Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 201 (2008) 223–253
www.elsevier.com/locate/entcs

Power Aware System Refinement

Johanna Tuominen [1,2] , Tomi Westerlund [2] and Juha Plosila [3]

[1] *Turku Centre for Computer Science, Finland*
[2] *Dept, of Information Technology, University of Turku, Finland*
[3] *Academy of Finland, Research Council for Natural Sciences and Engineering*

Abstract

We propose a formal, power aware refinement of systems. The proposed approach lays its foundation to the traditional refinement calculus of Action Systems and its direct extension, time wise refinement method. The adaptation provides well-founded mathematical basis for the systems modeled with the Timed Action Systems formalism. In the refinement of an abstract system into more concrete one a designer must that show conditions of both functional and temporal properties, and furthermore, power related issues are satisfied.

Keywords: refinement, time, power, Timed Action Systems

1 Introduction

Advances in technology over recent years have considerably decreased both physical and functional size of single chip systems. Moreover, with the advent of wireless and mobile high performance computing platforms, and the limited operational life time of batteries, the low-power designs are required. To estimate the power consumption, there is a trade-off between the accuracy and the abstraction level of detail which the system is analyzed. The more detailed the description, the more accurate the simulation will be, but on the other hand the more time consuming it will be. Moreover, a designer wants to make decisions as early as possible in the design flow to avoid costly design backtracking.

Formal methods provide an environment to design, analyze, and verify digital hardware with the benefits of rigorous mathematical basis. The base formalism in our study is Action Systems [5], henceforward conventional Ac-

doi:10.1016/j.entcs.2008.02.022

tion Systems, which is based on an extended version of Dijkstra's language of guarded commands [14]. We continue to base our research on the conventional Action Systems formalism and its timed extension [22]. The timing information allows the modeling and analysis of both logical and temporal aspects of both synchronous [18] and asynchronous [17] systems, and, furthermore, time is an essential concept for modeling power consumption.

In this study, we introduce a power modeling methodology in the Timed Action Systems context. It allows us to estimate energy and power consumption in a formal, abstract models, and to steer the development towards power-efficient implementations. Furthermore, we extend the refinement calculus for Timed Action Systems by introducing performance related constraints to it. The final goal in out research is to have modeling framework for SoC/NoC (System-on-chip/Network-on-Chip) systems in which, within one formalism, a system can be modeled from a specification down to an implementation model. The development of the systems starts from the high level system model $\mathcal{S}$ whose functional and temporal characteristics are given in the specification. The refinement relations between system models can be defined so that the total correctness is preserved: if $\mathcal{S} \sqsubseteq \mathcal{S}'$ and pSq then $pS'q$ will also hold. After several successive refinement steps ($\mathcal{S} \sqsubseteq \mathcal{S}' \ldots \sqsubseteq \mathcal{S}^n$) we obtain an implementable specification [3]. Thus, we have developed, in a stepwise manner, an implementation of the original specification. The purpose is to be able to model SoC/NoC designs within one formalism from a specification down to an implementable model by taking into account the correctness of both the temporal and functional characteristics, and, furthermore, the power modeling issues.

2 Timed Action Systems

In this section we shortly review both the conventional and timed actions that form the formal basis for out power analysis.

2.1 Conventional Actions

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

$A ::=$ *abort*	(*abortion, non-termination*)
$\mid$ *skip*	(*empty statement*)
$\mid x := x'.R$	(*non-deterministic assignment*)
$\mid x := e$	(*(multiple) assignment*)
$\mid \{p\}$	(*assertion statement*)
$\mid [p]$	(*assumption statement*)
$\mid p \to A$	(*guarded action*)
$\mid A_1 \ [] \ A_2$	(*non-deterministic choice*)
$\mid A_1; A_2$	(*sequential composition*)
$\mid A_1 \ /\!/ \ A_2$	(*prioritized composition*)
$\mid$ **do** A **od**	(*iterative composition*)

where A and A_i, $i = 1, 2$, are actions; x is a variable or a list of variables; 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: $vA \mathrel{\widehat{=}} wA \cup rA$.

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 prioritized (' /// ') composition [19] is a composition in which the execution order of enabled actions is prioritized. We have: $A \ /\!/\!/ \ B \mathrel{\widehat{=}} A \ [] \ \neg gA \to B$, where the highest priority belongs to the leftmost action in the composition; thus, the leftmost enabled action is always chosen for execution.

Body P of the procedure $p : p(\mathbf{in}\ x; \mathbf{out}\ y) : P$, is in general any atomic action A, possibly with some auxiliary local variables w initialized to $w0$ every time the procedure is called. The action A accesses the global and local variables g and l of the host/enclosing system and the formal parameters x and y. Hence, the body P can be generally defined by: $P[\mathbf{var}\ v; \mathbf{init}\ w := w0; A(g, l, w, x, y)]$. If there are no local variables, the begin-end brackets [ ] can be removed together: $[A(g, l, x, y] = A(g, l, x, y)$. If there are neither local variables nor parameters, the action only accesses the global and local variables of the host system, then the procedure p can be written as: $\mathbf{proc}\ p : A(g, l)$.

2.2 Semantics of actions

An action is considered to be atomic, which means that only the initial and final states are observed by the system. Therefore, when action is selected for execution, it is completed without any interference from other actions. The actions are defined using weakest precondition for predicate transformers [14]. The *weakest precondition* for action A to establish the post condition q is defined for example: $\mathbf{wp}(abort, q) = false$, $\mathbf{wp}(skip, q) = q$ and $\mathbf{wp}((A \;[\!]\; B), q) = \mathbf{wp}(A, q) \wedge \mathbf{wp}(B, q)$. The guard gA of an action A is defined by $gA \mathrel{\widehat{=}} \neg\mathbf{wp}(A, false)$. Considering a guarded action $A \mathrel{\widehat{=}} P \rightarrow B$ we have that $gA \mathrel{\widehat{=}} P \wedge gB$. An action A is said to be *enabled* in some state, if its guard is *true* (T) in that state, otherwise *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: $sA \mathrel{\widehat{=}} A \;[\!]\; \neg gA \rightarrow abort$. Moreover, for assert statement, we have that $\mathbf{wp}(\{p\}, q) = (p \wedge q)$ and for the assumption statement we have: $\mathbf{wp}([p], q) = p \rightarrow q$.

2.3 Timed Action System

Above we introduced conventional Action Systems, which is the formal basis of our timed notation. In conventional Action Systems computation does not take time, a reaction is instantaneous – and therefore atomic in any possible sense. Atomicity means that only the pre- and post-states of actions are observable, and when they are chosen for execution they cannot be interrupted by external counterparts. In modeling 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, and, furthermore, time is an essential element in power modeling. It should be observed that the complexity of a timed action is not restricted, and thus the operation time is not bounded either. Let us next introduce the form of a timed action system [22].

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

sys $\mathcal{A}$ (**imp** p_I; **exp** p_E;)(u;) ::
|[**delay** dA_i, dp;
constraint $\{\mathcal{A}\}$;
var l;
private proc $p[\![dp]\!](x)$: (P);
public proc $p_E[\![dp_E]\!](x)$: (P_E);
action $A_i[\![dA_i]\!]$: (aA_i); **init** g, l: $g0, l0$;
exec **do** [[] $1 \leq i \leq n$: A_i] **od**]|

where we can identify three main sections: (1) *interface*, (2) *declaration*, and (3) *iteration*:

(1) The interface part declares those variable, u, that are visible outside the action system boundaries, and thus accessible by another systems. It also introduces communication procedures that are either imported or exported by the system. The exported procedures are introduced by the system itself and the imported ones are introduced by some other systems. If a timed action system does not have any interface variables or procedures, it is a *closed* system, otherwise it is an *open* system.

(2) The declaration part declares the building blocks of the systems. In the **delay** clause it defines the delays of the timed actions, which are associated with a timed action using delay brackets $[\![\ ]\!]$. The most commonly used delays are deterministic and bounded, non-deterministic delays denoted by dA and $dA\colon [dA_{min}, dA_{max}]$, respectively. The former defines a precise delay, whereas the latter one a delay whose value is chosen non-deterministically between the given interval. The time domain is dense and continuous $\mathbb{T} = \mathbb{R}_{\geq 0}$, because it is a natural model for systems operating over continuous time. In the **constraint** clause is defined the functional and non-functional requirements of the system, and in the **var** clause is defined the local variables l. Then in the **private** and **public procedure** clauses is defined local p and exported p_E procedures. The private (local) procedures are used only within the system, whereas the public ones are used in communication with other systems. In the **action** clause is given the definitions of the action A_i that perform operations (aA_i) on local and global variables. The start time of a timed action is denoted by $A.st$ and the finish time by $A.ft$ whose separation in time is dA. The last item is the **init** clause that set the system in a safe state from which the system may commence its operation.

(3) Finally, the iteration section, the **exec do-od** loop of the system, contains the composition of the actions defined in the declarative part. The computation of a timed action system is started in an initialization in which the variables (both global and local ones) are set to predefined values. In the iteration part, the actions are selected for execution based on the composition and enabledness of the timed actions. If there are no such timed actions, the timed action system is considered temporary delayed. The computation resumes to execution when some other timed action system enables one of the action via interface variables or procedure calls.

2.4 Parallel Composition of Action Systems

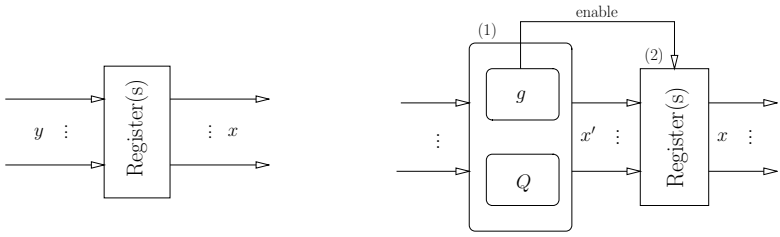
Consider two action systems $\mathcal{A}$ and $\mathcal{B}$ whose local variables are distinct, $l_A \cap l_B = \emptyset$, and communication variables are a set $u_A \cap u_B$. We require that the initializations of the communication variables $u_A \cap u_B$ are consistent with each other, so that the initial values are equivalent: $\forall v \in u_A \cap u_B.(v0_A = v0_B)$, where $v0_A \in u0_A$ and $v0_B \in u0_B$. The parallel composition of $\mathcal{A}$ and $\mathcal{B}$, denoted $\mathcal{A}\|\mathcal{B}$ is defined to be another action system whose global and local identifiers (procedures, variables, actions) consist of the identifiers of the component systems and whose **exec**-clause has the form: **exec do** $A \,[\!]\, B$ **od**, where A and B are the actions of the systems $\mathcal{A}$ and $\mathcal{B}$, respectively. The constituent systems communicate via their shared interface procedures. The definition of the parallel composition is used inversely in system derivation to decompose a system description into a composition of smaller separate systems or internal subsystems.

