CAMAS-TR-2.2.4.2

Progress Report

COMMISSION OF THE EUROPEAN COMMUNITIES

****

ESPRIT III

PROJECT NB 6756

****

CAMAS

COMPUTER AIDED MIGRATION OF APPLICATIONS SYSTEM

****

CAMAS-TR-2.2.4.2

SAD/PARASOL II Progress report

****

Date : SEPT.1993
Rev. 2.0

ACE - U. of AMSTERDAM - ESI SA - ESI GmbH - FEGS - PARSYTEC -
U. of SOUTHAMPTON.

Progress Report SAD/PARASOL II (march 15 - sept. 14 1993)

By Jan de Ronde, Berry van Halderen, Marcel Beemster and Peter Sloot.
University of Amsterdam
September 1993

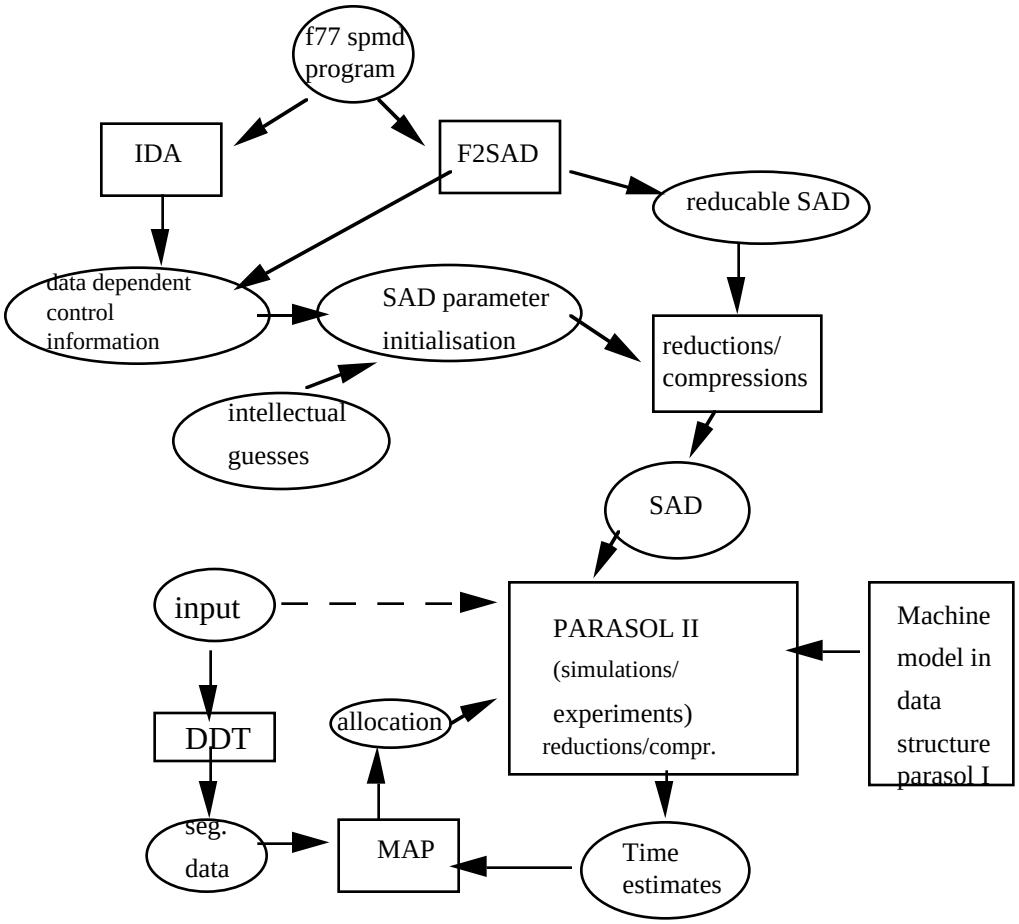
Contents:

1-Introduction
2-Development of the SAD formalism
2.1 - Research focus
2.2 - Data Dependent control information
2.3 - The reduction/compression phase
3 - PARASOL II: Simulations/Experiments
3.1 - The undeterminable X
4 - Implementation issues: F2SAD
4.1- How to get from Fortran to a SAD description
4.2- Structure of the front-end of the *f2sad* compiler
4.3- The semantics and back-end of the *f2sad* compiler
4.4- Status

Appendix :
-Abstracting the SAD formula of an F77 kernel by hand
-Additional information to the SAD formula: Data Dependent control
-Mathematical manipulations on the SAD formula

1. Introduction

The above mentioned period effort has been spent on the design and development of several tools (boxes in the overview picture below) that the University of Amsterdam contributes to the CAMAS Workbench.



