Johanna Tuominen | Tomi Westerlund | Juha Plosila

Feasibility Report on Formal Area Complexity Estimation

Turku Centre *for* Computer Science

TUCS Technical Report
No 907, August 2008

Feasibility Report on Formal Area Complexity Estimation

Johanna Tuominen
Turku Centre for Computer Science
Joukahaisenkatu 3-5 B, 20520 Turku, Finland
joeltu@utu.fi
Tomi Westerlund
University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, 20520 Turku, Finland
tovewe@utu.fi
Juha Plosila
Academy of Finland, Research Council for Natural Sciences and Engineering
juplos@utu.fi

TUCS Technical Report
No 907, August 2008

Abstract

The increasing size, complexity, speed, and power requirements poses challenges for the development methods of modern System-on-Chip designs. Traditionally, the system is implemented according to the specification using a hardware description language and then verified using simulation based methods. In case of unwanted mismatch between the initial specification and the implementation a new design cycle is needed. This is time consuming as well as an error prone approach to design systems. Therefore, there is a need for a framework that provides essential information of the system under design. With this information a designer is capable to make far-reaching decisions and avoid costly design backtracking later on in the project.

In exploring new approaches to outdo design challenges formal methods are a solution to be reckoned with. They provide an environment to specify, design, and verify systems with the benefits of rigorous mathematical basis. For this study we chose the Action Systems framework to be our base formalism. The framework will be extended with a method that allows us to estimate and analyze the area/size of a system in a formal, abstract system specification. The model relies on the size estimation of the Boolean functions. The model presents a test environment under which the formal model is evaluated and compared with the existing high level size model.

Keywords: Action Systems, area, formal, model

TUCS Laboratory
Communication Systems Distributed Systems

1 Introduction

The development methods and languages dedicated to modern, large System-on-Chip (SoC) designs are facing tremendous pressure of the ever increasing size, complexity, speed, and power requirements. To answer these challenges formal methods provide an environment to specify, design, and verify systems with the benefits of rigorous mathematical basis. Furthermore, large SoC designs need large and power hungry clock distribution networks that have been recognized one of the major challenges in modern deep submicron designs. Therefore, a keen interests towards multi-clocked and multiprocessor systems is growing. A commonly used method to alleviate the problems described above is to use Globally Asynchronous Locally Synchronous (GALS) design method introduced in [10]. A GALS design is composed of locally synchronous islands whose clocks are independent of each other, and therefore there is no need for a clock distribution network. Thus, in industry and academia the interest towards formal methods and GALS architecture is continuously increasing.

In general, the development of SoC design begins from a high level specification, which describes the functional and timing requirements of the end product. The functional requirements defines how the system operates according to the input and timing requirements are set to system components that must be satisfied. Traditionally, according to the specification, the system is implemented using, for instance, hardware specification language like VHDL or Verilog, and then verified using simulation based methods. In the case of an unwanted mismatch between the initial specification and the implementation a new design cycle is needed. This is time consuming as well as an error prone approach to design systems. Furthermore, modern wireless and mobile technology platforms pose low-power requirements for the system, which have to be taken into account during design flow.

In power consumption estimation there is a trade-off between accuracy and design time. The more detailed the analysis is the more time it consumes. To avoid costly design backtracking a designer wants to make decisions as early as possible. In this study, we present a method to evaluate the area / size of a system from formal system descriptions and compare the results of an existing high level synthesis tool. The purpose is to evaluate the accuracy of the method since the area of the system is an essential metric in a power consumption estimation [31][32][33] due to the static power consumption [18].

The formal area complexity modeling technique presented in this study is targeted to Action Systems [2], which is based on an extended version of Dijkstra's language of guarded commands [14]. The Action Systems formalism was chosen to be the base formalism because it has been successfully applied to the development of both synchronous [28] and asynchronous [23] systems. Moreover, it has a time spiced extension, Timed Action Systems, which can be used to analyze systems timing requirements. Owing to the complexity of the multiclocked and multiprocessor systems, the possibility to abstract unnecessary implementation

details at higher abstraction levels is an essential part of the used design language. With the encapsulation and abstraction techniques incorporated with the procedure based communication [24] enables a designer to use Action Systems as well as Timed Action Systems in modeling these systems. These techniques allow one to divide the development of the communication and computation into their own tasks. In addition, one of the benefits on choosing Action Systems formalism as the base formalism is the ability to use it throughout the design project: in other words, to be able to model systems from an abstract specification down to an implementable specification. The area complexity estimation is suitable for Timed Action Systems formalism, too, and it is used when discussed high level power estimation [32], [33].

1.1 Related Work

Recent years has shown, based on the active research carried out in the field, that there is a need for a rigorous development framework that operates at higher abstraction levels than the traditional approaches. That is, there is a need to evaluate the performance (time, area and power) of the system above RTL-level allowing us to detect performance related bugs earlier. The target application fields among the presented formalism varies from software systems to hardware systems and from embedded systems to hybrid systems.

To model VLSI systems several synchronous formalism exists such as *Signal* [4], *Lustre* [9] and *Esterel* [5]. All of these approaches rely on *synchronous hypothesis* in which computations and behaviors are divided into a discrete sequence of steps with deterministic concurrency. Signal is applied to modeling and validating globally asynchronous design in [20] and Esterel is extended to multiple clock domains in [6] and [26] allowing one to model both multiclocked and asynchronous systems, and furthermore, to capture asynchronous behavior within synchronous framework. These extensions enable one to use the formalisms for the same application area than the Action Systems. However, the research presented in this study is targeted to formal power modeling framework where time is a significant measure. Therefore, one should consider the timing analysis capabilities as well. The timing analysis of these synchronous languages is more restrictive than the timing analysis in the timed spiced extension of Action Systems, Timed Action Systems, because they rely on the *perfect synchrony hypothesis* that defines that the outputs are produced synchronously with the inputs. Furthermore, the rigorous system development, to our knowledge, is supported only in Signal. It supports system refinement via semantics-preserving transformations [30], but its mathematical basis seems to be less rigorous than the Refinement Calculus Paradigm [3] defined for Action Systems.

Esterel studio [40] is a tool set targeted to design SoC systems. It uses a formal description language and a verification environment to produce RTL-level system descriptions. These RTL-level descriptions can be exported at least in Verilog,

VHDL and SystemC. Furthermore, the generated system descriptions appeared to be equally good or in many cases better than hard coded ones [40]. The tool does not directly offer any area evaluation methods but its verification methods can be used to make the design as efficient as possible. For instance, a power manager design [15] framework relates to the Esterel Studio. The system specifications are written in terms of hierarchical concurrent state machines where the formal verification makes it possible to check critical properties and preserve behavior when beautifying the specification. That is, the Esterel verification environment is used to define a more efficient power management system for SoC. The power management device optimizes dynamic and static power reduction by dynamically distributing and controlling clock, reset, and power distribution for various SoC parts. This approach however is targeted to control power consumption by designing specific component using the most effective approach available. Our approach in terms of area complexity and later on power consumption is more general. That is, one can estimate performance related metrics to all components that are valid for the formalism. Furthermore, the presented model is more flexible since it is not restricted to synchronous systems. The closest high level area models operates at Boolean level, which are discussed in the next subsection.

1.1.1 On the Boolean Complexity

In an early work [27] of Shannon the area complexity of Boolean function was studied (switch count). In this paper Shannon proved that the asymptotic complexity of Boolean functions is exponential in the number of inputs (m), and that for large m, almost every Boolean function is exponentially complex. Muller demonstrated the same result for Boolean functions implemented using logic gates (gate-count measure) in [21]. Over the years several other researchers have reported results related to the area complexity of Boolean functions, for instance, the relationship between area complexity and entropy ($\mathcal{H}$) is reported, for instance, in [17], [25], [13] and [11]. Cheng and Agrawal [11] measured the area complexity as a literal count and it was generated for small number of inputs from randomly generated Boolean functions. As one tries to apply that model to realistic VLSI circuits, it quickly breaks down due to the exponential dependence on the number of inputs. Nemani et. al. [22] proposed a method for predicting the area of a single output Boolean function given only its functional specification and no structural information. The presented area complexity model is based on the *area cube complexity* and the results were compared with the SIS high-level synthesis tool nowadays known as MVSIS [36]. The presented model was reasonably accurate compared with the results given by the SIS tool.

Another approach to model the area complexity of a Boolean function is to use a graphical model. A Boolean function can be represent as a directed acyclic graph, where the size of a function can be evaluated by calculating the number of nodes needed to present the function. These graphs are often referred to as *Binary*

Decision Diagrams (BDD) and described, for instance, in [1] [7] [8]. Binary Decision diagram represent a Boolean function as a directed acyclic graph with each vertex labeled by Boolean variable. In an *Ordered Binary Decision Diagram (OBDD)*, the vertex label occurs in the same order along all directed paths. This presentation has many desirable algorithmic properties. For instance, it has proved to work well as a data structure for symbolically representing and manipulating Boolean functions [7]. Furthermore, for a given variable ordering, the smallest OBBD for a particular Boolean function is unique.

Several tools and packages exist to automate the BDD manipulation. For instance, packages such as CUDD [37] and BuDDy [38] offer functions to manipulate BDDs via C++ interface. However, benchmark circuits such as ISCAS and ACM/SIGMA benchmark set do not support these tools. University of Berkeley has research groups [36], [39] for high level synthesis and verification, which offer their own tool sets for that including the possibility to use and manipulate BDDs. Furthermore, decision diagrams are often related to verification tools, for instance, *Esterel* verification environment (*Xeve*) [41] uses BDDs to symbolically describe input events to analyze state machines.

2 Formal Basis

In this section we will introduce the formal basis for our area complexity (size) analysis. We start by reviewing the Action Systems formalism after which we discuss the area complexity modeling of those action constructs, which are essential for the work. Finally, we present the area complexity modeling at an action system level. The accuracy evaluation and comparison between existing area models will be given in the forthcoming sections.

2.1 Actions

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

$A ::= abort$	(*abortion, non-termination*)
$\mid skip$	(*empty statement*)
$\mid x := x'.Q$	(*non-deterministic assignment*)
$\mid x := e$	(*(multiple) assignment*)
$\mid g \rightarrow A$	(*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*)

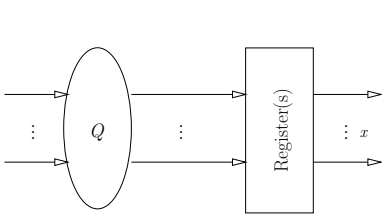
where A and A_i, $i = 1, 2$, are actions; x is a variable or a list of variables; e is an expression or a list of expressions; and g and Q are predicates (Boolean conditions). An action is considered to be *atomic*, which means that only the initial and final states are *observable* by the system. Therefore, when an action is selected for execution, it is completed without any interference from other actions. If an action does not establish any post-condition it behaves as an $abort$ statement (a never terminating statement), and if it does not change the state at all, it behaves as an $skip$ statement (an empty statement). The *non-deterministic choice* chooses one of the enabled actions non-deterministically without a chance of an external influence. The *sequential composition* executes the actions one by one in the given order. The *prioritized composition* [29] is a composition in which the execution order of enabled actions is prioritized. We have: $A \mathbin{/\!\!/} B \mathrel{\widehat{=}} A \mathbin{[\!]} \neg gA \rightarrow B$, where the highest priority belongs to the leftmost action in the composition; therefore, the leftmost enabled action is always chosen for execution. 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: $vA \mathrel{\widehat{=}} wA \cup rA$.