2.5 Procedure based communication

The procedure based communication [17] uses remote procedures to model communication channels between action systems. Consider the timed action systems $\mathcal{S}nd$ and $\mathcal{R}ec$ whose internal activities are denoted with actions $S \mathrel{\widehat{=}} (S_1;\ \textbf{call}\ p(l_s); S_2)$ and $R \mathrel{\widehat{=}} (R_1;\ \textbf{await}\ p; R_2)$, where p is an interface procedure defined in and exported by the receiver ($\mathcal{R}ec$), and imported and called by the sender ($\mathcal{S}nd$), with the variables l_s (sender's local variables) as actual parameters. The body P of p can be any atomic action writing onto variables l_r (receiver's local variables). The **exec**-clause is of the composed system $\mathcal{S}nd\|\mathcal{R}ec$ has the form: **do** S $[\!]$ R **od**. The construct $S \;[\!]\; R$, where S calls p (**call** command) and R awaits such a call (**await** command), is regarded as a single atomic action SR, defined by: $SR \mathrel{\widehat{=}} (S_1; R_1; P[l_s/x]; R_2; S_2)$. Hence, communication is based on sharing an action in which data is atomically passed from $\mathcal{S}nd$ to $\mathcal{R}ec$ by executing the body P of the procedure p hiding the details of the communication into the procedure call.

3 Performance modeling

In this section, we present a performance modeling framework for hardware systems by introducing how to model a size of an action system (timed or untimed one). Henceforward, we denote the size as area complexity. The purpose is to develop a framework that can be used to identify which one of the two arbitrary actions is larger, and therefore to make difference in energy consumption between these two actions. Furthermore, the energy of actions



(a) Substitution operation $x := y$

(b) Guarded non-deterministic assignment $g \to x := x'.Q$

Fig. 1. Illustrations of assignments.

is adopted to model power consumption of the formal system description. When reasoning power consumption, the Timed Action Systems formalism is assumed.

3.1 Area complexity

The preliminary constraint in our modeling approach is that we consider the *non-deterministic assignment* as a base action in our complexity model. The non-deterministic assignment, also know as a specification statement, is the generalization of the assignment operation. In other words, it can be used to describe any operations targeted to variables in action context. Furthermore, by adopting the guard g, we obtain a *guarded non-deterministic assignment*, which is executed when its guard evaluates to true. In addition, an action is considered atomic, which means that only its initial and final states are observable. That is, we utilize the information received from the set of read (input) and set of write (output) variables. We start the analysis by giving some illustrations for the internal structure of an action. In Fig. 1 (a) we have a substitution operation and in Fig 1 (b) a guarded non-deterministic assignment. Observe that, the non-deterministic assignment can be illustrated as depicted in Fig. 1 (b) by excluding the guard g.

3.1.1 Widths of types

Consider an (atomic) action A. We assume that each variable $v \in vA$ has a width w_v which defines *the number of bits needed to present the variable.* For example, consider an integer type of variable v, which is defined by the interval $(1..m)$, where $m \in \mathbb{R}^+$. The width is calculated by: $2^{w_v} = m \Rightarrow w_v = \lceil \log_2(m) \rceil$, $(m > 1)$. In this paper, we consider only fixed point numbers, and, furthermore, constants are considered as variables in the modeling process.

3.1.2 Complexity of a substitution operation

Let an action A be a substitution $A \mathrel{\widehat{=}} x := v$. Its write set is $wA = \{x\}$, and it is utilized to model the area complexity. If x and v are single variables of the

same type then the area complexity of A is the width w_x of the variable x. If x and v are lists of variables then the complexity is defined by the sum of the widths of the variables x. The area complexity of the substitution operation is defined by: $C(wA) \mathrel{\widehat{=}} \sum_{x \in wA} w_x$.

3.1.3 Complexity of assignment statements

The assignment statements often contains operations that are Boolean functions at lower abstraction levels, such as, addition and multiplication. These operations are combinatorial logic at lower abstraction levels, and therefore, we cannot illustrate assignments with the chain of registers as we did with substitution operations. That is, to model the structure of combinatorial logic requires, that the depth of the resulting circuit is taken into account in the area complexity model.

The idea behind the area complexity modeling of assignments arises from the graphical modeling of Boolean functions. Boolean functions can be represented as a directed acyclic graph where the size of the function can be evaluated by calculating the number of nodes needed to present the function. These graphs are often denoted by *Binary decision diagrams (BDD)* and described, for instance, in [2], [9], [10]. In BDD's each node has at least the following properties: index if it is a non-terminal node, value 0 or 1 if it is a terminal node. Furthermore, each non-terminal node has two children [2]. There are several extensions for the basic BDD's. For instance, in Boolean Expression Diagrams [1] there are three node types instead of the two presented above: a terminal node has a value 0 or 1, a variable node has a Boolean variable and operator node has a Boolean operator. Observe that the variable and operator nodes correspond the non-terminal nodes in the basic BDD description. However, these graph based presentations are not directly applicable to our approach because we do not have a exact Boolean function that could be use to generate the graph. In other words, our approach has a higher abstraction level. For instance, for an action $A \mathrel{\widehat{=}} x := x'.(x' = a + b)$, which adds together two variables a and b, we do not have a the detailed description of its implementation. That is we do not know, for instance, whether it is implemented using full adders or carry look ahead adders.

Another approach to model the size of the Boolean functions is to assume that each Boolean expression with m arguments has a size 2^m [9]. This would be more suitable approach for us, since we know the number of the input arguments for each action, that is *the sum of the widths of the variables in the read set*. However, the size expression grows exponentially, which gives rather pessimistic results when the value of m increases. For instance, if the number of input arguments is 64 ($m = 64$) then the size estimate would be

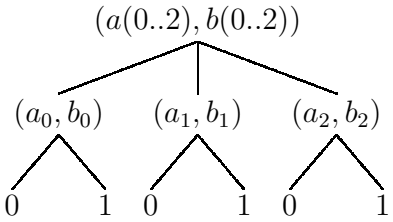


Fig. 2. Exemplification of the graph presentation.

(approximately) $1.84 \cdot 10^{19}$.

We adopt some properties from both above mentioned approaches to our area complexity model. Consider an action $A \mathrel{\widehat{=}} g \rightarrow x := x'.Q$. Its area complexity modeling is divided into three parts: *substitution*, *guard*, and *predicate*. Recall that the complexity of the substitution is evaluated from the write set wA as discussed in Sect. 3.1.2. To be able to calculate guard and predicate complexities, the read set is further divided into two subset: the set of *guard variables* rg that appear in the guard and the set of *predicate variables* rQ that appear in the predicate ($rA \mathrel{\widehat{=}} rg \cup rQ$). The area complexity is modeled for both sets (rg, rQ) separately.

Let us first define the area complexity for the set rQ ($rQ \neq \emptyset$): $C(rQ) \mathrel{\widehat{=}} \lceil \frac{sW}{eL} \rceil \cdot 2^{eL}$, where the eL denotes the number of elements in the set rQ and it is defined by: $eL \mathrel{\widehat{=}} |rQ|$, where $|rQ|$ denotes the cardinality of the set rQ, the sum of the widths (sW) of the variables v in the set rQ is $sW \mathrel{\widehat{=}} \sum_{v \in rQ} w_v$. In a similar manner, the complexity for the set rg can be modeled by substituting the rg to rQ in the definition above. First we discuss and compare the presented model to the size estimation methods presented above, and then we illustrate our model with an example.

The size estimation according to the input arguments has a drawback that its result increases exponentially. That is, it gives rather pessimistic results with larger number of input arguments, comparing to the size results obtained with BDD's [1]. Therefore, we adopted properties from the graph based approach presented with BDD's [9], [10]. More precisely, the first term in the complexity clause ($\lceil \frac{sW}{eL} \rceil$), divide the sum of the widths (number of input arguments) into smaller groups. These groups resemble non-terminal nodes of BDD graph and each these groups have possible output values $\{0, 1\}$ which correspond to terminal nodes. In other words, we estimate the area complexity of the predicate or guard by using the variable widths in the group as input arguments, and then multiply that area complexity with the number of the groups $\lceil \frac{sW}{eL} \rceil$, where the result of the division is rounded up. The rounding occur especially when the widths of the input variables in the set are not equal.

Example 3.1 Let us assume that the predicate Q is defined by $Q \mathrel{\widehat{=}} (x' =$

$a+b$), and furthermore, we assume that the variables a and b are of the same type and the widths of a and b are $w_a = w_b = 3$. We start by calculating the sum of the widths sW for the set rQ. That is, the sum of the widths is $sW = 3+3 = 6$, which corresponds with the number of the input arguments in the Boolean expression. The sum of the widths is divided into groups by dividing the sum of widths with the cardinality of the set: $sW/|rQ| = 3$. The graph idea behind this group division is illustrated in Fig. 3.1.3. Each of these groups has two possible output values $\{0,1\}$, and therefore, the area complexity is evaluated for each group separately, and we have: $C(rQ) = 2^2 \cdot 6/3 = 8$, where the first term represents the area complexity of single group, and the second term the number of these groups.
End of example.