2. Development of the SAD formalism.
The development of SAD (Symbolic Application Description) which is the mathematical formalism that describes the time complexity of sequential and SPMD Fortran programs has been one of the focal points this period. Firstly it was recognized that a mathematical formalism describing such programs in order to represent a time complexity description of the program, should consist of three separate levels:

We describe SPMD programs by the following functional hierarchy:

1) Statement block level (fixed complexities, machine dependent)
2) Control flow level (control dependent complexities)
3) Data locality level (data distribution dependent complexities)

2.1 Research focus
Investigations have been done on a formalism that can capture the first two levels in a description. Therefore several typical Fortran 77 programs/program parts have been analysed by hand. As the example in the appendix shows the first two levels of the functional hierarchy allow a generic description of the form:

$$SAD = \sum_{i=1}^{N} \prod_{m_i}^{M_i} \prod_{k_i}^{K_i} P_{k_i} X_{m_i} S[Block(i)]$$

Where N is the number of isolated statement blocks (containing no control flow characteristics whatsoever), P_{k_i} describes the k_i-th nested branch probability of the total of K_i branches in which *S[Block(i)]* is nested**.** Analogously X_{m_i} describes the loop-count of the m_i--th nested loop of a total of M_i loops in which *S[Block(i)]* is nested. *S[Block(i)]* is the time complexity of an isolated statement block labelled *i* .

1) The time complexity of a so called statement block $S[Block(i)]$ is given by cumulation of all the individual time complexities occurring in the block. For example an expression like:

```
a = a + b * c
```

has a time complexity of $T_{assign} + T_{multiply} + T_{addition}$

The corresponding actual time measures in such formal expressions are filled in by the machine database (the parameterised machine description).

2) The control flow level introduces indeterminability into the time complexity description. In general the execution path taken, given a specific set of input parameters, is only determinable by means of explicit execution. We approach this level in a (quasi) statistic manner by describing the possibility of branching in some specific direction along the execution graph. Branch directions in *if..then..else* constructs are specified by probabilities $P_1, P_2, P_3 \ldots . P_n$ (where we assume the number of branching directions = *n*) whereas in *loop c*onstructs the number of unkown iterations is presented by a stochastic variable X. So a parameterised description on these two levels leads to a mixture of the time-complexities of basic statement blocks, probabilities and stochastic variables.

3) The locality level describes the fact that some fraction of data is involved in communication and the remaining part is not. So far only the first two levels have been considered in the SAD formalism. How exactly the 3rd level will eventually appear in the time complexity formula is part of future work. This will among others depend on how the MAP subtask will evolve.

2.2 Data dependent control information

As is clear from the overview picture PARASOL II plays a key-role in the Workbench structure. What in fact happens in PARASOL II is actualisation of the parameters present in the SAD formula. The machine parameters are obtained from the machine database developed under Parasol I and have a clear origin (some existing or what-if hardware architecture that is). The other parameters originating from control flow and -when the 3rd level will be concerned- from data locality also need to be initialized in order to obtain a time complexity measure for the program of interest.The question of how to actualize these parameters is to be answered in PARASOL II.

Control dependency information will be substracted from the parse tree generated in the F2SAD transformation sequence. The ideas behind the implementation of such an automatic transformation tool will be discussed in one of the paragraphs below. Furthermore in the appendix an idea of what control dependencies will be substracted will be given. At this point obviously the Interprocedural Dependency Analyser of Southampton can be brought into the picture.

2.3 The reduction/compression phase

Given a general F77 numerical application. It can consist of thousands of lines of code and consequently performing F2SAD on it will lead to a formal expression that is out of proportion. Therefore in order to have an expression that is manageable it has to be reduced

in size by means of compression and reduction techniques. One can think of purely mathematical manipulations as the "Simplify" operation in Mathematica or reductions based on information obtained from data dependency analysis. For some examples of such reduction and compression operations see the appendix. The mathematical manipulation package Mathematica offers the possibility of performing reduction/compression techniques and several reductions/compressions can also automatically be performed within the F2SAD tool The reduction task consists of manipulating the SAD formula using control dependency information (and probably also expert programmers knowlegde) to do realistic actualisations of the parameter set that comprises the SAD formula. So the manipulations are reductions/compressions and actualisation of parameters. Both have to be done using some mathematical manipulation package. The solution to problems arising in actualizing the parameters are part of future research. The paragraph below will shed some light on the status of the simulation strategies as they have been developed until now.