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 $\mathbf{wp}((A \mathbin{[\!]} B), q) = \mathbf{wp}(A, q) \wedge \mathbf{wp}(B, q)$. 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: $sA \mathrel{\widehat{=}} A \mathbin{[\!]} \neg gA \rightarrow abort$.

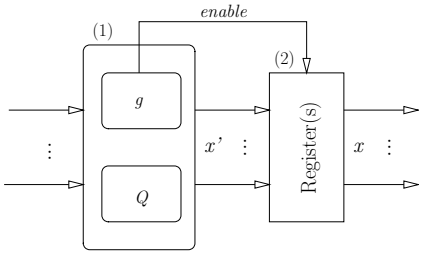
A *quantified composition* of actions is denoted by: $[\bullet\ 1 \leq i \leq n : A_i]$, and it is defined by: $A_1 \bullet \ldots \bullet A_n$, where the bullet $\bullet$ denotes any of the composition operators, and n is the number of actions. Furthermore, we have a *substitution* operation, denoted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

2.1.1 On modeling area complexity of actions

To model area complexity of an arbitrary action, we utilize the following information: the read and write variables of an action, and the abstract specification of its functionality. We start by defining *a variable width*, which is the number of bits needed to present the variable. At a high abstraction level, it is typical that variables are other types than Boolean, and therefore the width information is the basic necessity to model area complexity [32]. Thus, the area complexity of a variable x is defined by:



(a) A non-deterministic assignment statement $x := x'.Q$.



(b) A guarded non-deterministic assignment $g \rightarrow x := x'.Q$

Figure 1: Illustrations of assignments

$C(x) \mathrel{\widehat{=}} w_x$ (*area complexity of a variable (x)*)

---

where w_x is the variable width. For a set of variables, say S, the area complexity is obtained by adding together the widths of the variables $x \in S$. We have:

$C(S) \mathrel{\widehat{=}} \sum_{x \in S} w_x$ (*area complexity of a set (S)*) (1)

The premise of our area complexity modeling, in addition to the variable widths, is the *non-deterministic assignment* ($x := x'.Q$) because it can be used to describe any operations performed on variables in an action context. Consider an action $A \mathrel{\widehat{=}} x := x'.Q$, where the predicate Q is evaluated and the result is assigned to the variable x. The area complexity modeling of A is divided into two parts: *assignment*, and *predicate evaluation*. In assignment part the result of the predicate evaluation is written into a variable and it is illustrated as a chain of storage elements in our area complexity model, as shown in Fig. 1(a) and defined by: $C(wA) = \sum_{x \in wA} w_x$, where the wA is the write set of A. Observe that the definition is based on (1). The predicate evaluation on the other hand is thought as a "combinatorial cloud", shown in Fig. 1(a), which at a lower abstraction level is the logic that performs the computation. In general, the predicate is a Boolean function. The presented model is targeted to more abstract descriptions, and therefore the size evaluation methods for Boolean functions are not directly applicable. For instance, an action may define an addition operation but it does not define whether the addition is implemented using a ripple carry adder or a carry-look-ahead adder structure. Furthermore, by adopting directly the definition (1) for the set rQ, it would give the same area complexity regardless of the operation that Q performs. For instance, the area complexity between two predicates say $Q_1 \mathrel{\widehat{=}} v := v'(x' = y)$ and $Q_2 \mathrel{\widehat{=}} v := v'(v' = x + y)$, would be the same if we adopt the similar approach that was selected for the assignment.

Usually a combinatorial logic forms a layered structure, which can be described using tree alike structures. To model the depth of a tree, our model relies

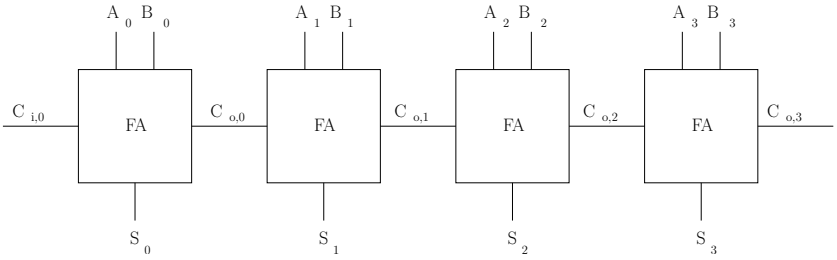


Figure 2: Example: Ripple Carry Adder Structure.

on BDD modeling of Boolean functions. Naturally, we cannot generate exact BDD because we only have an abstract description of a logic operation. In addition to the tree model, we will exploit the Shannon's size equation presented in the previous section but not directly for the set rQ because with the large number of input arguments it quickly breaks down due to its exponential dependency. Furthermore, as the abstraction level of the system decreases during system refinement [3], the system descriptions will be closer to Boolean definitions, and therefore the evaluation methods targeted to Boolean functions will be more and more accurate. Thus, in the end there should be enough information to generate the BDD description for the system, and, furthermore, to use it as an area complexity model.

To model area complexity of the predicate evaluation we define the set rQ, which consist of those variables that appear in the predicate Q ($rQ \in rA$). First we define the area complexity of the set rQ, denoted by $C(rQ)$ as stated in (1). To imitate the tree like structure mentioned above, we set the area complexity of the set $C(rQ)$ as a root node. The number of children is calculated by $\frac{C(rQ)}{|rQ|}$, where the area complexity ($C(rQ)$) of the set is divided by the cardinality of the set ($|rQ|$). The cardinality of the set describes the number of variables in the set, that is, the number of children is the average variable width in the set rQ. The idea is that if we have a Boolean function between variables the operation is carried out in a "bit wise" manner. As an example, assume that we perform an addition operation between two variables having width of four. At circuit level, see Fig 2, the variables are added together in a way that the least significant bits of both variables are added together and then the most significant bits (and carry). Each child node in the tree has two possible output values, namely $'0'$ or $'1'$. That is, every node has $|rQ|$ arguments and by combining these variables using different orders we have $2^{|rQ|}$ different combinations to produce either $'0'$ or $'1'$ as an output value according to the Shannon's size equation. Adopting this approach we assume that the number of input arguments does not increase so dramatically, and therefore the exponential dependence does not have significant negative effect to the model. The area complexity of the predicate evaluation (Q) in the *non-deterministic assignment* is defined by:

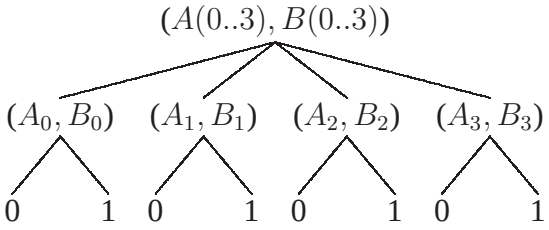


Figure 3: Example area complexity modeling.

$$C(Q) \mathrel{\widehat{=}} \left\lceil \frac{C(rQ)}{|rQ|} \right\rceil \cdot 2^{|rQ|} \qquad (\textit{area complexity of a predicate evaluation}) \quad (2)$$

where the first term calculates the average width of the input variables of the predicate. Observe that the variable width in a system description, at any abstraction level, is usually integer, and therefore the result of the average variable width is rounded. Let us illustrate the area complexity evaluation with the following to examples:

Example 1 The predicate Q defines integer addition: $Q \mathrel{\widehat{=}} A + B$, where the variables A and B have widths $w_A = W_B = 4$. The set rQ of the predicate Q is $rQ \mathrel{\widehat{=}} \{A, B\}$. To clarify the "bit wise" addition where the model is based on, see the Fig. 2. We start by calculating the area complexity $C(rQ)$ for the set rQ: $C(rQ) = 4 + 4 = 8$. The number of children in the tree is then calculated by dividing the $C(rQ)$ with the cardinality of the set $|rQ|$, we have: $\frac{C(rQ)}{|rQ|} = \frac{8}{2} = 4$. Each of these four children have two possible output values and the area complexity of one child node is $2^2 = 4$. The area complexity of the predicate Q is calculated using (2): $C(Q) = \left\lceil \frac{C(rQ)}{|rQ|} \cdot 2^{|rQ|} \right\rceil = 8$. The tree model for the area complexity model is shown in Fig. 3.

End of example.

In Example 1 both of the variables involved in the area complexity evaluation had the same width. Let us further introduce an area complexity modeling with the general example with input variable widths.

Example 2 Let us assume that the set of variables rQ of the predicate Q is $rQ \mathrel{\widehat{=}} \{X, Y, Z\}$. The variables X, Y, Z are of same type and their widths are $w_X = 4$, $w_Y = 4$ and $w_Z = 6$. The area complexity of the set rQ is $C(rQ) = w_X + w_Y + w_Z = 14$, and for the predicate evaluation we have: $C(Q) = \lceil \frac{14}{3} \rceil \cdot 2^3 = 40$. The tree like structure is shown in Fig. 4, where the number of successor nodes from the root is defined by the average variable width. Each of these nodes have three input arguments and assumed to perform a Boolean operation. Therefore, the area complexity is calculated using the Shannon's 2^m equation, where $m = 3$ in this example.

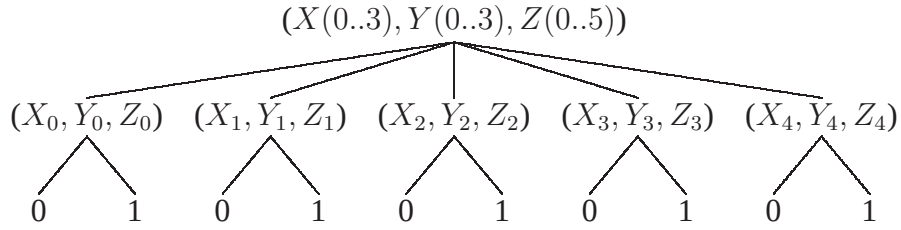


Figure 4: Example area complexity illustration.

End of example.
To summarize, the area complexity of the action $A \mathrel{\widehat{=}} x := x'.Q$ is defined by:

$$C(A) \mathrel{\widehat{=}} C(wA) + C(Q) \qquad (\textit{area complexity of an action } A) \quad (3)$$

where the first term is the area complexity of the write set is defined in (1) and the second term is the area complexity of the predicate evaluation defined in (2). A special case in the area complexity modeling of the non-deterministic assignment occurs when the predicate Q is of form: $Q \mathrel{\widehat{=}} (x'. = y)$, where y is a variable. For instance, consider an action $A \mathrel{\widehat{=}} x := x'.(x' := y)$, where x and y are the variables of the same type. The actions area complexity is defined by: $C(A) \mathrel{\widehat{=}} C(wA)$, where the area complexity of the predicate $C(Q)$ is zero as there is no computation just the assignment ($x' := y$).