The area complexity for the guarded non-deterministic assignment $A \mathrel{\widehat{=}} g \rightarrow x := x'.Q$ is defined by:

$$C(A) = C(rg) + C(rQ) + C(wA) \qquad (\textit{complexity of assignment}) \quad (1)$$

where $C(rQ)$ denotes the area complexity of the predicate Q, $C(rg)$ denotes the area complexity of the guard g, and $C(wA)$ denotes the area complexity of the substitution operation. Observe that in the non-deterministic assignment the set rg is empty ($rg = \emptyset$), and therefore the complexity of the guard is evaluated to zero. Therefore, we need to adjust the predicates area complexity model by introducing a *complexity factor* C_k ($C_k \in \mathbb{R}^+$). Thus, the area complexity of the set rQ is $C(rQ) \mathrel{\widehat{=}} bg \cdot (2^{eL})^{C_k}$, where, in general, it is assumed that $C_k = 1$. Consider the following example.

Example 3.2 Let $A \mathrel{\widehat{=}} x := x.'Q$, where the predicate is defined by $Q \mathrel{\widehat{=}} (x' = a+b)$. Assume that x, a and b are variables of the same type and the widths of a and b are w_a and w_b and the width of x is w_x. Let us further assume that the widths of the variables a and b are equal $w_a = w_b = m$ ($m \in \mathbb{R}^+$) and the width of the variable x is $w_x = m+1$. The complexity for the set rQ when the predicate performs an addition operation is discussed in Example 3.1 with read variable widths $w_a = w_b = 3$. In this example, both of the input variable have a width m, and therefore, we have: $C(rQ) = (4m)^{C_k}$, where the value of C_k is one ($C_k = 1$). The complexity of A is $C(A) = (m+1) + 4m = 5m+1$, where the $(m+1)$ is the area complexity of the substitution ($C(wA)$). Next, we assume that the predicate Q is a multiplication and the widths of the variables a and b are $w_a = w_b = m$, and the width of the variable x is $w_x = 2m$. The complexity of the action A is $C(A) = (4m)^{C_k} + 2m$. For multiplication, we can, for instance, assume that the complexity factor is two ($C_k = 2$). That is, we have, $C(A) = (4m)^2 + 2m = 16m^2 + 2m$. Notice that by adopting the complexity factor of $C_k = 2$, the result is close to school book multiplication

[12]. (That is, if the complexity of addition is m, then the complexity of the multiplication would be around m^2).
End of example.

In this paper we will assume $C_k = 1$ for addition and subtraction and $C_k = 2$ for multiplication and division. However, in general the value assigned to the factor can be any positive real number.

In general, the result of the area complexity modeling for substitution operations is the sum of the widths of the variables in the action. However, assignment statements often contains operations that are combinatorial logic at lower abstraction levels, and therefore we adopted the graph based procedure. In both cases, the result is related to the BDD's size evaluation. Although our model operates at higher abstraction level. The accuracy of the complexity model presented above was analyzed using *Ordered Binary Decision Diagrams (OBDD)* library *BuDDy* [8], which is freely down-loadable from the Internet. OBDD is a BDD, where certain reduction and ordering rules are emphasized, see [9]. The advantage of OBBD is that it is canonical (unique) for a particular functionality.

Example 3.3 We briefly analyzed the accuracy of the addition and multiplication operations using the *BuDDy* library. The OBBD was generated for addition operation described using full adders, and for basic binary multiplication using block multiplier structure. Both of these operators were analyzed using widths from 16 bits to 64 bits. The node counts of OBDD were compared with the width count of our area complexity model. The accuracy of our presentation was in both cases approximately 83%. This result is quite good for such a high abstraction level estimation, comparing to the amount of the information we have from the actions versus the fully defined Boolean functions, which are inputs for BuDDy.
End of example.

3.1.4 Complexity of systems

Area complexity ($C_{\mathcal{A}\|\mathcal{E}nv}$) defines the area of the parallel system $\mathcal{A} \parallel \mathcal{E}nv$. We form a set $S_{\mathcal{A}\|\mathcal{E}nv}$, which contains the timed actions A, and Env of the system $\mathcal{A} \parallel \mathcal{E}nv$. It is easy to see that $S_{\mathcal{A}\|\mathcal{E}nv} \mathrel{\widehat{=}} S_{\mathcal{A}} \cup S_{\mathcal{E}nv}$, where the sets $S_{\mathcal{A}}$ and $S_{\mathcal{E}nv}$ are defined by $S_{\mathcal{A}} \mathrel{\widehat{=}} \{A | A \in S_{\mathcal{A}}\}$ and $S_{\mathcal{E}nv} \mathrel{\widehat{=}} \{Env | Env \in S_{\mathcal{E}nv}\}$. The area complexity of the system is the sum of the complexities of timed actions in the set $S_{\mathcal{A}\|\mathcal{E}nv}$ and defined by:

$$C_{\mathcal{A}\|\mathcal{E}nv} \mathrel{\widehat{=}} C_{\mathcal{A}} + C_{\mathcal{E}nv} = \sum_{A \in S_{\mathcal{A}}} C(A) + \sum_{Env \in S_{\mathcal{E}nv}} C(Env) \qquad (\textit{area complexity}) \quad (2)$$

where $\sum_{A \in S_{\mathcal{A}}} C(A)$ is the area complexity of the system $\mathcal{A}$, and $\sum_{Env \in S_{Env}} C(\mathcal{E}nv)$ is the area complexity of the system $\mathcal{E}nv$. Consider the following example.

Example 3.4 Consider a timed action system $\mathcal{A}$ which reads data from its local input buffer, performs an addition operation, and stores the result to its local output buffer. The system is modeled as a *closed system*, which does not interact with its environment. the system is defined by:

sys $\mathcal{A}$ ( ) ::
|[
delay $dOp; dAdd;$
var $ibuf, obuf$: **set of** $data;$
$d_1, d_2, r : data;$
private proc $Add[\![dAdd]\!]$(**in** $x_1, x_2 : data;$ **out** $r : data) : r := r'.(r' = x_1 + x_2);$
action $Op[\![dOp]\!] : d_1, d_2 := d_1', d_2'.(d_1', d_2' \in ibuf); Add(d_1, d_2, r); obuf := obuf \cup \{r\};$
init $ibuf, obuf, d_1, d_2, r := ibuf0, obuf0, d_10, d_20, r0;$
exec **do** Op **od**
]|

where the functionality of the system is placed into procedure Add, which is called by the timed action Op, which is executed by the system $\mathcal{A}$ forever, because its guard gOp is invariantly true.

To model the area complexity for the system $\mathcal{A}$, we describe the read and write sets of timed action Op. The read set is $rOp = \{x_1, x_2\}$ and the write set is $wOp = \{r, d_1, d_2\}$. Observe that the procedure Add is included within the timed action that calls it (x_1, x_2 in the read set are the read variables of the procedure). The area complexity modeling is divided into two parts: (1) substitution from input buffer $ibuf$ to data variables d_1, d_2; substitution from result variable r to output buffer $obuf$; and (2) non-deterministic assignment (in the procedure Add). We start from the non-deterministic assignment (2), by extracting the sets rg and rQ from the read set rOp. The set rg is empty ($rg = \emptyset$), and therefore only the set rQ is considered in the area complexity modeling. We have: $rQ = \{x_1, x_2\}$. The area complexity of the set is $C(rQ) = 4m$, described more detail in Examples 3.1 and 3.2. The area complexity of the procedure is $C(Add) = 4m + (m + 1) = 5m + 1$, where the $m + 1$ is the area complexity of the substitution in the procedure. The complexity of the substitution operations (1) of the timed action Op is $C(wA) = w_{d_1} + w_{d_2} + w_r$, where the width of the variables d_1, d_2 is m, whereas the substitution of the result variable r to the output buffer has the width $m+1$. The area complexity of the timed action Op is then: $C(Op) = C(Add) + C(wA) = (5m+1) + (3m+1) = 8m + 2$.

The area complexity of the system $\mathcal{A}$ requires that the local buffers are

modeled as well. We have $C(ibuf) = |ibuf| \cdot m$, where the cardinality of the set describes the number of elements in the set, and m is the width of each element. In a similar manner we analyze the output buffer $C(obuf) = |obuf| \cdot (m + 1)$. Thus, the area complexity of the system $\mathcal{A}$ is $C_{\mathcal{A}} = C(ibuf)+C(obuf)+C(Op)$.

3.2 Power model

The power consumption is defined by: $P(system) \mathrel{\widehat{=}} P_{dyn}(system) + P_{stat}(system)$, where the first term describes the *dynamic power* consumption of the system, and the second term the *static power* consumption of the system. Dynamic power consumption is related to system operation. That is, dynamic power consumption can be reduced using different design methods such as self-timed design, see for example [16]. However, the static power consumption is caused by the leakage current I_{leak}, which is the combination of the sub threshold leakage (a weak inversion current across the device) and the gate-oxide leakage (a tunneling current through the gate oxide insulation) [15]. Detailed analysis of the leakage current can be found, for instance, in [15], [11]. In general, both the sub threshold and the oxide leakage depends on the total gate width or more approximately the gate count, temperature (sub threshold leakage), supply voltage, and oxide thickness (oxide leakage). The static power consumption can no longer be ignored, since the power consumption of chip leakage is approaching the dynamic power consumption, and the projected increases in off-state sub threshold leakage show it exceeding total dynamic power consumption as the technology drops below the 65 nm feature size [15]. Emerging techniques to moderate the gate-oxide tunneling effect could bring gate leakage under control by 2010. Let us first discuss the definition of the dynamic power consumption of an action including its energy model, and then the static power loss caused by an action.

3.2.1 Dynamic power

The energy dissipated by the timed action A every time it is executed is denoted by $E(A)(\in \mathbb{R}^+)$ and defined by:

$$E(A) \mathrel{\widehat{=}} \alpha \cdot C(A) \cdot E^1 \qquad (\textit{energy of } A) \quad (3)$$

where $C(A)$ is the area complexity of the timed action A, α is a switching activity parameter, and E^1 is the energy of an unit action A^1. The unit action is defined by: $A^1 \mathrel{\widehat{=}} x := x'.(x' = \neg y)$, where x and y are Boolean variables. The action A^1 models an inverter, whose energy E^1 in CMOS can be calculated by: $E^1 = \frac{1}{2} \cdot C_L \cdot V_{DD}^2$, where V_{DD} is the applied supply voltage, and C_L is the output capacitance of a unit size inverter driving another unit size inverter in a given technology [13]. The switching activity parameter

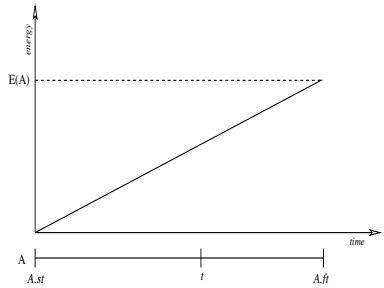


Fig. 3. Energy model for timed action A

denotes the amount of bits that change state during execution of an action. We assume that 50 % of the bits in action changes their state ($\alpha = 0.5$).