3 PARASOL II: simulations/experiments

The goal of PARASOL II is : given the following:
- A reduced SAD description of a data parallel (SPMD) Fortran 77 application
- A Machine parameterization
- A Mapping tool which optimizes the data distribution

to obtain a prediction of the time consumption of the application given a spectrum of input values and machine parameters (PARASOL I). This would mean in the idealized case: Fill in the various input parameters to the SAD expression and the time complexity given the specific actualization rolls out of the black box. Due to the undeterminability of control flow this procedure will only work for generic applications when in fact the code is executed. Program execution is undesirable for a variety of reasons:

-Probably performance prediction on a "What-if machine", or "still-to-be-built "machine is desired so that execution is not possible.
-The time complexity response on "What-if" input may be of interest.
-Simulation allows for the exploration of different coding strategies for core codes without actually being forced to implement every strategy explicitly.

3.1 The undeterminable X

The SAD contains a number of specific parameters originating from the three hierarchical levels. Some of these parameters will depend trivially on input parameters (e.g. problem size = N) others will depend on the entire program evolution (e.g. the stop condition in a matrix inverting algorithm). The parameters with trivial dependendce are to be identified using control dependency analysis . The problematic ones are to be approximated using stochastic modelling. That is: based on characteristic features of a parameter (of importance) that can be supplied by the expert programmer or that can be derived from the source code (reversibility) the respons of such a parameter on variations $\delta\vec{I}$ of the input vector $\vec{I}$ can be approximated/modelled.

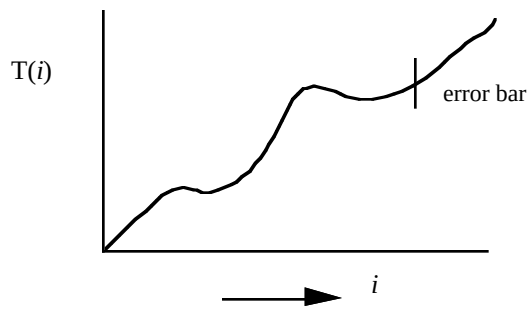
Notational: $\vec{I}$ represents the list of input parameters that is given as ordinary input to enable an application to execute. E.g. : the elements of a matrix and a vector when the calculation of a matrix vector product is concerned. Formally we shall denote the list with input values as:

$$\vec{I} = \{i_1, i_2, \ldots\ldots, i_n\}$$

In general the parameters X or P that describe loop and branching constructs in the SAD are functions of the input vector $\vec{I}$: X = f ($\vec{I}$). What is desired is an estimation of this function, as the only way to get the exact representation of X = f ($\vec{I}$) is execution of substantial parts of the application. Furthermore,some measure telling how probable this function tells the "truth" is convenient. E.g. for a specific X you can show that within a deviation of $\delta f(\vec{I})$ the used function is approximated. In other words:

$$X = f(\vec{I}) + \delta f(\vec{I})$$

The picture below gives an idea how this undeterminable behaviour would appear in the time complexity function when varying one of the input parameters*i*, the other paramters kept constant. The error bar characterizes the uncertainty for specific *i* that the time complexity formula T (= SAD formula) has at this point. These error bars are present for the whole spectrum of *i*.



So X is in general a function which depends on the input vector $\vec{I}$. Suppose we have a row of M functions

$$f_1 \circ f_2 \circ f_3 \circ \ldots\ldots f_M(\vec{I})$$ where $f \circ g$ means: the function f acting on the function g

that consecutively operate on $\vec{I}$. The complexity of these functions determines if you can calculate the response that these functions cause in a simple manner or that one has to flee to approximation methods. One can roughly discriminate several grades of complexity to obtain knowlegde about the dependence of a parameter X on the input list:

- X directly corresponds to some element *i* of $\vec{I}$: X = *i*
- X depends on $\vec{I}$ via a simple function and this function is tractable via some data dependency analysis.
- X depends on $\vec{I}$ via a function that is not simple but corresponds to the computation of significant parts of the application.