Consider a guarded action of form: $g \rightarrow B$. When the guard g evaluates to *true* (T) the action B is enabled, otherwise the action is disabled (the g evaluates to *false* (F)). An illustration of the guarded non-deterministic assignment for area complexity modeling is shown in Fig. 1(b). The area complexity modeling starts by defining the set of write variables wB and read variables rB of the action. The area complexity of the set wB is defined as shown in (1). However, the area complexity of the read set rB requires further studying. The variable(s) in the read set appears either in the guard g, in the predicate Q, or in both. Therefore, we define a set of variables rg that appear in the guard g and a set of variables rQ that appear in the predicate Q ($rB \mathrel{\widehat{=}} rg \cup rQ$). The area complexity is evaluated for the sets rg and rQ separately by replacing the rQ with the set rg in (2). Thus, for an action B we have:

$$C(B) \mathrel{\widehat{=}} C(wB) + C(Q) + C(g) \qquad (\textit{area complexity of an action } B) \quad (4)$$

The predicate Q in the non-deterministic assignment (or in the guarded non-deterministic assignment) is often used to describe arithmetic operations. The above presented area complexity model would give identical result, for instance, for multiplication and addition due to the high abstraction level. In other words, the read set of the multiplication action and the read set of the addition actions

contains the same variables, and therefore the area complexity model does not make difference between these operations. To overcome this, a *complexity factor* $\phi \in \mathbb{R}^+$ is introduced for arithmetic operations, and thus the area complexity of the predicate Q evaluation becomes of form:

$$C(Q) \widehat{=} \left\lceil \frac{\sum_{v \in w_v} w_v}{|rQ|} \right\rceil \cdot (2^{|rQ|})^{\phi}$$

The value of the complexity factor can be any positive real number, and furthermore, it is adjusted by the designer. In this thesis, the complexity factor is adjusted in a way that it takes the high abstraction level into account. For instance, the complexity factor for addition is assumed to be one ($\phi = 1$) and for multiplication it is assumed to be ($\phi = 2$). This follows from the complexity relation between binary addition and multiplication. That is, in [12], the complexity of binary addition is n and the complexity of schoolbook multiplication is n^2. Observe that, the complexity factor is adjustable, for instance, when the abstraction level decreases. However, this study is targeted to abstract system descriptions, and therefore, it is fair to assume the above presented values.

2.2 Action System

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

sys $\mathcal{A}$ (**imp** p_I; **exp** p_E;)(u_A;) ::
|[
type
$type$: Def;
variable
l_A;
private procedure
p_I(**in** x : **out** y): (P_I);
public procedure
p_E(**in** x : **out** y): (P_E);
action
A_i: (aA_i);
initialisation
$u_A, l_A := u_A 0, l_A 0$;
execution
forever do `composition of actions` A_i **od**
]|

where we can identify three main sections: *interface*, *declaration* and *iteration*. The interface part declares those variables, u_A, that are visible outside the action system boundaries and therefore accessible by other action systems. Global variables maybe of type input, output or bi-directional input-output, and the types are denoted by the following identifiers: **in**, **out** and **inout**, respectively. It also introduces *interface procedures* p_I and p_E that are imported in or introduced in

and exported by the system. These are denoted by the **imp** and **exp** identifiers, respectively. If an action system has no interface variables or procedures, it is *a closed action system*, otherwise it is *an open action system*.

The declaration part introduces all new, local type definitions (**type**) and the local variables l_A (**variable**). Furthermore, the declaration part introduces private p_I and public p_E procedures (**private procedure** / **public procedure**) and action definitions aA_i (**action**) that perform operations on local and global variables. Furthermore, a label A_i is given for every action definition.

The operation of action system is started by the initialization in which the variables are set to predefined values. In the iteration part, the **execution** clause, actions are selected for execution based on their composition and enabledness. This is continued until there are no enabled actions, whereupon the computation terminates. Hence, an action system is essentially an initialized block with a body that contains an iteration, that is, a statement which is repeatedly executed.

2.2.1 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 **execution** clause has the form: **forever 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.

2.2.2 Procedures

A body P of the procedure $p : p($**in** $x;$ **out** $y) : P$, is in general any atomic action A, possibly with some auxiliary local variables u initialized to $u0$ 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: $P[$**var** $v;$ **init** $u := u0; A(g, l, u, x, y)]$. 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: **proc** $p : A(g, l)$.

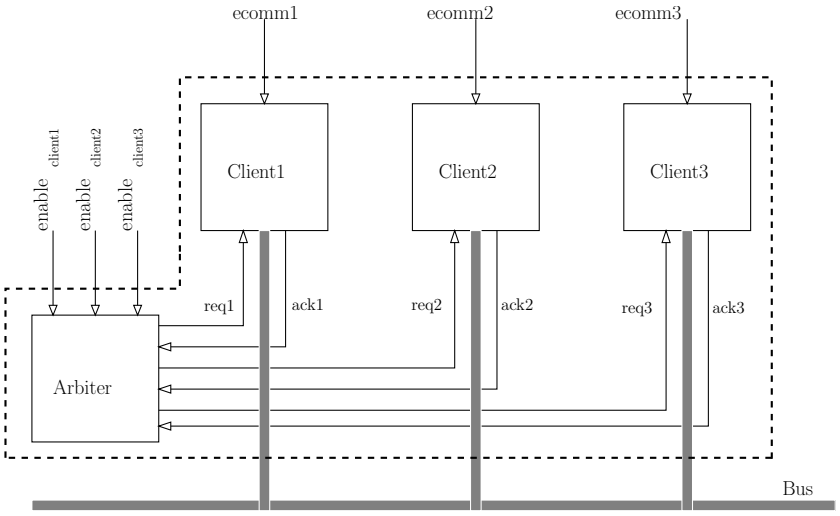


Figure 5: Bus arbitration with three clients.

2.2.3 Area complexity of systems

Consider a system $\mathcal{A} \parallel \mathcal{E}nv$ where the latter is the environment of the former. Its area complexity is denoted by: $C(\mathcal{A} \parallel \mathcal{E}nv)$ and defined as follows: we form a set $S_{\mathcal{A}}$, which contains actions A, and Env of the system $\mathcal{A} \parallel \mathcal{E}nv$. It is easy to see that $S_{\mathcal{A}\parallel\mathcal{E}nv} \mathrel{\widehat{=}} S_{\mathcal{A}} \cup S_{\mathcal{E}nv}$, where the sets $S_{\mathcal{A}}$ and $S_{\mathcal{E}nv}$ are defined by $S_{\mathcal{A}} \mathrel{\widehat{=}} \{A | A \texttt{ is an action of } \mathcal{A}\}$ and $S_{\mathcal{E}nv} \mathrel{\widehat{=}} \{Env | Env \texttt{ is an action of } \mathcal{E}nv\}$. The area complexity of the system is the sum of the complexities of actions in the set $S_{\mathcal{A}\parallel\mathcal{E}nv}$ and defined by:

$$C(\mathcal{A} \parallel \mathcal{E}nv) \mathrel{\widehat{=}} \qquad (\textit{area complexity of systems}) \quad (5)$$
$$C(\mathcal{A}) + C(\mathcal{E}nv)$$
$$\sum_{S_{\mathcal{A}}} C(A) + \sum_{S_{\mathcal{E}nv}} C(Env)$$

---

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 $\mathcal{E}nv$.

Example 3. Consider a system $\mathcal{A}rbiter \parallel \mathcal{E}nv$ [$\parallel 1 \leq i \leq 3 : \mathcal{C}ontroller(i)$], where the $\mathcal{A}rbiter$ controls the bus access between the three controllers systems and the $\mathcal{E}nv$ system as shown in Fig. 5. The dotted line in Fig. 5 describes the above mentioned system. In other words, the environment and the clients are not described here, and furthermore, they are not part of the area complexity evaluation. The $\mathcal{A}rbiter$ defines the operation of the arbiter and its interface to the

environment and to the clients. The $\mathcal{C}ontroller$ system defines the interface to the $\mathcal{A}rbiter$ from the client side and to the data bus. Originally the arbiter system is a benchmark circuit from ACM/SIGMA benchmark set [35] and it is used to evaluate the accuracy of the presented area complexity model in the forthcoming sections, and therefore the presented parallel system imitates the operation of the benchmark circuit. The operation of the benchmark is described in A.1.3. The $\mathcal{A}rbiter$ system is defined by:

sys $\mathcal{A}rbiter$ (**in** $req1, req2, req3, ereq1, ereq2, ereq3 : Bool;$
out $ack1, ack2, ack3, eack1, eack2, eack3 : Bool;$
$areq1, areq2, areq3 : Bool;$) ::

|[

type
$Status = \{idle, operation\};$
variable
$direction : Status;$
action
$A1 : req1 \wedge \neg ack2 \wedge \neg ack3 \wedge direction = idle \rightarrow$
$ack1, direction := T, operation;$
$A2 : \neg ack1 \wedge req2 \wedge \neg ack3 \wedge direction = idle \rightarrow$
$ack2, direction := T, operation;$
$A3 : \neg ack1 \wedge \neg ack2 \wedge reg3 \wedge direction = idle \rightarrow$
$ack3, direction := T, operation;$
$A4 : \neg req1 \wedge ack1 \rightarrow ack1, direction := F, idle;$
$A5 : \neg req2 \wedge ack2 \rightarrow ack2, direction := F, idle;$
$A6 : \neg req3 \wedge ack3 \rightarrow ack3, direction := F, idle;$
$A7 : ereq1 \wedge \neg eack2 \wedge \neg eack3 \wedge direction = idle \rightarrow$
$eack1areq1, direction := T, T, operation;$
$A8 : ereq2 \wedge \neg eack1 \wedge \neg eack3 \wedge direction = idle \rightarrow$
$eack2, areq2, direction := T, T, operation;$
$A9 : ereq3 \wedge \neg eack1 \wedge \neg eack2 \wedge direction = idle \rightarrow$
$eack3, areq3, direction := T, T, operation;$
$A10 : \neg ereq1 \wedge eack1 \wedge aack1 \rightarrow eack1, areq1, direction := F, F, idle;$
$A11 : \neg ereq2 \wedge eack2 \wedge aack2 \rightarrow eack2, areq2, direction := F, F, idle;$
$A12 : \neg ereq3 \wedge eack3 \wedge aack3 \rightarrow eack3, areq3, direction := F, F, idle;$
initialisation
$direction := idle;$
$ack1, ack2, ack3, req1, req2, req3 := F;$
$eack1, eack2, eack3, ereq1, ereq2, ereq3 := F;$
$areq1, areq2, areq3 := F;$
execution
forever do $[\ ]\ 1 \leq i \leq 12 : A_i]$ **od**

]|

where the actions $A1, A2,$ and $A3$ describe the operation when the client side requests the bus access. The bus access is granted for one client at a time by setting one of the acknowledgement signals *ack1, ack2,* or *ack3* to *true*. After the

data transfer is completed the actions $A4, A5,$ and $A6$ set the acknowledgement signals *ack1, ack2,* or *ack3* to *false* after which a new communication may begin. The original benchmark circuit defines that the environment may select the client as well. To ensure that the client side and the environment side cannot obtain simultaneous write access to the system bus, we define a variable $direction$ which has two possible states *idle, operation*. A communication cycle may begin only if the state is *idle*. The communication from the environment is defined by the actions $A7 - A12$. The environment requests the access by setting request signals *ereq1, ereq2,* or *ereq3* to *true*, and if the access is granted then $\mathcal{A}rbiter$ sends a request to controller unit by setting one of the requests *areq1, areq2,* or *areq3* to *true*. The client performs data transfer and sends an acknowledgement to the arbiter *aack1, aack2, aack3* after which the environment is acknowledged *eack1, eack2,* or *eack3*. Observe that for simplicity the request *areq2, areq3* and acknowledgement *aack2, aack3* signals are not illustrated in Fig. 5. Furthermore, these actions define only the communication between the environment and the selected client through the arbiter. However, there were no definition on the type of the communication that environment performs with the clients, and thus we do not give a detailed specification of that. For instance, the environment may transfer data to the client, which in turn transfers the data into the system bus. Observe that neither the presented system nor the original benchmark system cannot guarantee *fairness*.