As shown in Fig. 3, the energy starts from zero when the action A starts operation at the time $A.st$ and reaches the full value E_A when A finishes operation at the time $A.ft$. This simple linear energy model indicates that the power consumption of the action A is considered constant during the action operation period $[A.st, A.ft]$, given by the slope of the energy line depicted in Fig. 3. The linear energy model also means, that the energy dissipated by A during the time period $[A.st, t]$, denoted by E_A^{*t}, where $t \in [A.st, A.ft]$, is given by : $E_A^{*t} \;\widehat{=}\; k_A^{*t} E_A$, where the fraction k_A^{*t} is defined as $k_A^{*t} \;\widehat{=}\; \frac{t - A.st}{\Delta(A)}$, where $\Delta(A) \;\widehat{=}\; A.ft - A.st$.
Similarly, the amount of energy dissipated by A during the time period $[t, A.ft]$, where $t \in [A.st, A.ft]$, is denoted by E_A^{t*} and defined by $E_A^{t*} \;\widehat{=}\; k_A^{t*} E_A$,where $k_A^{t*} \;\widehat{=}\; \frac{A.ft - t}{\Delta(A)}$. Naturally, $k_A^{*t} + k_A^{t*} = 1$ holds for the fractions k_A^{*t} and k_A^{t*}, and $E_A^{*t} + E_A^{t*} = E_A$. As an example, if the action A is interrupted (killed) prematurely by some other action at the time $t \in [A.st, A.ft]$, the amount of energy the action A dissipated before the interruption is equal to E_A^{*t}. Thus, the dynamic power consumption of an arbitrary action A during single execution is:

$$P_{dyn}(A) \;\widehat{=}\; \frac{E(A)}{\Delta(A)} \qquad (\textit{dynamic power}) \quad (4)$$

where the execution time of the action A is $\Delta(A) = A.ft - A.st$.

3.2.2 Static power consumption

Unlike dynamic power consumption, the static power consumption is not activity based. As stated above, it is a certain percent from the total power consumption of the system. To model the static power consumption of an action A, we adopt the unit action A^1. The static power consumption in CMOS for the unit actions is calculated by $P_{stat}^1 = V_{DD} \cdot I_{leak}$. Observe that, at this abstraction level, we do not have any information either from supply voltage

or from the leakage current. Therefore, the static power consumption of unit action A^1 is P^1_{stat} and an arbitrary action A causes a static power loss $P_{stat}(A)$:

$$P_{stat}(A) \mathrel{\widehat{=}} C(A) \cdot P^1_{stat} \qquad (\textit{static power}) \quad (5)$$

where the $C(A)$ is the area complexity of the action A. That is, the area information is utilized to evaluate the difference in the static power loss between actions.

3.2.3 Average power consumption

To model and analyze the power consumption of a digital system, we consider here a finite period of time $T \in \mathbb{T}$, defined by: $T \mathrel{\widehat{=}} [T.st, T.ft]$, where $T.st$, and $T.ft$ denote the start and finish times of the period, such that $(T.st, T.ft \in \mathbb{T}) \wedge (0 \leq T.st < T.ft)$. The difference between finish and start times is denoted by: $\Delta(T) \mathrel{\widehat{=}} T.ft - T.st$. During the observation period T, a timed action system $\mathcal{A}$ executes a set $S_{\mathcal{A}}(T)$ of timed actions. We have:

$$\begin{aligned} S_{\mathcal{A}}(T) \;\mathrel{\widehat{=}}\; & \{A | (A \in S_{\mathcal{A}} \\ & \wedge (\exists t : T.st < t < T.ft : gA(t))\} \end{aligned} \qquad (\textit{set of timed actions}) \quad (6)$$

where $gA(t)$ refers to the guard of an action A at a given time t. Hence, the set $S_{\mathcal{A}}(T)$ contains all the actions that are enabled at some point during the observation period T. This includes actions that are either started or finished, or both started and finished, within the observation period T, as well as those actions that are started before and finished after the observation period T. The order of events in the set $S_{\mathcal{A}}(T)$ is determined by the temporal relations of the involved actions [22].

Consider the set $S_{\mathcal{A}\parallel\mathcal{E}nv}(T)$ which contains those actions of the composite system $\mathcal{A} \parallel \mathcal{E}nv$ that are enabled and either partly or completely executed within the observation period T, based on the definition given in (6). Observe that, the set $S_{\mathcal{A}\parallel\mathcal{E}nv}$, contains all the timed actions of composite system $\mathcal{A} \parallel \mathcal{E}nv$ whereas the presented set $S_{\mathcal{A}\parallel\mathcal{E}nv}(T)$ contains only those timed actions that are enabled during the observation period T. Let us assume that actions $A \in S_{\mathcal{A}}(T)$ and $Env \in S_{\mathcal{E}nv}(T)$ are repeated n_A and n_{Env} times ($n_A, n_{Env} \geq 1$), respectively, during the observation period T. Then the average power consumption of $\mathcal{A} \parallel \mathcal{E}nv$ during T can be defined by:

$$P^{\mathcal{A}\parallel\mathcal{E}nv}_{avg}(T) \;\mathrel{\widehat{=}}\; P^{\mathcal{A}}_{avg}(T) + P^{\mathcal{E}nv}_{avg}(T) \qquad (\textit{average power } (\mathcal{A} \parallel \mathcal{E}nv))$$

where the average power consumption $P^{\mathcal{A}}_{avg}(T)$ of $\mathcal{A}$ and $P^{\mathcal{E}nv}_{avg}(T)$ of $\mathcal{E}nv$ are calculated by: $P^{\mathcal{A}}_{avg}(T) \mathrel{\widehat{=}} \frac{E_{\mathcal{A}}(T)}{\Delta(T)} + P^{\mathcal{A}}_{stat}$, and $P^{\mathcal{E}nv}_{avg}(T) \mathrel{\widehat{=}} \frac{E_{\mathcal{E}nv}(T)}{\Delta(T)} + P^{\mathcal{E}nv}_{stat}$, where $P_{stat}(\mathcal{A})$ and $P_{stat}(\mathcal{E}nv)$ are the static power consumption of the systems $\mathcal{A}$ and $\mathcal{E}nv$, respectively. The observation period is not included into static power,

because it is not dependent on the amount of enabled actions during the observation period. That is, it is certain percent of the total power consumption of the system, whenever the system is enabled. The total energies $E_{\mathcal{A}}(T)$ and $E_{\mathcal{E}nv}(T)$ of the systems $\mathcal{A}$ and $\mathcal{E}nv$ during T are defined by:

$$E_{\mathcal{A}}(T) \mathrel{\widehat{=}} \sum_{A \in S_{\mathcal{A}}(T)} \sum_{j=1}^{n_A} k_A^j E_A \qquad (\textit{energy } (\mathcal{A}))$$

$$E_{\mathcal{E}nv}(T) \mathrel{\widehat{=}} \sum_{Env \in S_{\mathcal{E}nv}(T)} \sum_{j=1}^{n_{Env}} k_{Env}^j E_{Env} \qquad (\textit{energy } (\mathcal{E}nv))$$

If $S_{\mathcal{A}}(T)$ or $S_{\mathcal{E}nv}(T)$ is empty, then the energy $E_{\mathcal{A}}(T)$ or $E_{\mathcal{E}nv}(T)$, respectively, is considered zero. In the above definitions, k_A^j and k_{Env}^j are fractions indicating the portion of an action A or Env which resides inside the observation period T in the jth execution of A or Env. Such a fraction can be generally defined for an action $X \in \{A, Env\}$ as: $k_X^j \mathrel{\widehat{=}} \frac{\Delta_T^j(X)}{\Delta(X)}$, where $\Delta_T^j(X)$ denotes the time the action X spends within the observation period T in its jth execution during T. Thus, we have that $0 < \Delta_T^j(X) \leq \Delta(X)$. If the action X is executed only once during T, i.e., $n_X = 1$, we write simply $\Delta_T(X)$ and the corresponding fraction is denoted by k_X.

Example 3.5 Consider the timed action system defined in Example 3.4. To model its average power, we define an observation period $T \mathrel{\widehat{=}} [T.st, T.ft]$, where the start time of the period is assumed to be zero ($T.st = 0$) and the finish time $T.ft$ denotes the time when the system is performed one operation cycle $T.ft = Op.ft$. Naturally, the observation period is selected by the designer, but for simplicity, we illustrate the power analysis with single operation cycle. The average power consumption for the system is defined by $P_{avg}^{\mathcal{A}}(T) = \frac{E_{\mathcal{A}}(T)}{\Delta T} + P_{stat}^{\mathcal{A}}$, where the $E(\mathcal{A})$ is the energy consumption of the system defined by: $E_{\mathcal{A}} = \sum_{Op \in S(\mathcal{A})} \alpha \cdot (E^1 \cdot (C(Op) + C(obuf) + C(ibuf)))$, where the complexities of the action Op and buffers $ibuf$ and $obuf$ are multiplied by the unit energy E^1. The area complexity modeling for system $\mathcal{A}$ is discussed more detailed in Example 3.2.

