule is an important step towards that direction, as we are now able to start development of a system from an abstract specification and refine it towards a more concrete one in a stepwise manner preserving both the temporal and functional characteristics.

References

[1] R.-J. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Procs. of the 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[2] 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 comp. sci., Springer-Verlag.

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

[4] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

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

[6] I. J. Hayes. The real-time refinement calculus: A foundation for machine-independent real-time programming. In *ICATPN '02: Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, pages 44–58, London, UK, June 2002. Springer-Verlag.

[7] M. E. Majster-Cederbaum, J. Wu, and H. Yue. Refinement of actions for real-time concurrent systems with causal ambiguity. *Acta Inf.*, 42(6-7):389–418, 2006.

[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.

[10] E. Sekerinski and K. Sere. A theory of prioritizing composition. *The Computer Journal*, 39(8):701–712, 1996. The British Computer Society. Oxford University Press.

[11] T. Westerlund and J. Plosila. Formal timing model for hardware components. In *Proceedings of the 22nd NORCHIP Conference*, pages 293–296, Norway, Nov 2004.

[12] T. Westerlund and J. Plosila. Back-annotation of timing information into a formal hardware model: A case study. In *International Symposium on Signals, Circuits, and Systems - ISSCS 2005*, page to appear, Romania, July 2005.

[13] T. Zheng, F. Khendek, and B. Parreaux. Refining timed mscs. In *SDL Forum*, volume 2708 of *Lecture Notes in Computer Science*, pages 234–250. Springer, 2003.