The controller interface for each client is defined as follows:

sys $\mathcal{C}ontroller(i)$ (**in** $ack(i), creq(i) : Bool;\ \ din(i) : Data;$
out $req(i), cack(i) : Bool;\ \ dBus : Data;$
$aack(i) : Bool;$) ::
|[
action
$C1 : creq(i) \wedge \neg cack(i) \rightarrow req(i) := T;$
$C2 : ack(i) \rightarrow dBus, cack(i) := din(i), T;$
$C3 : \neg creq(i) \wedge cack(i) \rightarrow cack(i), req(i) := T, F;$
$C4 : areq(i) \rightarrow aack(i) := T;$
$C5 : \neg areq(i) \wedge aack \rightarrow aack := F;$
initialisation
$dBus, din(i) := dBus0, din(i)0;$
$creq(i), cack(i) := F;$
$aack(i) := F;$
execution
forever do $C1(i)\ [\!]\ C2(i)\ [\!]\ C3(i)\ [\!]\ C4(i)\ [\!]\ C5(i)$ **od**
]|

The controller system is identical in each clients, and therefore we used a single system definition using quantified composition. The client side starts the communication cycle by setting the request $creq(i)$ to the controller *true* after which the controller sets the request signal $req(i)$ to the arbiter to *true* ($C1$). If the bus

Table 1: Area complexity evaluation of the $\mathcal{A}rbiter$ system

Action	Write variables	Read variables	Area complexity
$A1$	$ack1(1), direction(1)$	$req1(1), ack2(1),$ $ack3(1), direction(1)$	$3 + \lceil \frac{4}{4} \rceil \cdot (2^4) = 18$
$A2$	$ack2(1), direction(1)$	$ack1(1), req2(1),$ $ack3(1), direction$	$2 + \lceil \frac{4}{4} \rceil \cdot (2^4) = 18$
$A3$	$ack3(1), direction$	$ack1(1), ack2(1),$ $req3(1), direction(1)$	$2 + \lceil \frac{4}{4} \rceil \cdot (2^4) = 18$
$A4$	$ack1(1), direction(1)$	$req1(1), ack1(1)$	$2 + \lceil \frac{2}{2} \rceil \cdot (2^2) = 8$
$A5$	$ack2(1), direction(1)$	$req2(1), ack2(1)$	$2 + \lceil \frac{2}{2} \rceil \cdot (2^2) = 8$
$A6$	$ack3(1), direction(1)$	$req3(1), ack3(1)$	$2 + \lceil \frac{2}{2} \rceil \cdot (2^2) = 8$
$A7$	$eack1(1), direction(1),$ $areq1(1)$	$ereq1(1), eack2(1),$ $eack3(1), direction(1)$	$4 + \lceil \frac{4}{4} \rceil \cdot (2^4) = 19$
$A8$	$eack2(1), direction(1),$ $areq2(1)$	$ereq2(1), eack1(1),$ $eack3(1), direction(1)$	$4 + \lceil \frac{4}{4} \rceil \cdot (2^4) = 19$
$A9$	$eack3(1), direction(1),$ $areq3(1)$	$ereq3(1), eack1(1),$ $eack2(1), direction(1)$	$4 + \lceil \frac{4}{4} \rceil \cdot (2^4) = 19$
$A10$	$eack1(1),$ $direction(1), areq1(1)$	$ereq1(1), eack1(1),$ $aack1(1)$	$3 + \lceil \frac{3}{3} \rceil \cdot (2^3) = 11$
$A11$	$eack2(1),$ $direction(1), areq2(1)$	$ereq2(1), eack2(1),$ $aack2(1)$	$3 + \lceil \frac{3}{3} \rceil \cdot (2^3) = 11$
$A12$	$eack3(1),$ $direction(1), areq3(1)$	$ereq3(1), eack3(1),$ $aack3(1)$	$3 + \lceil \frac{3}{3} \rceil \cdot (2^3) = 11$

access is granted, data is transferred to the system bus $dBus$ and the acknowledgement to the client side is set to *true* indicating that data transfer is completed ($C2$). Once the client side sets the request $creq(i)$ to *false*, the acknowledgement to the client side $cack(i)$ and the request $req(i)$ to the arbiter are set to *false* ($C3$). The actions $C4$ and $C5$ are the interface for the communication between environment and client through the arbiter. Therefore, the functionality between this kind of communication parties is not described.

The area complexity of the system is calculated as defined in Sect. 2.2.3: $C(\mathcal{A}rbiter \parallel \mathcal{C}lient1 \parallel \mathcal{C}lient2 \parallel \mathcal{C}lient3) \mathrel{\widehat{=}} C(\mathcal{A}rbiter) + C(\mathcal{C}lient1) + C(\mathcal{C}lient2) + C(\mathcal{C}lient3)$, where the area complexities are calculated for each system separately and then added together. We start by defining the read and write sets of the $\mathcal{A}rbiter$ system, and then using this information we calculated the area complexity of the actions in the $\mathcal{A}rbiter$ system shown in Table 1, where the number in the parenthesis after each variable is its width. The width of the $direction$ variable is one because all the two states in the type definition can be presented using one bit. Based on Table 1, the area complexity of the system

Table 2: Area complexity evaluation of the $\mathcal{C}ontroller$ system

Action	Write variables	Read variables	Area complexity
$C1$	$req(i)(1)$	$creq(i)(1), cack(i)(1)$	$1 + \lceil \frac{2}{2} \rceil \cdot (2^2) = 5$
$C2$	$dBus(32), cack(i)(1)$	$ack(i)(1)$	$33 + \lceil \frac{1}{1} \rceil \cdot (2^1) = 34$
$C3$	$cack(i)(1), req(i)(1)$	$creq(i)(1), cack(i)(1)$	$2 + \lceil \frac{2}{2} \rceil \cdot (2^2) = 6$
$C4$	$aack(i)(1)$	$areq(i)(1)$	$1 + \lceil \frac{1}{1} \rceil \cdot (2^1) = 3$
$C5$	$aack(i)(1)$	$areq(i)(1), aack(i)(1)$	$1 + \lceil \frac{2}{2} \rceil \cdot (2^2) = 5$

$\mathcal{A}rbiter$ is $C(\mathcal{A}rbiter) \mathrel{\widehat{=}} C(A1) + \ldots + C(A12) = 168$.

In a similar manner we evaluate the controller system. Each of the clients have a similar controller unit, and therefore it is enough to present the area complexity calculations for one controller and then multiply it by three. The result of the area complexity estimation is shown in Table 2, where the area complexity of the system $\mathcal{C}ontroller$ is $C(\mathcal{C}lient3) = C(C1) + C(C2) + C(C3) = 53$. This area complexity is also the area complexity of the systems $\mathcal{C}lient2$ and $\mathcal{C}lient3$. Therefore, the area complexity of the parallel system $\mathcal{A}rbiter \parallel \mathcal{C}lient1 \parallel \mathcal{A}rbiter \parallel \mathcal{C}lient1$ is $C(\mathcal{A}rbiter \parallel \mathcal{C}lient1 \parallel \mathcal{C}lient2 \parallel \mathcal{C}lient3) \mathrel{\widehat{=}} C(\mathcal{A}rbiter) + C(\mathcal{C}lient1) + C(\mathcal{C}lient2) + C(\mathcal{C}lient3) = 168 + 53 + 53 + 53 = 327$. The accuracy of this modeling technique will be evaluated in the forthcoming sections.
End of example.

3 Test Environment

The accuracy of the presented area complexity model is estimated using binary decision diagrams. As stated in 1.1.1, there are several tools to manipulate binary decision diagrams. The amount of available benchmark circuits had significant impact to our test environment selection. That is, it ruled out the BDD packages with C++ interface and drove our interests to the high-level synthesis tools from UC Berkeley, and to *Berkeley Logic Interchange Format (BLIF)*. Next we shortly present the BLIF format and then we discuss tool alternatives. Finally, we presented the properties of the selected test environment using an example circuit.

3.1 Berkeley Logic Interchange Format

The *Berkeley Logic Interchange Format* is used to describe a logic-level hierarchical circuit in a textual form. A BLIF file represents a sequential circuit as an interconnection of logic gates and latches as a state transition table of a finite state machine or both. The syntactical information of the BLIF language can be found at MVSIS group home page [36]. Furthermore, BLIF is an entry point for logic optimizers such as *Synthesis System (SIS)* and its follower *Multivalued*

Logic Synthesis System (MVSIS), a synthesis tool developed by MVSIS group at UC Berkeley [36]. The BLIF-MV is an extension to the BLIF format. It is a language designed for describing hierarchical systems with non-determinism. The non-determinism is accomplished by allowing the use of non-deterministic gates in an input description. These gates generate the output arbitrarily from a set of predefined outputs. Although the BLIF is the input language for the *SIS* logic synthesis tool has constructs for hierarchical system description they are automatically flattened into single level circuit once they are read in. This is because the internal structure of *SIS* does not support hierarchical representations. The successor of the *SIS* tool supports the *BLIF-MV* format, too.

3.2 Tool Selection

After studied both BLIF and BLIF-MV languages, the use of the MVSIS tool seemed to be a good choice. The tool can, for instance, do initial manipulation of a hardware description before it is encoded into binary and processed by a standard binary logic synthesis programs. Furthermore, it offers a front-end to a software compiler, which allows its usage in embedded systems applications. However, we encountered several problems, in particular, when a benchmark circuit written by BLIF-MV was read in. Therefore, we also considered another high-level synthesis tool: Verification Interacting with Synthesis [39] (VIS), which is a joint work of University of Berkeley, University of Colorado at Boulder, and more recently University of Texas at Austin. This program offers pretty much similar BDD manipulation environment, and, furthermore, it has a compiler, which allows to turn *Verilog* code into BLIF-MV. The Verilog compiler supports only a small subset of the language, and therefore any system description written in Verilog is not applicable. More information on compiler, the Verilog subset, and the VIS tool can be found from the VIS documentation [39]. The syntactical aspects of the different input languages are not studied in this paper because we adopt existing and documented benchmarks circuits [35].