3.2.4 Temporal power modeling

Consider the set $S_{\mathcal{A}\|\mathcal{E}nv}(T)$, where the observation period T is divided into time segments T_i $(1 \leq i \leq m)$, which satisfy the following condition:

$$
\begin{aligned}
&(T_i \subseteq T) \wedge (i \geq 1) && \text{(p1)}\\
\wedge\ &(\forall t : T_i.st < t < T_i.ft : (\forall A : A \in S_{\mathcal{A}\|\mathcal{E}nv}(T) : t \notin \{A.st, A.ft\})) && \text{(p2)}\\
\wedge\ &\Big(\exists m : m \geq i :\\
&\qquad (i = 1 \Rightarrow T_i.st = T.st)\\
&\quad \wedge (1 \leq i < m \Rightarrow T_i.ft = T_{i+1}.st)\\
&\quad \wedge (\exists A : A \in S_{\mathcal{A}\|\mathcal{E}nv}(T) : (T_i.ft \in \{A.st, A.ft\})) && \text{(p3)}\\
&\quad \wedge (i = m \Rightarrow T_i.ft = T.ft)\Big)
\end{aligned}
$$

Hence, the start time of a time segment T_i is either the start time $T.st$ of the observation period T or the start or finish time of an action $A \in S_{\mathcal{A}\|\mathcal{E}nv}(T)$ *(p1,p3)*. Analogously, the finish time of a time segment T_i is either the finish time $T.ft$ of the whole observation period T or the start or finish time of an action $A \in S_{\mathcal{A}\|\mathcal{E}nv}(T)$. Furthermore, the finish time of a segment T_i is the start time of the next segment T_{i+1} for $1 \leq i < m$ and $m > 1$. No action $A \in S_{\mathcal{A}\|\mathcal{E}nv}(T)$ is started or finished *inside* a time segment T_i, i.e. when $T_i.st < t < T_i.ft$ *(p2)*.

From the definition of the time segment T_i follows that the set $S_{\mathcal{A}\|\mathcal{E}nv}(T)$ of actions executed during the observation period T is the union of the subsets $S_{\mathcal{A}\|\mathcal{E}nv}(T_i)$ for $1 \leq i \leq m$: $S_{\mathcal{A}\|\mathcal{E}nv}(T) = \bigcup_{i=1}^{m} S_{\mathcal{A}\|\mathcal{E}nv}(T_i)$. Moreover, with the whole observation period T, we have for the time segments T_i ($\subseteq T$) that $S_{\mathcal{A}\|\mathcal{E}nv}(T_i) = S_{\mathcal{A}}(T_i) \cup S_{\mathcal{E}nv}(T_i)$. Notice that each timed action can be executed only once within the time segment due to the definition of T_i.

The temporal power consumption during a time segment T_i, denoted by $P_{tmp}^{\mathcal{A}\|\mathcal{E}nv}(T_i)$, is then defined as:

$$
P_{tmp}^{\mathcal{A}\|\mathcal{E}nv}(T_i) \ \widehat{=}\ P_{tmp}^{\mathcal{A}}(T_i) + P_{tmp}^{\mathcal{E}nv}(T_i) \qquad \textit{(temporal power)}
$$

The subsystem-specific power consumptions $P_{tmp}^{\mathcal{A}}(T_i)$ and $P_{tmp}^{\mathcal{E}nv}(T_i)$ are given by: $P_{tmp}^{\mathcal{A}}(T_i) \ \widehat{=}\ \frac{E_{\mathcal{A}}(T_i)}{\Delta(T_i)} + P_{stat}^{\mathcal{A}}$ and $P_{tmp}^{\mathcal{E}nv}(T_i) \ \widehat{=}\ \frac{E_{\mathcal{E}nv}(T_i)}{\Delta(T_i)} + P_{stat}^{\mathcal{E}nv}$ where $\Delta(T_i)$ denotes the time segment ($\Delta(T_i) \ \widehat{=}\ T_i.ft - T_i.st$), and $E_{\mathcal{A}}(T_i)$ and $E_{\mathcal{E}nv}(T_i)$ are the total energies of the actions in the sets $S_{\mathcal{A}}(T_i)$ and $S_{\mathcal{E}nv}(T_i)$, respectively. The static power consumption of the systems $\mathcal{A}$ and $\mathcal{E}nv$ are $P_{stat}^{\mathcal{A}}$ and $P_{stat}^{\mathcal{E}nv}$, respectively. If the set $S_{\mathcal{A}}(T_i)$ or $S_{\mathcal{E}nv}(T_i)$ is empty, then the energy $E_{\mathcal{A}}(T_i)$ or $E_{\mathcal{E}nv}(T_i)$, respectively, is considered zero. Otherwise we have that:

$$
\begin{aligned}
&E_{\mathcal{A}}(T_i) \mathrel{\widehat{=}} \textstyle\sum_{A \in S_{\mathcal{A}}(T_i)} k_A(T_i) E_A && \textit{(energy } (\mathcal{A}))\\
&E_{\mathcal{E}nv}(T_i) \mathrel{\widehat{=}} \textstyle\sum_{Env \in S_{\mathcal{E}nv}(T_i)} k_{Env}(T_i) E_{Env} && \textit{(energy } (\mathcal{E}nv))
\end{aligned}
$$

where $k_A(T_i)$ and $k_{Env}(T_i)$ are fractions indicating the portion of an action A or Env which occurs inside the time segment T_i.

4 Modeling and Constraining the Behavior of the Systems

4.1 Constraints

A constraint is an expression, a Boolean condition B, according to which the timed actions are obliged to operate. In a hardware systems, the violation of the constraints indicates a useless, unpredictable computation, and therefore it is modeled either as an assert statement or as an assumption statement. When a constraint is modeled as a assert statement, it behaves as a *skip* statement if it holds ($B \equiv true$), but otherwise it behaves as an *abort* statement ($B \equiv false$). 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 does not establish any post condition causing an abnormal termination of the system. This model reflects the definition of a *hard constraint* whose adherence is mandatory and a violation terminates the program. In a text, we denote a constraint by $\{B\}$, where B defines the Boolean condition according to which the involved timed actions are obligate to operate. We start by shortly reviewing the time related constraints, defined in [23], which are divided into *hard and relative constraints*. Relative constraints define the order in which timed actions are allowed to be executed. Then, we introduce new performance related constraints.

4.1.1 Deadline and relative constraints

A deadline defines the *maximum time* that a timed action is allowed to consume in its operation. Applying the time constraint definitions given above, we define [23]: $\{A, d\} \mathrel{\widehat{=}} \{\Delta(A) \leq d\}$, where A defines those timed actions and their relations whose operation time is bounded and $\Delta(A)$ is a time expression that evaluates to a time value ($\mathbb{T}$). Time expressions are composed of mathematical operations, e.g. $+$ and $-$.

Relative constraints use relative timing of operations to define the order in which the operations much be executed with respect to each other in the time domain. The relative constraints are defined using relative orderings defined below. With the can be directly used in constraints to restrict the temporal behavior of timed actions. The relative constraints are, for example:

A **starts before** $B \mathrel{\widehat{=}} A.st < B.st$	(*starts before*)	(7)
A **ends after** $B \mathrel{\widehat{=}} B.ft < A.ft$	(*ends after*)	(8)
A **precedes** $B \mathrel{\widehat{=}} A.ft < B.st$	(*precedes*)	(9)
A **meets** $B \mathrel{\widehat{=}} A.ft = B.st$	(*meets*)	(10)
A **starts with** $B \mathrel{\widehat{=}} A.st = B.st$	(*starts with*)	(11)
A **ends with** $B \mathrel{\widehat{=}} A.ft = B.ft$	(*ends with*)	(12)

where the $A_i.st$ and $A_i.ft$ are the start and finish times of a timed action, respectively.

4.1.2 Performance related constraints

Let us next introduce performance related constraints: *area*, *average power* and *temporal power*, where the two latter ones are defined as a hard constraint and the former using the assumption based constraint. The assumption based constraint, denoted by $[B]$, where B defines the Boolean condition, do not terminate the system. That is, it gives the designer information about the system, which can be used in the design process. The area constraint defines the maximum area complexity for a timed action system. For the following system $\mathcal{A} \parallel \mathcal{E}nv$, we have: $[\mathcal{A} \parallel \mathcal{E}nv, l] \mathrel{\widehat{=}} [C_{\mathcal{A}\parallel\mathcal{E}nv} \leq l]$, where l defines the maximum area complexity allowed for the system.

The average power constraint defines the maximum average power that the system is allowed to consume during the observation period T. The power related constraint is evaluated after timing related constraints and the area constraint is verified. That is, we need both area and time information to validate the power consumption. We have: $\{\mathcal{A} \parallel \mathcal{E}nv, p_{avg}\} \mathrel{\widehat{=}} \{P_{avg}^{\mathcal{A}\parallel\mathcal{E}nv}(T) \leq p_{avg}\}$ where the p_{avg} is the limit value for average power during the observation period T. Observe that, functionality inside the observation period in not altered during refinement steps. This restrictions follows from the deadline constraint, we cannot exceed the observation period if the deadline constraint holds for all timed actions in the system.

The temporal power constraint is defines the maximum peak power that the system is not allowed to exceed. This constraint is evaluated after timing and area related constraints are validated. We have: $\{\mathcal{A} \parallel \mathcal{E}nv, p_{peak}\} \mathrel{\widehat{=}} \{\forall i.P_{tmp}^{\mathcal{A}\parallel\mathcal{E}nv}(T_i) \leq p_{peak}\}$, where the p_{peak} is the limit value for temporal power for each time segment T_i in the observation period T. That is, we set a peak power value p_{peak} that is not allowed to exceed in any segment T_i ($\in T$). The power constraints are defined as hard constraint, because exceeding the assigned power limit can impose serious problems to a system.

5 Refinement of Systems

Action Systems are intended to be developed in a stepwise manner within the *refinement calculus* framework [3]. The refinement calculus guarantees the correctness of each performed transformation step. In this section, we concentrate on applying the refinement calculus framework for power modeling. We start by describing the basic concepts of the refinement calculus and derivation of conventional Action Systems.

An (atomic) action A is said to be (correctly) *refined* by action C, denoted $A \leq C$, if $\forall Q.(wp(A,Q) \Rightarrow wp(C,Q))$ holds. This is equivalent to the condition

$\forall P, Q((P\ A\ Q) \Rightarrow (P\ C\ Q))$ (*total correctness property*)

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

5.1 Data refinement of conventional actions

Let us assume an (atomic) action A on the variables a and u is refined by 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. Then the abstract action A is *data-refined* by the concrete action C, denoted by $A \leq_R C$, if the following condition holds.