3.2.1 Verification Interacting with Synthesis tool

VIS tool has three main parts: a front-end to read and traverse a hierarchical system described by BLIF-MV, which may have been translated from Verilog; a verification core; and a core to perform logic synthesis. In this study, we exploit the front-end and the synthesis parts of the tool, which are highlighted using dotted lines in Fig. 6. More precisely, we use the BDD manipulation properties of the tool, which are demonstrated using the Greatest Common Divisor (GCD) Benchmark circuit [35]. The GCD algorithm (implemented by the benchmark circuit) is described in Appendix A.1.4. The BDD generation starts by reading the BLIF-MV (or BLIF) description into VIS, where it is stored as a “hierarchy” tree. The term “hierarchy” tree refers to the method, which the tool uses to store the initial

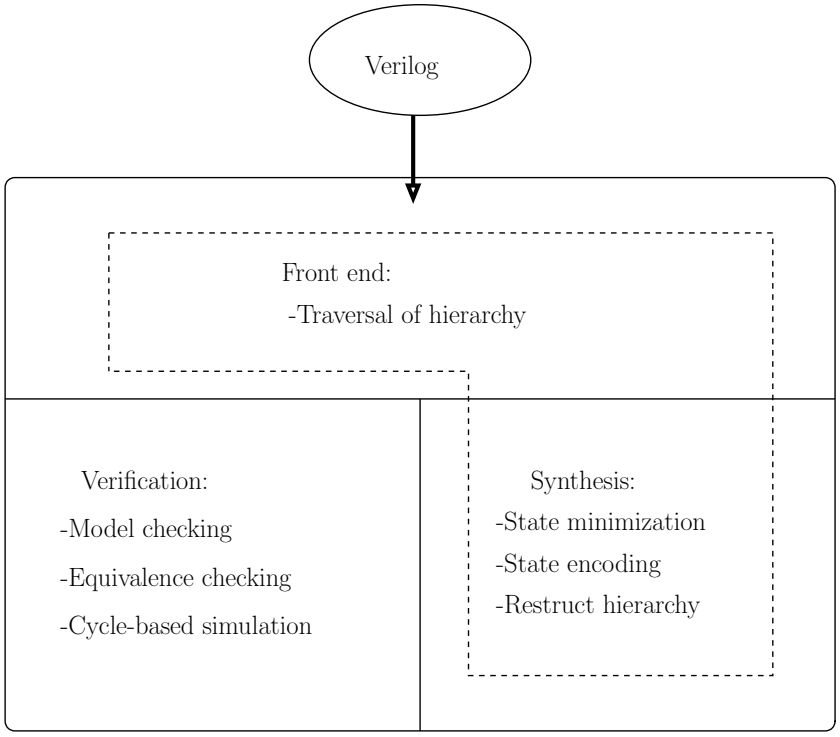


Figure 6: Overview of the VIS tool.

specification of the design. It consist of modules that in turn consist on sub modules that are related in some fashion. This relation is described as a table, which implements the output function in terms of the sub module inputs. The command line procedure is shown below:

```
vis release 2.1 (compiled 27-Mar-08 at 11:59 AM)
vis> read_blif_mv test_data/gcd.mv
vis> print_hierarchy_stats
Model name = testGcd, Instance name = testGcd
inputs = 17, outputs = 0, variables = 97, tables = 40,
latches = 17, children = 1
```

where the BLIF-MV file is read in using the `read_blif_mv` command. In a similar manner a BLIF file could be read in with `read_blif` command. The properties of the hierarchy tree is enquired using `print_hierarchy_stats` command.

The hierarchy structure described above can be expressed as a tree. The root of the tree is the main module, and the child nodes are the lower level instantiations of modules. The hierarchy in a VIS can be traversed in a similar manner as directories in UNIX. It is possible to reach the desired node by walking up and down with the `cd` command. At any node the verification and synthesis opera-

tions can be performed. The command `pwd` prints the name of the current node and the command `ls` lists all the nodes from the current node.

The first step towards the BDD construction is to ”flatten” the hierarchical description into a single network (netlist of multivalued logic gates). The output is computed from the inputs of the design by the network circuit which consist of logic gates, interconnections between them, and latches to represent the sequential elements. The `flatten_hierarchy` command does this all and the `print_network_stats` command prints the network statistics. For the *GCD* example these commands work as follows:

```
vis> flatten_hierarchy
vis> print_network_stats
testGcd  combinational=657  pi=17  po=0  latches=45
pseudo=0  const=45  edges=1362
```

The network description is transferred into a functional description that represents the output and next state variables as a function of the inputs and next state variables. In this study, the BDDs are used to represent Boolean and discrete functions. Before creating a BDD it is necessary to order the variables to reach the most optimal solution. The ordering is started by the static ordering method by invoking the command: `static_order`, which gives the initial ordering. The ordering can be canceled at any point by invoking the `flatten_hierarchy` command or the current variable ordering can be written out using the command `write_order`. For the GCD, the static ordering is invoked as follows:

```
vis> static_order -o all -r depth
vis> print_bdd_stats
```

where the `-o all` defines that the ordering is done to all nodes and the `-r depth` defines the ordering method. The ordering method is selected to give the most optimal BDD structure. In other words, the ordering method, which gives the smallest BDD is selected. The command `print_bdd_stats` gives information on ordering, memory usage and more importantly the BDD node count. The node count for *GCD* is 768, which is the size estimate for the circuit. Often the ordering can be improved by using dynamic variable ordering methods, which are techniques to reorder the BDD variables to reduce the size of the existing BDDs. Furthermore, dynamic ordering may be time consuming but sometimes it can reduce the size of the BDD dramatically. Observe that the commands `flatten_hierarchy` and `static_order` must be invoked before the dynamic ordering command: `dynamic_var_order`. Available methods for dynamic reordering are *window* and *sift*. We adopted the tools default reordering mode:

```
vis> dynamic_var_ordering
Dynamic variable ordering is enabled with method sift.
vis> print_bdd_stats
```

After the dynamic variable ordering is completed the BDD node count is given by the command `print_bdd_stats`. In this case the dynamic ordering did not found better ordering of the variables, and thus the BDD node count (and the size estimate) for the GCD circuit is 768.

4 Formal Area Complexity Modeling

In this section, we model the GCD benchmark circuit using the Action Systems formalism and then evaluate its area complexity as defined in Sect.2. The area complexity is then compared with the result given by the VIS tool. Finally, we present and compare other area complexity results from several other benchmark circuits.

4.1 Modeling the area complexity of the GCD-system

The greatest common divisor system operates in parallel with its environment: $\mathcal{G}CD \parallel \mathcal{E}nv$, where $\mathcal{G}CD$ describes the greatest common divisor system and the system $\mathcal{E}nv$ is its environment. The environment provides the necessary stimulus for the $\mathcal{G}CD$ system, but we are only interested in the area complexity of the $\mathcal{G}CD$ system, and therefore the environment is excluded from the system modeling and the area complexity estimation. The system $\mathcal{G}CD$ is defined on Page 21, where the $enable_{gcd}$ signal is set *true* by the environment to start the calculation. The input variables u and v are non-negative integers. The output variable gcd is of type integer and returns the greatest common divisor between u and v. The actions $G1$ and $G2$ calculates the trivial cases when one of the input variables is zero, and therefore the greatest common divisor is the non-zero input. The actions store the GCD value into the variable gcd and then they disable the calculation by setting $enable_{gcd}$ to *false*. The actions $G3 - G19$ compute GCD in those cases when both of the inputs are non-zero. The action $G3$ checks whether both of the input variables are *even*, *odd*, or another is *even* and another is *odd*. This is accomplished by calling the procedure *parity*, which receives the input variables u and v as input parameters. The parity is decided using the *modulo* operator (mod). The input variable is *even* if the *input variable* $mod\ 2 = 0$, and *odd* if the *input variable* $mod\ 2 \neq 0$. According to the parity check the GCD calculation is carried by the actions $G4$ - $G8$. The GCD calculation is completed when u equals with v ($u = v \wedge u \neq 0 \wedge v \neq 0$), which is defined by the action $G9$. If both u and v have been *even* at the same time during the GCD computation the result is multiplied, which follows the algorithm presented in A.1.4. The variable $mcoef$ is multiplied by two (initially $mcoef = 1$) every time the action $G4$ is executed. Once the result of the calculation is stored into the output variable gcd, the action $A9$ sets the parity check variable $pcheck$ to $true$ and the $enable_{gcd}$ to *false* indicating to the environment that a new calculation may begin. Furthermore,

the multiplication variable $mcoef$ is set to its initial value.

sys $\mathcal{GCD}$ (**in** $u, v : integer$; **out** $gcd : integer$;
inout $enable_{gcd} : Bool$;) ::
|[**type**
$status = \{even, odd\}$;
variable
$pcheck : Bool$;
$mcoef : integer$;
$stat_u, stat_v : status$;
private procedure
$parity$: (**in** $x, y : integer$ **out** $stat_x, stat_y : status$)
$(stat_x := stat'_x.(x \textbf{ mod } 2 = 0 \Rightarrow stat'_x = even)$
$\wedge (x \textbf{ mod } 2 \neq 0 \Rightarrow stat'_y = odd)$;
$(stat_y := stat'_y.(y \textbf{ mod } 2 = 0 \Rightarrow stat'_y = even$
$\wedge (y \textbf{ mod } 2 \neq 0 \Rightarrow stat'_y = odd)$;
$mult$(**out** $z : integer$) : $(z := 2 * z)$;
div : (**in** $x, y : integer$ **out** $z : integer$) : $(z := z'.(z' = x/y))$;
sub : (**in** $x, y : integer$ **out** $z : integer$) : $(z := z'.(z' = x - y)$;
action
$G1 : enable_{gcd} \wedge u = 0 \wedge v \neq 0 \rightarrow gcd := v; enable_{gcd} := F$;
$G2 : enable_{gcd} \wedge u \neq 0 \wedge v = 0 \rightarrow gcd := u, enable_{gcd} := F$;
$G3 : enable_{gcd} \wedge pcheck \wedge u \neq v \rightarrow$
$parity(u, v, stat_u, stat_u); pcheck := F$;
$G4 : \neg pcheck \wedge stat_u = even \wedge stat_v = even \rightarrow$
$div(u, 2, u); div(v, 2, v); mult(mcoef); pcheck := T$;
$G5 : \neg pcheck \wedge stat_u = even \wedge stat_v = odd \rightarrow$
$div(u, 2, u); v := v); pcheck := T$;
$G6 : \neg pcheck \wedge stat_u = odd \wedge stat_v = even \rightarrow$
$u := u); div(v, 2, v); pcheck := T$;
$G7 : \neg pcheck \wedge stat_u = odd \wedge stat_v = odd \wedge u \geq v \rightarrow$
$sub(u, v, u); div(u, 2, u); v := v); pcheck := T$;
$G8 : \neg pcheck \wedge stat_u = odd \wedge stat_v = odd \wedge u < v \rightarrow$
$u := u), sub(v, u, v); div(v, 2, v); pcheck := T$;
$G9 : u = v \wedge \neg pcheck \rightarrow gcd := gcd'.(gcd' = mcoef * u)$,
$enable_{gcd}, pcheck, mcoef := F, T, 1$;
initialisation
$enable_{gcd}, pcheck := F, T$;
$mcoef := 1$;
execution
forever do $G1$ [] $G2$ [] $G3$ [] $G4$ [] $G5$ [] $G6$ [] $G7$ [] $G8$ [] $G9$ **od**]|