$\forall Q(R \wedge wp(A,Q)) \Rightarrow wp(C, \exists a.R \wedge Q)$ (*condition of the data refinement*)

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

$R \wedge gC \Rightarrow gA$ (*body*)
$\forall Q.(R \wedge gC \wedge wp(sA,Q) \Rightarrow wp(sC, \exists a.R \wedge Q))$ (*guard*)

The data refinement $A \leq_R C$ replaces a with c preserving the variables u.

5.2 Refinement of Timed Action Systems

Rather than going into the details of refinement of parallel and reactive systems [4], we review here a commonly used method to prove the refinement step. The presented data refinement method has been used to prove a correctness of superposition refinement of action systems [6] as well as the trace refinement of action systems. In the trace refinement of an action system the trace is preserved, or the sequence of global states (observable behavior), of

the system in question. A fundamental study of the trace refinement can be found in [7]. In the superposition refinement the behavior of a system model is enhanced by adding new functionality into the model while preserving the old one. The method is also extended to prove the correctness of data refinement of action systems with remote procedures in [20].

Below we describe how to establish the refinement of action systems. We will first describe the refinement formally and thereafter give an informal description of the required conditions.

5.2.1 Refinement Rule

Consider action systems operating in an arbitrary environment $\mathcal{E}nv$ of form (simplified model): $\mathcal{E}nv(u) :: |[\textbf{action } E;\ \textbf{init } u := u0;\ \textbf{exec do } E \textbf{ od}]|$:

```
sys  A  (  u; ) ::
|[
    constraint  {A};
    delay  dA;
    var  a;
    action  A[[dA]] : (aA);
    init  g, a : g0, a0;
    exec  do A od
]|
```

```
sys  C  (  u; ) ::
|[
    constraint  {C};
    delay  dC; dX;
    var  c;
    action  C[[dC]] : (cC);
            X[[dX]] : (cX);
    init  g, c : g0, c0;
    exec  do C [] X od
]|
```

where the constraints of a timed action system, say $\mathcal{A}$, are denoted by: $\{\mathcal{A}\} = cnst_1 \wedge \ldots \wedge cnst_n$, where $cnst_i$, $1 \leq i \leq n$, are the constraints of the timed action system $\mathcal{A}$.

An abstract action A on the variables a (local variables) and u (global variables) is refined by the 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 abstract system $\mathcal{A}$ is correctly refined by the concrete system $\mathcal{C}$, denoted $\mathcal{A} \sqsubseteq \mathcal{C}$, if there exists an *invariant* $R(a, c, u)$ (an abstraction relation) on state variables, if the following conditions are satisfied:

$R(a_0, c_0, u_0) = true$	(*Initialization*)	(i)
$A \leq_R C$	(*Main action*)	(ii)
$skip \leq_R X$	(*Auxiliary action*)	(iii)
$R \wedge gA \Rightarrow gC \vee gX$	(*Continuation condition*)	(iv)
$R \Rightarrow \textbf{wp}(\textbf{do } X \textbf{ od}, true)$	(*Internal convergence*)	(v)
$R \wedge \textbf{wp}(E, true) \Rightarrow \textbf{wp}(E, R)$	(*Non-interference*)	(vi)
$R \wedge \{\mathcal{C}\} \Rightarrow \{\mathcal{A}\}$	(*Timed behavior*)	(vii)
$R \wedge [\mathcal{C}] \vee R \wedge \{\mathcal{C}\} \Rightarrow [\mathcal{A}] \vee \{\mathcal{A}\}$	(*Performance related*)	(viii)

(i) The first condition says that the initialization of the systems $\mathcal{A}$ and $\mathcal{C}$ establish the abstraction relation R.

(ii) The second condition requires the abstract action A to be data-refined by the concrete action C using R.

(iii) The third condition, 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.

(iv) The fourth condition requires that whenever the action A of the abstract system $\mathcal{A}$ is enabled, assuming the abstraction relation R holds, there must be a enabled action in the concrete system $\mathcal{C}$ as well.

(v) The fifth condition states that if R holds, the execution of the auxiliary action X, taken separately, must terminate at some point.

(vi) The sixth condition guarantees that the interleaved execution of E actions preservers the abstraction relation R

(vii) The seventh condition requires that all the time constraints are met in the concrete timed action system $\mathcal{C}$.

(viii) The eight condition requires that the area constraint is verified in the concrete timed action system $\mathcal{C}$ and the power constraints are met in the concrete timed action system $\mathcal{C}$.

The conditions (i)-(v) guarantee, in terms of global and local variables, that the behavior of an abstract system is preserved in the concrete one. That is, the action system is executed in isolation, as a closed system. Therefore, the five first requirements and the seventh (vii) are sufficient in proving the trace refinement of the timed action system. However, when the system is operating in a parallel composition with other action systems the non-interference condition (vi) must be validated, too.

The functionality of the timed action is refined using data refinement of conventional actions. The correctness of the data refinement of timed actions is proved by the condition $A \leq_R C \Rightarrow A[\![dP]\!] \leq_R C[\![dP]\!]$. The proof of this condition is shown in [23]. The performance related constraints (viii) are evaluated after the time related constraint are verified. The area constraint is evaluated first because it provides necessary information to the verification of the power constraint. Moreover, the power constraint is evaluated after the defined observation period is completed.

5.3 Refinement of the constraints

We illustrate the refinement of constraints through several examples.

Example 5.1 The operation of the timed action system $\mathcal{A}$ presented in Example 3.4 was not restricted by any constraints. We start by defining time when the result of the computation of the timed action Op must be written onto the output buffer. The deadline is of form $(\Delta(Op[Add]) \leq D)$, where $Op[Add]$ denotes the fact that Op calls a procedure in its body and D is the given deadline $(D \in \mathbb{T})$.

In Example 3.5, the average power of the system $\mathcal{A}$ was modeled during observation period T: $T = [0, Op.ft]$. Before we can constrain the average power of the system, we set the area constraint: $(C_{\mathcal{A}} \leq l)$, where l is the maximum allowed area complexity for the system. Notice that the value of l remains through refinement steps performed, whereas the system complexity increased/decreases. The value of l is set by the designer, and in this example, we assume that it is twice the area complexity of the original system $l = 2 \cdot C_{\mathcal{A}(orig)}$. By adopting the limit value l in area constrain, we constraint the average power consumption of the system $\mathcal{A}$ by: $P^{\mathcal{A}}_{avg}(T) \leq p_{avg}$, where the p_{avg} is the average power limit constraining the system and it is defined by $p_{avg} = \frac{E^1 \cdot l}{\Delta(T)} + P^1_{stat} \cdot l$, where l is the limit value of the area constraint and $\Delta(T)$ is the difference between start and finish times of observation period T and P^1_{stat} is the static power consumption of an unit action.
End of example.

Example 5.2 Consider the timed action system $\mathcal{A}$, described in Example 3.4. The timed action Op reads data from the input buffer, performs the computation and writes the result onto the output buffer. To separate the read and write operations, the timed action $Op : [\![dOp]\!] : d_1, d_2 := d'_1, d'_2.(d'_1, d'_2 \in ibuf); Add(d_1, d_2, r); obuf := obuf \cup \{r\};$ is divided into two timed actions Ro(read operate) and W (write): $Ro[\![dRo]\!] : \neg b \rightarrow d_1, d_2 := d'_1, d'_2.(d'_1, d'_2 \in ibuf); Add(d_1, d_2, r); b, s := T, r;$ and $W[\![dW]\!] : b \rightarrow obuf, b := obuf \cup \{s\}, F;$, whose delays are $\Delta(Ro)$ and $\Delta(W)$ defined such that $\Delta(Ro)+\Delta(W) = \Delta(Op)$. Notice that only the delay $\Delta(Op)$ of timed action Op is reallocated between the new timed actions. The delay of the procedure is not altered, it is included into the delay of the timed action that calls it. The variable b $(b \in \{T, F\})$ sequences the operation between the new timed actions. After the refinement we have a timed action system $\mathcal{A}'$ whose behavior is given by: **exec do** $Ro \,[\!]\, W$ **od**. Showing the correctness of this refinement we need to prove that the conditions (i)-(viii) of timed refinement are satisfied. We have:

(i) *Initialization.* The initializations of the timed action systems $\mathcal{A}$ and $\mathcal{A}'$ do not contradict.

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

	Body:
	$I \wedge gR \wedge wp(sOp, Q) \Rightarrow wp(sR, I \wedge Q)$
	$\Leftrightarrow$ {weakest precondition of sA and sCp } $I \wedge b \wedge$
	$Q \Rightarrow I[F/b] \wedge Q[r/s]$
Guard:	$\Leftrightarrow$ {the invariant $I \mathrel{\widehat{=}} b \Rightarrow r = s$}
$I \wedge gR \Rightarrow gW$	$(b \Rightarrow r = s) \wedge b \wedge Q \Rightarrow (b \Rightarrow r = s)[F/b] \wedge Q[r/s]$
$\Leftrightarrow I \wedge b \Rightarrow T$	$\Leftrightarrow$ { logic }
$\Leftrightarrow T$	$(b \Rightarrow r = s) \wedge b \wedge Q \Rightarrow T \wedge Q[r/s]$
	$\Leftrightarrow$ {logic }
	$b \wedge (r = s) \wedge Q \Rightarrow Q[r/s]$
	$\Leftarrow b \wedge Q \Rightarrow Q[r/s] \wedge r = s$
	$\Leftrightarrow b \wedge Q \Rightarrow Q$
	$\Leftrightarrow T$

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

(iv) *Continuation condition.* There is always either of the new timed actions Ro or W enabled when the original timed action Op is enabled.

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

(vi) *Non-interference.* Holds trivially as the new system is a closed action system.