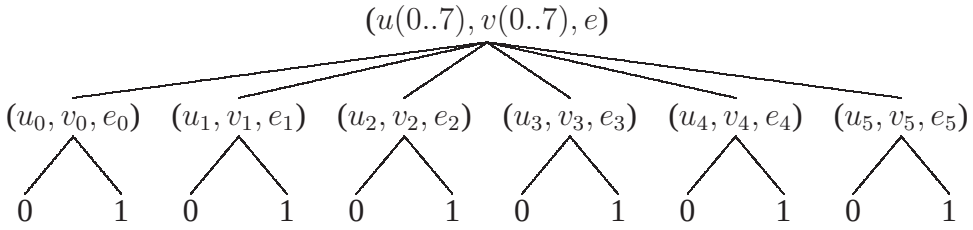


Figure 7: Example area complexity illustration for action G1 ($enable_{gcd} = e$).

4.1.1 Area complexity modeling of the $\mathcal{G}CD$ system

In the modeling of the area complexity we assume that the variable widths of Boolean ($Bool$) variables is one (1) and the integer variables width is eight ($n = 8$). The area complexity for the system $\mathcal{G}CD$ is estimated by calculating the area complexity for each action and procedure separately. As an example, consider the action $G1$. Its write variables is a set $wG1 \mathrel{\widehat{=}}$
$\{gcd(8), enable_{gcd}(1)\}$ and its read variables is a set $rG1 \mathrel{\widehat{=}} \{enable_{gcd}(1), u(8), v(8)\}$, where the number inside parenthesis denote the width of the particular variable. The area complexity estimation of the write set, as defined in 1: $C(wG1) = w_{gcd} + w_{enable_{gcd}} = 8 + 1 = 9$. The area complexity estimation of the read set is based on (2), and it is defined by: $C(rG1) = \lceil \frac{C(rG1)}{|rG1|} \rceil \cdot 2^{|rG1|} = \lceil \frac{17}{3} \rceil \cdot 2^3 = 48$. The area complexity of the read set is illustrated using a tree model, shown in Fig. 7, which describes the structure of the logic needed to implement the guard. The area complexity of the action $G1$ is then calculated by adding the complexities of the read set and write set together as defined in (4): $C(G1) = C(wG1) + C(rG1) = 57$.

Let us consider another action, say $G9$, whose write variables is a set: $wG9 \mathrel{\widehat{=}} \{gcd(8), enable_{gcd}(1), pcheck(1), mcoef(8)\}$, and the area complexity for the set is calculated as presented above: $C(wG9) = 18$. The set of read variables of the action $G9$ are divided into two sets: the variables that are used in the guard and the variables that are used in the predicate: $rG9 \mathrel{\widehat{=}} rg \cup rQ$. The set rg is $rg \mathrel{\widehat{=}} \{u(8), v(8), pcheck(1)\}$ and the set rQ is $rQ \mathrel{\widehat{=}} \{u(8), mcoef(8)\}$. The area complexity for the set rg is $C(rg) = \lceil \frac{17}{3} \rceil \cdot 2^3 = 48$, and its tree model is shown in Fig. 8. Next, we calculate the area complexity for the set rQ. The predicate Q performs a multiplication, and therefore, its area complexity is calculated by: $C(rQ) = \lceil \frac{C(rQ}{|rQ|} \rceil \cdot (2^{|rQ|})^{\phi} = \lceil \frac{16}{2} \rceil \cdot (2^2)^2 = 128$, where ϕ is the complexity coefficient defined in Sect. 2.1.1. The tree model is shown in Fig. 9. To summarize, the area complexity of the action $G9$ is $C(G9) = C(wG9) + C(rg) + C(rQ) = 194$.

The read and write sets and the area complexity calculation for every action and procedure in the system are summarized in Table 3, where the subtraction and the division operations are defined using separate procedures. This highlights the fact that it would be too expensive to implement own division and subtraction units

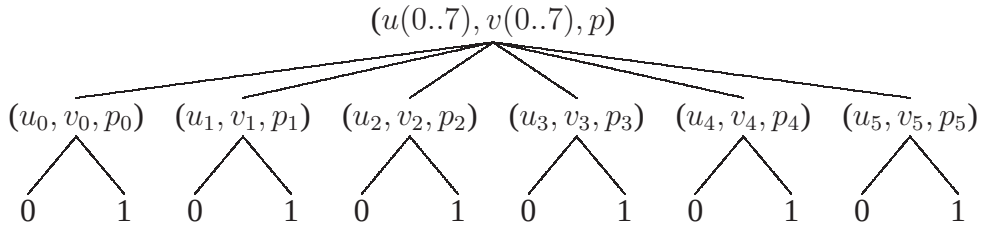


Figure 8: Example area complexity illustration for action G9 (guard, *pcheck* = *p*).

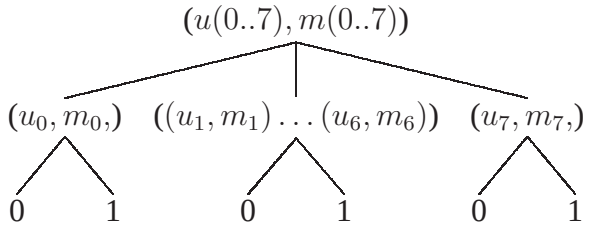


Figure 9: Example area complexity illustration for action G9 (predicate, *mcoef* = *m*).

for each action. In other words there are only one division unit and one subtraction unit in the system. Furthermore, none of these actions / procedures do not acquire the division simultaneously. The multiplication unit is implemented inside the action $A9$ because it is the only action that uses it. The total area complexity of the system $\mathcal{G}CD$ is: $C(\mathcal{G}CD) \mathrel{\widehat{=}} C(G1) + \cdots + C(G11) + C(parity) + C(div) + C(sub))$, where the $C(G1) \ldots C(G11)$ are the area complexities of the actions in the system. The area complexity of the system $\mathcal{G}CD$ is: $C(\mathcal{G}CD) = 967$. Comparing this result with the BDD-result from the VIS tool, we see that our model gave slightly smaller area complexity estimate. The accuracy of our model in this case study is $79\ \%$.

4.2 Results summary

The area complexity analysis was carried out for several benchmark circuits. The system descriptions of the functionality and the formal specifications are defined in Appendix A. The BLIF/BLIF-MV benchmark circuits were analyzed in a similar manner as presented in Sect. 3. The formal area complexity analysis is done as shown in the previous section. The variable widths are assumed to be one (1) for Boolean variable ($Bool$) and for *integer* variables we use the same width as is used in the benchmark circuits. Furthermore, generic variable types such as *data* used, for instance, in *full-adder* on page 33 are turned either into Boolean variables or some width is assigned to those variables. The results are summarized in Table 4, where the first column describes the benchmark circuit, the second col-

Table 3: Results of the area complexity estimation for $\mathcal{G}CD$.

Action	Write variables	Read variables	Area complexity
$G1$	$gcd(8)$, $enable_{gcd}(1)$	$enable_{gcd}$, $u(8)$, $v(8)$	$9 + \lceil \frac{17}{3} \rceil \cdot 2^3 = 57$
$G2$	$gcd(8)$, $enable_{gcd}(1)$	$enable_{gcd}$, $u(8)$, $v(8)$	$9 + \lceil \frac{17}{3} \rceil \cdot 2^3 = 57$
$G3$	$pcheck(1)$	$enable_{gcd}(1)$, $u(8)$, $v(8), pcheck(1)$	$2 + \lceil \frac{18}{4} \rceil \cdot 2^4 = 42$
$G4$	$pcheck(1)$	$stat_u(1)$, $stat_v(1)$, $pcheck(1)$	$1 + \lceil \frac{3}{3} \rceil \cdot 2^3 = 9$
$G5$	$v(8)$, $pcheck(1)$	$stat_u(1)$, $stat_v(1)$,$pcheck(1)$	$9 + \lceil \frac{3}{3} \rceil \cdot 2^3 = 17$
$G6$	$u(8)$, $pcheck(1)$	$stat_u(1)$, $stat_v(1)$, $pcheck(1)$	$9 + \lceil \frac{3}{3} \rceil \cdot 2^3 = 17$
$G7$	$v(8)$, $pcheck(1)$	$stat_u(1)$, $stat_v(1)$, $pcheck(1)$, $u(8)$, $v(8)$	$9 + \lceil \frac{19}{5} \rceil \cdot 2^5 = 137$
$G8$	$u(8)$, $pcheck(1)$	$stat_u(1)$, $stat_v(1)$, $pcheck(1)$, $u(8)$, $v(8)$	$9 + \lceil \frac{19}{5} \rceil \cdot 2^5 = 137$
$G9$	$gcd(8)$, $pcheck(1)$, $enable_{gcd}(1)$ $mcoef(8)$	$u(8)$, $v(8)$, $pcheck(1)$,$mcoef(8)$	$18 + \lceil \frac{17}{3} \rceil \cdot 2^3 + \lceil \frac{16}{2} \rceil$ $\cdot(2^2)^2 = 194$
$parity$	$stat_u(1)$, $stat_v(1)$	$x(8)$, $y(8)$	$2 + \lceil \frac{16}{2} \rceil \cdot (22)2 = 130$
div	$z(8)$	$x(8)$, $y(8)$	$8 + \lceil \frac{16}{2} \rceil \cdot (22)2 = 136$
sub	$z(8)$	$x(8)$, $y(8)$	$8 + \lceil \frac{16}{2} \rceil \cdot 22 = 40$

umn an area estimate from VIS-tool, the third column the area complexity of the action systems descriptions and the last one the accuracy of the presented model with respect to the VIS results.

The accuracy of the presented formal area complexity model varies from 56 % to 99 % compared to the BDD estimate given by the VIS tool (average 72 %). The 8-bit ALU gave the worst accuracy for our area complexity model. However, this benchmark circuit is also the one with the most open questions. It reserves some amount of space for the extra control logic and we do not know how well it is actually described in the original code. Therefore, it is fair to assume that the definition of the system might differ most between the original benchmark circuit and the action system $\mathcal{A}LU8$.

The most common level of accuracy was in between 60 % to 70 %, which is acceptable since the formal system definitions are at a higher abstraction level whereas the Benchmark circuit operates at lower (Boolean) level. Furthermore, it is expected that accuracy increases as the abstraction level decreases.

In addition to the above presented results, we compared the full-adder structure using different bit widths. The results are shown in Fig. 10, where the circles

Table 4: Results.

Design	VIS-2.1	Action model	%
16-bit multiplier	198	272	72
32-bit multiplier	374	544	69
16-bit adder	48	72	67
32-bit adder	96	160	60
8-bit alu + control	447	250	56
Arbiter with 3 clients	323	327	99
Greatest common divisor	768	967	79

represent the full-adder areas from the VIS tool and the boxes are the areas from the formal model. The accuracy in the addition operation is $60\ \%$ for variable widths 4,8,32,64. For 16-bit adder the accuracy is $67\ \%$, and therefore the average accuracy of the area complexity of the addition operation is $61\ \%$.

Considering all of the results presented in this chapter in a formal point of view, we see that the majority of the area complexity estimates were larger than the corresponding BBD estimate. This benefits, for instance, the refinement [3] of systems in terms of area [32]. That is, the result indicates that during system development the area limit is rarely exceeded.

5 Conclusion

In this paper, we presented and compared formal area complexity model targeted to abstract system specifications. The model adopts properties which are used on the size estimation of Boolean functions. The accuracy of the model was analyzed using various benchmark circuits and a high-level (above RTL) synthesis tool. We used the VIS tool developed in the Berkeley University for high-level (Boolean) synthesis and verification. One of the criteria was that there exists freely available benchmark circuits written the correct input language for the tool. The operation of the benchmark circuits were then written using Action Systems formalism. The goal with all benchmark circuits were to increase the level of abstraction while preserving the similar functionality as the original benchmark circuit.

The size of the benchmark circuits were analyzed using BDDs and then compared with the formal definitions. The accuracy of the model varied most commonly between 60% and 70% from the size of the benchmark circuits. Observe that in the most cases the formal model gave larger result than the BDD one. Therefore, it is fair to assume that as the level of abstraction decreases the model the performance analysis gets more and more accurate because the system descriptions will be closer to Boolean descriptions. At certain phase we might be able to use the actual BDD instead of the presented area complexity model for the

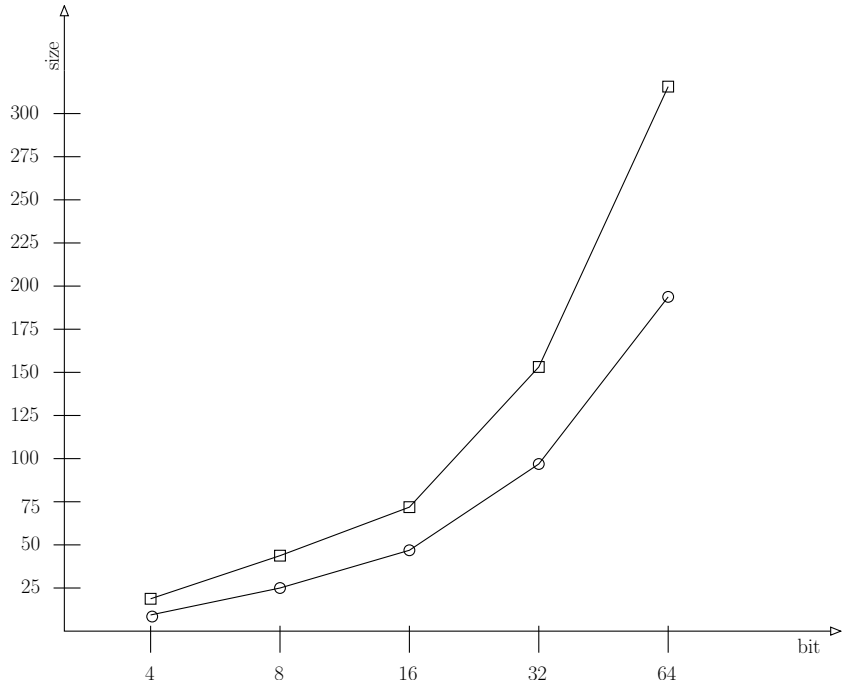


Figure 10: Size of full-adder with different bit widths.

estimation. Thus, the benefit of this is that during system development, where the accuracy of the system descriptions increases, the area complexity limit set for the system is rarely exceeded.

References

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

[2] R. J. Back and K. Sere. *From Modular Systems to Action Systems*, in Proc. of Formal Methods Europe'94, LNCS.

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

[4] A. Benveniste and P. Le Guernie. *Hybrid Dynamical Systems Theory and the SIGNAL Language*, IEEE Transactions on Automatic Control, 35(5):535-546, 1990.

[5] G. Berry and L. Cosserat. *The ESTEREL Synchronous Programming Language and Its Mathematical Semantics*, In Seminar on Concurrency, volume 1197,, pp. 389-448, LNCS, 1985.

[6] G. Berry and E. Sentovich. *Multiclock Esterel*, In Correct Hardware Design and Verification Methods, vol. 2144, pp. 110-125 LNCS, 2001.

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

[8] 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.

[9] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. *LUSTRE: A Declarative Language for Programming Synchronous Systems*, In Proc. of 14th Symposium on Principles of Programming Languages (POPL'87), pp. 178-188, 1987.

[10] D. M. Chapiro. *Globally Asynchronous Locally Synchronous Systems*, PhD thesis, Standford University, 1984.

[11] K. T. Cheng and V. Agrawal. *An Entropy Measure for the Complexity of Boolean Functions*, in Proc. of DAC, pp. 302-305, 1990.

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

[13] R. W. Cook and M. J. Flynn. *Logical Network Cost and Entropy*, in IEEE Transaction on Computers, vol. 22, no. 9, pp. 823-826, 1973.

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

[15] G. Dubost, S. Granier and G. Berry. *An Esterel-Based Formal Specification Methodology for Power manager Development*, in SAME2007 forum, October 4, 2007.

[16] M. C. Hansen, H. Yalcin and J. P. Hayes. *Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering*, in IEEE Design and Test of Computers, 1999, pp. 72-80.

[17] E. Kellerman. 'A Formula for Logical Network Costs, IEEE Transactions on Computers, vol. 17, no.9, pp. 881-884, 1968.

[18] 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. 32, no. 12, pp. 68-75.

[19] J. Lind-Nielsen. *BuDDy BDD package version 2.4*, http://sourceforge.net/projects/buddy/.

[20] M. R. Mousavi, P. L. Guernic, J. P. Talpin, S. K. Shukla and T. Basten. *Modeling and Validating Globally Asynchronous Design in Synchronous Frameworks*, In Proc. of the Conference on Design Automation and Test in Europe (DATE'04), pp. 384-389, IEEE Computer Society Press.

[21] D. E. Muller. *Complexity in Electronic Switching Circuits*, IRE Transactions on Electronic Computers, vol. 5, pp.15-19, 1956.

[22] M. Nemani and F. n. Najm. *High-Level Power Estimation and the Area Complexity of Boolean Functions*, in Proc. of ISLPED, 1996, pp. 329-334.

[23] J. Plosila. *Self-Timed Circuit Design - the Action Systems Approach*, PhD thesis, University of Turku, 1999.

[24] J. Plosila, P. Liljeberg and J. Isoaho. *Modelling and Refinement of an On-Chip Communication Architecture*, In Formal Methods and Software Engineering: 7th International Conference on Formal Engineering Methods, pages 219234, 2005

[25] N. Pippenger. *information Theory and the Complexity of Boolean Functions*, Mathematical System Theory, vol. 10, pp. 129-167, 1977.

[26] B. Rajan and R: Shyamasundar. *Multiclock Esterel: a Reactive Framework for Asynchronous Design*, in Parallel and Distributed Processing Symposium, pp. 201209, 2000.

[27] C. E. Shannon. *The Synthesis of Two-Terminal Switching Circuits*, Bell System Technical Journal, vol. 28. no.1, pp. 59-98, 1949.

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

[29] 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.

[30] J. P. Talpin, P. Le Guernic, S. Shukla, R. Gupta and F. Doucet. *Polychrony for Formal Refinement-Checking in a System Level Design Methodology*, in Proc. of the 3rd IEEE International Conference on Application of Concurrency to System Design (ACSD'2003), pp. 9-19, 2003.

[31] 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.

[32] J. Tuominen, T. Westerlund and J. Plosila. *Power Aware System Refinement*, in Proc. of International Refinement Workshop (REFINE'07), Oxford, England, in Electronic Notes in Theoretical Computer Science, vol 201C, pp.223-253.

[33] J. Tuominen, T. Westerlund and J. Plosila. *Formal Power Analysis of On-Chip Communication*, in Proc. of Brazilian Symposium on Formal Methods (SBMF 2007), August 29-31, Ouro Preto, Brazil, pp. 87-102.

[34] T. Westerlund. *Time Aware Modeling and Analysis of Systems-on-Chip*. PhD thesis, University of Turku, 2007.

[35] *ACM/SIGMA 1985-1995 benchmark sets.*
http://www.cbl.ncsu.edu/benchmarks/.

[36] *MVSIS: Logic Synthesis and Verification*
http://embedded.eecs.berkeley.edu/Respep/Research/mvsis/

[37] *CUDD: CU Decision Diagram Package*
http://vlsi.colorado.edu/ fabio/CUDD/

[38] *BuDDy Decision Diagram Package*
http://sourceforge.net/projects/buddy/

[39] *Verification Interacting with Synthesis (VIS)*
http://embedded.eecs.berkeley.edu/research/vis/

[40] *Esterel EDA Technologies*
http://esterel-eda.com/

[41] *Esterel Verification Environment (XEVE)*
http://www-sop.inria.fr/meije/verification/Xeve/

A Models for Area Complexity Estimation

This Section describes the models used in this study. First, we give an informal description of the benchmark systems. Then, we give the formal definitions for those systems that are not yet defined in this paper. The reference models for the formal area complexity model are either from *ACM/SIGMA* [35] benchmark set or from *ISCAS'85* [16] benchmark set. There is one exception the adder circuit is from *BuDDY* [19] BDD package, which consist on ready made C++ code to generate n-bit full adder circuit where n is any positive integer. Observe that in BDD generation we used same ordering methods between BuDDy and VIS in order to keep the results consistent. Second, we give the formal definitions for those systems that are not yet defined in this paper

A.1 Model descriptions

A.1.1 Full-adder and multiplier

To evaluate the accuracy of our area complexity model in arithmetic circuits we selected a basic full adder circuit and a basic multiplication unit. The adder circuit was simulated with the following bit widths: 4, 8, 16, 32, 64. The multiplier unit were analyzed using 16-bit and 32-bit word lengths.