(vii) *Timing behavior.* The tenability of the constraint $(\Delta(Op[Add]) \leq D)$, defined in Example 5.1, must be validated to show correctness of this requirement. In verification, we use a computation path [23], which defines a path through the system. The computation path in the new system, based on the functional behavior, the following: $CP(Ro \mapsto W)$, where the timed action Ro is performed before the execution of W. The computation path delay is $\Delta({}^*Ro \mapsto W^*) = \Delta(Ro) + \Delta(W)$, where * shows by its position, is the delay of the timed action included in the calculation or not: for the former action we have: ${}^*Ro \mathrel{\widehat{=}}$ "*Ro included*", $Ro^* \mathrel{\widehat{=}}$ "*Ro excluded*"; and for the latter action the other way around:${}^*W \mathrel{\widehat{=}}$ "*W excluded*", $W^* \mathrel{\widehat{=}}$ "*W included*". By calculating the computation path delay and comparing the result to the delay of the original timed action Op, we are able to show that the timing requirement is satisfied. We have: $\Delta({}^*Ro \mapsto W^*) = \Delta(Ro[Add] + W) = \Delta(Ro[Add]) + \Delta(W) = \Delta(Ro) + \Delta(Add) + \Delta(W)$. Based on the requirement: $\Delta(Ro) + \Delta(W) = \Delta(Op)$

and the fact that $\Delta(Op[Add]) = \Delta(Op) + \Delta(Add)$ we may conclude that the deadline is satisfied, and thus the requirement is satisfied as well.

(viii) *Performance related.* The area complexity constraint, described in Example 5.1, is validated by defining the area complexity of the system $\mathcal{A}'$: $C_{\mathcal{A}'} = C(Ro) + C(W) + C(ibuf) + C(obuf)$, where the area complexity of the input and output buffers is defined in Example 3.4. That is, the area complexity of the buffers $(ibuf, obuf)$ is not altered during refinement. To evaluate the effect of the refinement step it is enough to compare the area complexity of the timed action Op from the system $\mathcal{A}$ with the complexities of the timed actions Ro and W from the system $\mathcal{A}'$. We have: $C(Op) = 8m + 2$, whereas $C(Ro) = 8m + 4$ and $C(W) = m + 3$. We conclude that, the area complexity of the system is increased by $m + 5$ nodes during the refinement step $\mathcal{A} \sqsubseteq \mathcal{A}'$. However, the area constraint set for the system, in the Example 5.1 is satisfied, because the area complexity of the new system is less than twice the area complexity of the old system $\mathcal{A}$. Thus, the area constraint is updated by: $(C_{\mathcal{A}'} \leq l)$.

The average power constraint verification is divided into two phases. First, we verify the observation period from the power equation. The deadline of the action Op is satisfied by the new actions Ro and W as stated in (vii), and, furthermore, the delay of the timed action Op is reallocated between the two new timed actions Ro and W, we can adopt the same observation period T as defined in Example 5.1. In other words, the denominator of the dynamic power clause does not change. Second, we revise the area constraint, which has effect on both dynamic and static power consumption. The new system $\mathcal{A}'$ satisfies the area constraint, and therefore, we may conclude that the average power constraint is satisfied as well. We have: $(P^{\mathcal{A}'}_{avg} \leq p_{avg})$.

Thus, we have performed the refinement $\mathcal{A} \sqsubseteq \mathcal{A}'$.

sys $\mathcal{A}'$ ( ) ::
|[
delay $dRo, dW, dAdd$;
constraint $cnst_D : (\Delta^* Ro \mapsto W^*) \leq D$;
$cnst_A : (C_{\mathcal{A}'} \leq l)$;
$cnst_P : (P^{\mathcal{A}'}_{avg} \leq p_{avg})$;
var $ibuf, obuf$: **set of** $data$;
$d_1, d_2, r : data$;
private proc $Add[\![dAdd]\!]$(**in** $x_1, x_2 : data$; **out** $r : data$) : $r := r'.(r' = x_1 + x_2)$;
action $Ro[\![dRo]\!] : \neg b \rightarrow d_1, d_2 := d'_1, d'_2.(d'_1, d'_2 \in ibuf); Add(d_1, d_2, r); b, s := T, r$;
$W[\![dW]\!] : b \rightarrow obuf, b := obuf \cup \{s\}, F$;
init $b, d_1, d_2, r, ibuf, obuf := F, d_10, d_20, r_0, ibuf0, obuf0$;
exec **do** Ro [] W **od**
]|

To have detailed information on power consumption of the two new timed actions, we set the temporal power constraint for the system $\mathcal{A}'$. The order of events in the observation period T is defined using relative constraints, we have: Ro **meets** W. According to this, we divide the observation period into two time segments, and constrain the peak power consumption for both segments (T_1, T_2). We have: $(P^{\mathcal{A}'}_{tmp}(T_1) \leq p_{peak})$ for time segment T_1, and $(P^{\mathcal{A}'}_{tmp}(T_2) \leq p_{peak})$ for time segment T_2. The time segments are defined by $T_1 = [Ro.st, Ro.ft]$ and $T_2 = [W.st, W.ft]$. The peak power limit is a power limit set by the designer, and therefore in this example we assume that it is twice the average power constraint $(p_{peak} = 2 \cdot p_{avg})$.

Example 5.3 The communication for the system $\mathcal{A}'$ is implemented using procedure based communication. That is, we include a sender and a receiver actions to implement the communication between the system $\mathcal{A}'$ and its environment $\mathcal{E}nv$. The behavior of the environment is given as (simplified model): **exec do** Snd [] E [] Rec **od**, where the timed actions Snd and Rec implements the procedure based communication with the system $\mathcal{A}''$ and E describes timed actions in the environment. The environment interacts with the system $\mathcal{A}''$ only through the new interface procedures. That is, it does not have access to the inner operation of the system $\mathcal{A}''$. To accomplish this behavior, we introduce a receiver action: $Rec[\![dRec]\!] : \neg id \rightarrow$ **await** $n; id := T$, where n is the communication procedure of form: $n[\![dn]\!] : (\textbf{in}\ y : data) : ibuf := ibuf \cup y$. Furthermore, we introduce a sender action: $Snd[\![dSnd]\!] : od \rightarrow$ **call** $p(obuf); od := F$, where the communication procedure p is defined by the environment. The existing timed actions Ro and W are slightly updated, and we have timed actions Ro_2 and W_2: $Ro_2[\![dRo_2]\!] : \neg b \wedge id \rightarrow d_1, d_2 := d'_1, d'_2.(d'_1, d'_2 \in ibuf); Add(d_1, d_2, r); b, s :=$

T, r and $W_2[\![dW_2]\!] : b \rightarrow obuf, b, id, od := obuf \cup \{s\}, F, F, T$. The delays for these four actions Snd, Ro_2, W_2 and Rec are defined such that $\Delta(Snd) + \Delta(Ro_2) + \Delta(W_2) + \Delta(Rec) = \Delta(Ro) + \Delta(W)$. After refinement, we have a parallel system composition $\mathcal{A}'' \| \mathcal{E}nv$, where the behavior of the system $\mathcal{A}''$ is given: **exec do** $Snd \;[\!]\; Ro_2 \;[\!]\; W_2 \;[\!]\; Rec$ **od**. Showing the correctness of this refinement we need to prove that the conditions (i)-(viii) of timed refinement are satisfied. We have:

(i) *Initialization.* The initializations of the timed action systems $\mathcal{A}'$ and $\mathcal{A}''$ do not contradict.

(ii) *Main action.* Our goal is to prove that $Ro \leq_I Snd$, where I is an invariant of form $I \mathrel{\widehat{=}} n(y) \Rightarrow r = s$. We have ($n(y)$ is n in the proof):

Guard:
$I \wedge gSnd \Rightarrow n$
$\Leftrightarrow I \wedge od \Rightarrow T$
$\Leftrightarrow T$

Body:
$I \wedge gSnd \wedge wp(sn, Q) \Rightarrow wp(sSnd, I \wedge Q)$
$\Leftrightarrow$ {weakest precondition of sn and sSnd } $I \wedge n \wedge$
$Q \Rightarrow I[F/n] \wedge Q[r/s]$
$\Leftrightarrow$ {the invariant $I \mathrel{\widehat{=}} n \Rightarrow r = s$}
$(n \Rightarrow r = s) \wedge n \wedge Q \Rightarrow (n \Rightarrow r = s)[F/n] \wedge Q[r/s]$
$\Leftrightarrow$ { logic }
$(n \Rightarrow r = s) \wedge n \wedge Q \Rightarrow T \wedge Q[r/s]$
$\Leftrightarrow$ {logic }
$n \wedge (r = s) \wedge Q \Rightarrow Q[r/s]$
$\Leftarrow n \wedge Q \Rightarrow Q[r/s] \wedge r = s$
$\Leftrightarrow n \wedge Q \Rightarrow Q$
$\Leftrightarrow T$

(iii) *Auxiliary action.* Because the auxiliary actions Rec and Ro_2 do not modify any interface variables, they behave like a skip with respect to this kind of variables.

(iv) *Continuation condition.* There is always either of the timed actions Rec, Ro_2, W_2 or Snd enabled when the original timed action Op is enabled.

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

(vi) *Non-interference.* Holds because the environment $\mathcal{E}nv$ interacts with the system only through the new interface procedures and has no effect on its inner operation.

(vii) *Timing behavior.* From the timing point of view, we have to show that the constraint $cnst_D$ is satisfied by the new system. This can be proved by showing that the duration that it takes for data item to traverse through the new system does not violate the constraint. Therefore, we

need to determine the computation path for the sent items. The computation path is: $(^*Rec \mapsto Snd^*) = Rec; n; Ro_2; W_2; Snd$ and its delay is $(\Delta(^*Rec \mapsto Snd^*)) = \Delta(Rec) + \Delta(Ro_2) + \Delta(W_2) + \Delta(Snd)$. Thus, we may conclude that the new system adheres the constraint of the original timed action system, because we required in the refinement that $\Delta(Rec) + \Delta(Ro_2) + \Delta(W_2) + \Delta(Snd) = \Delta(^*Ro \mapsto W^*)$.