A.1.2 8-bit ALU and control logic

A high level model [16] of the ALU benchmark is shown in Fig. 11. The core of the circuit is an 8-bit adder, which consist of two 4-bit carry look ahead adders (modules *M4* and *M5*) and the CLA generator circuit, which produces generate, propagate and sum signals (module *M3*). Multiplexers (modules *M1* and *M6*) select the incoming and outgoing data buses. The control unit (module *M2*) controls both multiplexers such that only one function is activated at time. Furthermore, the control unit contains extra control logic, which can be used to generate additional control logic, for instance, for devices next to the ALU.

A.1.3 Arbiter

The arbiter is a simple circuit where three modules (clients) are competing to get a bus access. Each client has a controller attached to it from which the acknowledgement is given. The controllers communicate with the arbiter in a way that at any time at most one controller gives an acknowledgment. The arbiter is a simple three-state machine. It has a single output indicating which controller can be selected among the three. If the "active" input of the controller is 0, the output is X, meaning that no one is selected.

The protocol here is that a controller takes a control from the arbiter if it is selected by the arbiter and it has a request. Otherwise, the control is passed to

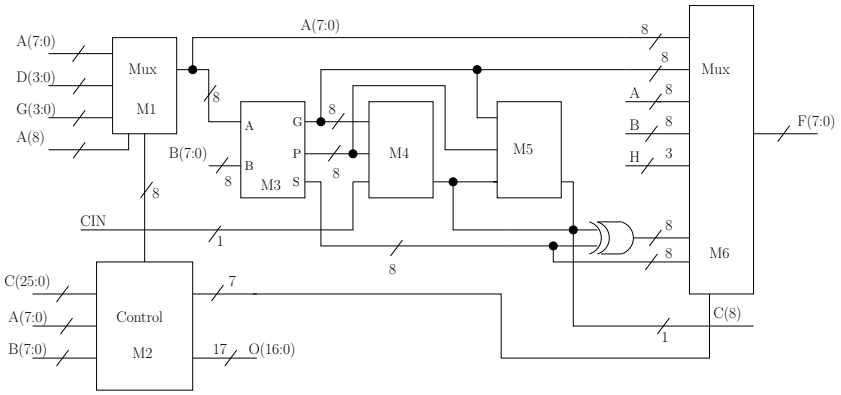


Figure 11: 8-bit ALU with additional control logic.

the next controller. A signal $pass_{ctrl}$ is set to one if the controller needs to pass a control to another controller. There are two cases: 1) when the controller is done processing a request, 2) when the controller has no request waiting, but is selected. In both cases, the variable is set to one in the next clock so that another controller waiting for an access can take a control. The "active" signal of the arbiter is set to one if one of the $pass_{ctrl}$ is set to one.

A client has to keep a request signal high until an acknowledgment is given. Even after an acknowledgment is returned from the corresponding controller, req can be high for a finite amount of time. This means that different requests take different time to complete. Fairness constraints arise here since we do not want to keep req high for infinite time. Simple block diagram of the arbiter is shown in Fig. 5.

A.1.4 Greatest common divisor

The binary greatest common divisor (GCD) algorithm is an algorithm which computes the greatest common divisor of two non-negative integers. It gains a measure of efficiency over the ancient Euclidean algorithm by replacing divisions and multiplications with shifts, which are cheaper when operating on the binary representation used by modern computers. The algorithm reduces the problem of finding GCD by repeatedly applying the identities $1 - 5$ shown below. The u and v are non-negative integers.

1. The greatest common divisor between 0 and v is $gcd(0, v) = v$, because everything divides zero, and v is the largest number that divides v. Similarly, $gcd(u, 0) = u$ and $gcd(0, 0)$ is not defined.

2. If u and v are both even, then $gcd(u, v) = 2 \cdot gcd(u/2, v/2)$, because 2 is a common divisor.

3. If u is even and v is odd, then $gcd(u, v) = gcd(u/2, v)$, because 2 is not a common divisor. Similarly, if u is odd and v is even, then $gcd(u, v) = gcd(u, v/2)$.

4. If u and v are both odd, and $u \geq v$, then $gcd(u, v) = gcd((u - v)/2, v)$. If both are odd and $u < v$, then $gcd(u, v) = gcd((v - u)/2, u)$.

5. Repeat steps 3 and 4 until $u = v$, or until $u = 0$.

A.2 Action System descriptions

The action system description are written using the information given in the subsection 2.2.3 as a guideline. The purpose is to keep the functionality of the systems as similar as possible whereas the syntactical aspects may differ. All formal system descriptions are assumed to work in parallel with its environment system $(\mathcal{S}ys \parallel \mathcal{E}nv)$. The purpose of the environment system is to provide necessary stimulus for the actual system but its internal specifications are left intact. Furthermore, it is not part of the area complexity estimation presented in Section 2 and therefore it is not modeled.

A.2.1 Full-adder and multiplier

An addition operation is described by the system $\mathcal{A}$ and it is defined by:

sys $\mathcal{A}$ (op_{add};) ::
|[
variable
$d1, d2, r : data;$
action
$Add : op_{add} \rightarrow r := r'.(r' = d1 + d2);$
initialisation
$d1, d2, r := d10, d20, r0;$
execution
forever do Add **od**
]|

where the environment sets the variable op_{add} to *true* whenever addition is requested. Observe that when the variable op_{add} is set to *false* the system $\mathcal{A}$ is in idle state waiting another addition request from its environment. The action Add performs the addition of the two input variables $d1$ and $d2$ and stores the result to the output variable r. These variables are of type $data$ and its variable length is n ($n \in \mathbb{N}^+$)

In a similar manner we define the multiplication operation $\mathcal{M}ult$, we have:

Table 5: Results for adder and multiplication.

Action	Write variables	Read variables	Area complexity
$Add16$	$\{r(16)\}$	$\{d1(16), d2(16)\}$	$C(Add16) = 16 + \lceil \frac{32}{2} \rceil \cdot (2^2)^1 = 80$
$Add32$	$\{r(32)\}$	$\{d1(32), d2(32)\}$	$C(Add32) = 32 + \lceil \frac{64}{2} \rceil \cdot (2^2)^1 = 160$
$Mult16$	$\{r(16)\}$	$\{d1(16), d2(16)\}$	$C(Mult16) = 16 + \lceil \frac{32}{2} \rceil \cdot (2^2)^2 = 272$
$Mult32$	$\{r(32)\}$	$\{d1(32), d2(32)\}$	$C(Mult32) = 32 + \lceil \frac{64}{2} \rceil \cdot (2^2)^2 = 544$

sys $\mathcal{M}ult$ (op_{mult};) ::
|[
variable
$d1, d2, r : data;$
action
$Mult : op_{mult} \rightarrow r := r'.(r' = d1 * d2);$
initialisation
$d1, d2, r := d10, d20, r0;$
execution
forever do $Mult$ **od**
]|

where the environment enables the multiplication by setting the variable op_{mult} to *true* (when set to *false* the system is disabled). Once the multiplication is requested the action $Mult$ is executed and the result of the multiplication is stored to the output variable r. The variables are of type *data* which length is n ($n \in \mathbb{N}^+$).

The area complexity calculations for 16-bit and 32-bit adder and 16-bit and 32-bit multiplier are summarized in Table 5.

The first column describes the action under evaluation. The second and third column describes the set of write variables and the set read variables for each action, respectively. Observe that variable widths are denoted inside the parenthesis. Naturally the widths are similar with the original benchmark circuit. The fourth column shows the area complexity calculation.

A.2.2 8-bit ALU and control logic

The implementation of the 8-bit ALU and control logic is defined by:

sys $\mathcal{A}lu8$ (**in** $a, b : data8; d, g : data4; select : instruction;$ **out** $f : data8;$) ::

|[

type

$instruction : \{add8, add4, idle, other\};$

variable

$tmp_a, tmp_b : data8;$

$tmp_{co} : bool;$

$select : instruction;$

private procedure

$Control($**in** $c : instruction25;$ **out** $o : instruction17) :$ *control logic*;

$Add($**in** $x, y : data8;$ **out** $r : data8) : (r := r'.(r' = x + y));$

action

$A1 : select = add8 \rightarrow (tmp_a, tmp_b := a, b), enable_{add} := T;$

$A2 : select = add4 \rightarrow (tmp_a, tmp_b := d, g), enable_{add} := T;$

$A3 : enable_{add} \rightarrow Add(tmp_a, tmp_b, r);$

$f, enable_{add}, select := r, F, idle;$

$A4 : select = other \rightarrow Control(c, a, b); select := idle;$

initialisation

$tmp_a, tmp_b := tmp_a 0, tmp_b 0;$

$select := idle;$

execution

forever do $A1 \ [] \ A2 \ [] \ A3 \ [] \ A4$ **od**

]|

where the system consist of two procedures: the *add* procedure performs the 8-bit addition operation and the *control* describes the control logic that the benchmark implements. The operation of the *control* procedure is described simply with *control logic* because the type of the control logic is black box in the original circuit as well. That is, there is reserved space for some amount of extra logic. The control unit in the original benchmark controls the operation of the ALU. In this specification the ALU control is implemented outside the control unit. In all cases the variable widths match with the lengths in the benchmark circuit. The *select* signal operates as a communication path between the ALU and its environment. It has four possible states: *add8* requests 8-bit addition,*add4* requests 4-bit addition, *other* chooses to use the external control logic and the *idle* informs the environment that the ALU is ready accept new operation requests. The environment requests all operations from the ALU.

The read and write sets and the area complexities for each actions of the system $\mathcal{A}LU$ are shown in Table 6. The variable widths are denoted after inside parenthesis. Observe that even though the system description defines only four states to the *select* variable, which could be encoded using two-bit numbers, its width is 8. This follows from the benchmark circuit definitions. most likely there is left space to include more states in future for instance due to the external control logic.

Table 6: Read and Write sets and area complexities for system $\mathcal{A}LU8$.

Action	Write variables	Read variables	Area complexity
$A1$	$\{enable_{add}(1), tmp_a(1), tmp_b(1)\}$	$\{select(8)\}$	$C(A1) = 3 + \lceil \frac{8}{1} \rceil \cdot (2^1) = 19$
$A2$	$\{enable_{add}(1), tmp_a(1), tmp_b(1)\}$	$\{select(8)\}$	$C(A2) = 1 + \lceil \frac{8}{1} \rceil \cdot (2^1) = 19$
$A3$	$\{select(8), f(8), enable_{add}(1)\}$	$\{enable_{add}(1)\}$	$C(A3) = 17 + \lceil \frac{1}{1} \rceil \cdot (2^1) = 19$
$A4$	$\{select(8)\}$	$\{select(8)\}$	$C(A4) = 8 + \lceil \frac{8}{1} \rceil \cdot 2^1 = 24$
Add	$\{r(8)\}$	$\{x(8), y(8)\}$	$C(Add) = 8 + \lceil \frac{16}{2} \rceil \cdot 2^2 = 40$
$Control$	$\{o(17)\}$	$\{c(25), a(8), b(8)\}$	$C(Control) = 17 + \lceil \frac{41}{3} \rceil \cdot 2^3 = 129$

Turku
Centre *for*
Computer
Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-2136-1
ISSN 1239-1891