(viii) *Performance related.* The area complexity constraint is validated by defining the area complexity of the system $\mathcal{A}''$: $C_{\mathcal{A}''} = C(Snd)+S(Rec)+C(Ro_2)+C(W_2)+C(ibuf)+C(obuf)$. The effect of the refinement step $\mathcal{A}' \subseteq \mathcal{A}''$ is $m+6$ to the area complexity and all together the area complexity is increased by the amount of $m+11$ during these two refinement steps ($\mathcal{A} \subseteq \mathcal{A}' \subseteq \mathcal{A}''$). Thus, we may conclude that the area constraint is satisfied and we have: $(C_{\mathcal{A}''} \leq l)$.

The average power constraint is verified in a similar manner as in Example 5.2. The verification of the time constraint in (vii) stated that the new system adheres the deadline constraint from the timed action system $\mathcal{A}'$. Therefore, we may conclude that the observation period T is not altered. Furthermore, the area constraint was satisfied for the system $\mathcal{A}''$, and therefore, we may conclude that the average power constraint is satisfied as well, we update the power constraint: $(P_{avg}^{\mathcal{A}''} \leq p_{avg})$.

To verify the temporal power constraints, we define the order of events during the observation period T using temporal relations, and we have the following execution sequence: Rec; Ro_2; W_2; Snd. That is, after the Rec action is finished, the read operate Ro_2 is enabled and so on. As a result, we have four time segments each of which contains the operation of a one timed action according to the execution sequence presented above (the first time segment contains the operation of the timed action Snd, the second contains the operation of the timed action Ro_2 and so on.). The time segments are defined by: $T_1 = [Snd.st, Snd.ft]$, $T_2 = [Ro_2.st, Ro_2.ft]$, $T_3 = [W_2.st, W_2.ft]$, and $T_4 = [Rec.st, Rec.ft]$. Furthermore, we have that $\Delta T_1 + \Delta T_2 + \Delta T_3 + \Delta T_4 = \Delta(^*Rec \mapsto Snd^*)$, where $\Delta(^*Rec \mapsto Snd^*)$ is the computation path delay of a one operation cycle. To verify the temporal power we have to show that the limit value p_{peak} is not exceeded in any of these segments.

The area complexities for the timed actions Ro_2 and W_2 are $C(Ro_2) = 8m+5$ and $C(W_2) = m+5$, respectively. The communication actions Snd and Rec are evaluated as follows: $C(Snd) = 2m+2$, which includes the area complexity of the communication procedure n. The area complexity of the receiver action Rec is $C(Rec) = m+2$, where the procedure call is modeled as a substitution operation. Observe that the area complexity of

exported procedure p would be modeled in the environment, but since the environment is excluded we can leave it out from this discussion. We can see that the area complexities of the communication actions are significantly smaller than the actions Ro_2 and W_2. Furthermore, by comparing the area complexities of timed actions Ro and W, defined in Example 5.2, we see that there is no significant increase. Therefore, the effect of area complexity is small to the temporal power. The delays of the time segments are smaller as they were in Example 5.2, which increases the temporal power consumption. However, we may conclude that the peak power consumption limit p_{peak} is not exceeded (in other words, the power consumption in each time segment is smaller than four times the average power limit p_{avg}. This is due to the moderate increase in area, and due to the sequential operation of the actions (see the above definitions of the actions). That is, even though the duration of time segments are smaller, the operation of the action is divided equally for the whole time segment. Thus, we may conclude that the temporal power constraint is satisfied for the time segments T_1, T_2, T_3 and T_4. We have: $(P^{\mathcal{A}''}_{tmp}(T_1) \leq p_{peak})$, $(P^{\mathcal{A}''}_{tmp}(T_2) \leq p_{peak})$, $(P^{\mathcal{A}''}_{tmp}(T_3) \leq p_{peak})$, and $(P^{\mathcal{A}''}_{tmp}(T_4) \leq p_{peak})$

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

sys $\mathcal{A}''$ (**imp** p; **exp** n;) ::

|[

delay $dRec, dSnd, dn, dRo_2, dW_2, dAdd$;

constraint $cnst_{Op} : (\Delta^* Snd \mapsto Rec^*) \leq D$;
$cnst_A : (C_{\mathcal{A}''} \leq l)$;
$cnst_P : (P^{\mathcal{A}''}_{avg} \leq p_{avg})$;
$cnst_P : (P^{\mathcal{A}''}_{tmp}(T_1) \leq p_{peak})$;
$cnst_P : (P^{\mathcal{A}''}_{tmp}(T_2) \leq p_{peak})$;
$cnst_P : (P^{\mathcal{A}''}_{tmp}(T_3) \leq p_{peak})$;
$cnst_P : (P^{\mathcal{A}''}_{tmp}(T_4) \leq p_{peak})$;

var $ibuf, obuf$: **set of** $data$;
$b, id, od : bool; d_1, d_2, r, s; data$;

public proc $n : [\![dn]\!] : (\textbf{val}\ y : data) : ibuf := ibuf \cup y$;

private proc $Add[\![dAdd]\!](\textbf{in}\ x_1, x_2 : data; \textbf{out}\ r : data) : r := r'.(r' = x_1 + x_2)$;

action $Rec[\![dRec]\!] : \neg id \rightarrow \textbf{await}\ n; id := T$;
$Ro_2[\![dRo_2]\!] : \neg b \wedge id \rightarrow d_1, d_2 := d_1', d_2'.(d_1', d_2' \in ibuf); Add(d_1, d_2, r); b, s := T, r$;
$W_2[\![dW_2]\!] : b \rightarrow obuf, b, id, od := obuf \cup \{s\}, F, F, T$;
$Snd[\![dSnd]\!] : od \rightarrow \textbf{call}\ p(obuf); od := F$;

init $id, od, b, s, ibuf, obuf, d_1, d_2 := T, F, F, d0, ibuf0, obuf0, d_1 0, d_2 0$;

exec **do** Rec [] Ro_2 [] W_2 [] Snd **od**

]|

6 Conclusion

In this paper, we have proposed a method to develop, in a stepwise manner, an abstract system towards more concrete one. One of the advantages of the defined refinement rules is that it is a clear extension to existing refinement rules of both conventional Actions Systems and Timed Actions Systems, and therefore easily adopted. We illustrated using simple examples, how the area and power related refinement rules are step wisely validated into a form where the operation is sequenced by a new auxiliary Boolean variable.

In this paper, we did not considered the scalability of our power modeling approach, as the scope was to introduce this property into the time domain. However, the introduced power modeling approach and the refinement rules is an important step towards that direction, as we are now able to start the 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, and furthermore, the power related issues.

References

[1] H. R. Andersen and H. Hulgaard. *Boolean Expression Diagrams*, in Proc. IEEE Logic in Computer Science, Poland, 1997.

[2] S. B. Akers. *Binary Decision Diagrams*, IEEE Transaction on Computers, Vol C-27, No.6, August 1978, pp. 509-516.

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

[4] R. J. Back. *Refinement Calculus, part II: Parallel and Reactive Programs*, in Proc. on Stepwise refinement of distributed systems: models, formalisms, correctness, vol. 430, 1990, pp. 67-93.

[5] R. J. R. Back and K. Sere. *From Modular Systems to Action Systems*, in Proc. of Formal Methods Europe' 94, Spain, October 1994. Lecture notes on computer science, Springer-Verlag.

[6] R. J. Back and K. Sere. *Superposition Refinement of Reactive Systems*, in Formal Aspects of Computing, vol. 8, no. 3, 1996, pp. 324-346.

[7] R. J. Back and J. von Wright. *Trace Refinement of Action Systems*, in International Conference of Concurrent Theory, 1994, pp. 367-384.

[8] BuDDy BDD package by J. Nielsen. http://sourceforge.net/projects/buddy/.

[9] R. E. Bryant. *Graph.Based Algorithms for Boolean Function Manipulation*, IEEE Transaction on Computer, Vol C-35, No. 8, August 1986, pp. 677-691.

[10] R. E. Bryant. *On the Complexity of VLSI Implementations and Graph Presentations of Boolean Functions with Application to Integer Multiplication.*, IEEE Transaction on Computers, Vol. 40, No. 2, February 1991, pp.205-213.

[11] A. Chandrakasan, W. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*, IEEE press, 2001.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to algorithms - second edition*, MIT Press, Cambridge, 2001.

[13] W. J. Dally, and J. W. Poulton. *Digital systems engineering*, Cambridge University Press, 1998.

[14] E. W. Dijkstra. *A discipline of programming*, Prentice-Hall International, 1976.

[15] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir and V. Narayanan. *Leakage Current: Moore's Law Meets Static Power*, in Computer (IEEE Computer Society), vol. 36, Issue 12, pp. 68-75.

[16] P. Liljeberg, J Tuominen, S. Tuuna, J Plosila, and J. Isoaho. *Self-Timed Methodology and Techniques for Noise Reduction in NoC Interconnects*, Chapter 11 in Interconnect-Centric Design for Advanced SoC and NoC, Kluwer Academic Publishers, Boston, July 2004, pp. 285-313, ISBN 1-4020-7835-8.

[17] J. Plosila, P. Liljeberg, and J. Isoaho *Modeling and refinement of an on-chip communication architecture*, in Formal Methods and Software Engineering: 7th International Conference on Formal Engineering Methods, pp. 219-234.

[18] T. Seceleanu. *Systematic Design of Synchronous Digital Circuits*, Ph.D Thesis, Turku Centre of Computer Science, 2001.

[19] E. Sekerinski, and K. Sere. *A theory of prioritizing composition*, in The Computer Journal, vol. 39, no.8, pp. 701-712, The BCS, Oxford University Press, 1996.

[20] K. Sere and and Marina Waldén. *Data Refinement of Remote Procedures*, in Formal Aspect of Computing, vol. 12, no. 4, 2000, pp. 278-297.

[21] J. Tuominen, T. Säntti, and J. Plosila. *Towards a formal power estimation framework for hardware systems*, in Proc. of International Symposium of System of Chip, Tampere, Finland, Nov. 2005.

[22] T. Westerlund, and J. Plosila. *Time aware modeling and analysis of multiclocked VLSI systems*, LNCS, vol 4260, pp. 737-756, in Proc. of 8th International Conference on Formal Engineering Methods, Macau SAR, China, November 2006.

[23] T. Westerlund, and J. Plosila. *Time Aware System Refinement*, In REFINE workshop 2006, Macau SAR, China, Nov. 2006,