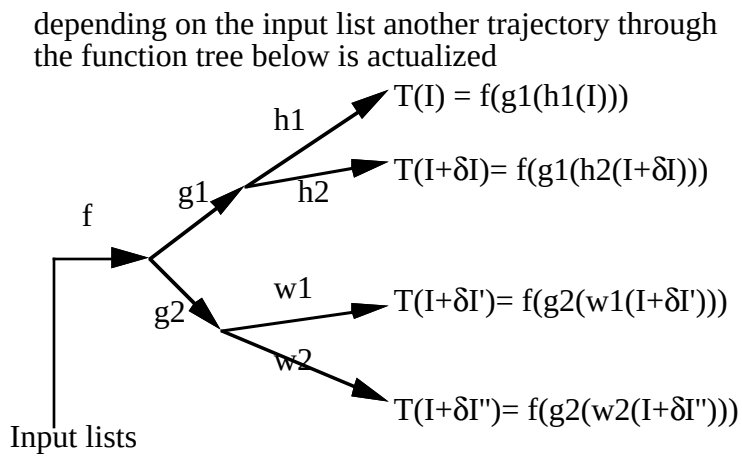
The problem with approximating the responses using some deviation function lies in the fact that in general the sequence of functions that act on the specific input list changes when a fluctuation $\delta\vec{I}$ is superposed on $\vec{I}$. So a continous variational solution method is not applicable to this problem (see picture below for elucidation). Formally one could write this down as:

$$i_m \rightarrow \mathrm{T}(i_m)$$
$$i_m + \delta i_m \rightarrow \mathrm{T}(i_m + \delta i_m)$$
$$\mathrm{T}(i_m + \delta i_m) - \mathrm{T}(i_m) = ?$$

where T denotes the time complexity formula and i_m denotes an input variable which is varied and is an element from the input list :

$$\vec{\mathrm{I}} = \{i_1, i_2, \ldots\ldots, i_n\}$$

The ? denotes the fact that a small variation in the input can lead to small variations in the response function and to large non-linear jumps. So it is clear that the problem of how to model/estimate response functions in this area is part of future research.



The picture above visualizes the non linear behaviour of the respons to variations of the input list. f, g1, g2, h1, h2, w1, w2 are functions.

4. Implementation issues : F77 to SAD

4.1 How to get from Fortran to a SAD description

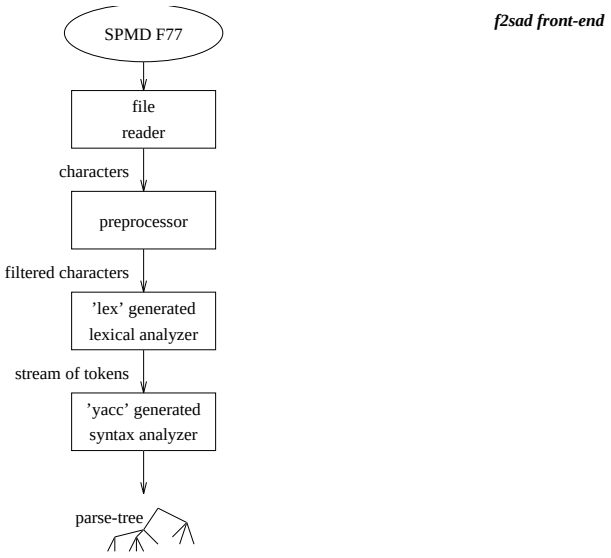
To obtain a SAD description of a Fortran program, a transformation of the source code has to be done. This translation process, implemented as a compiler, takes Fortran 77 source code augmented with communication primitives for the SPMD programming model as input and builds the SAD formula with the corresponding data dependent control flow and additional information about the program.

In the section below the general structure of such a translator is discussed. We then turn to the implications of trying to process Fortran. Finally we turn to the way the translation process can be implemented.

4.2 Structure of the front-end of the *f2sad* compiler

Fortran was the first high level programming language ever developed. Unfortunately, during the development of this early programming language a number of decisions were made which turned out to be bad practise in computer language design. Therefore the standard tools (lex & yacc) for compiler engineering are not directly applicative to Fortran 77.

There are several difficulties with Fortran which must be solved before standard tools can be used. These problems will be discussed in the software layer where they are solved.



The structure of the front-end of the *f2sad* compiler is structured as shown in the above figure, which is comparable to the generic model of a compiler [1]. The source program -- Fortran 77 code-- is read using a separate module. This module performs tasks like buffering, keeping track of the file position, line and column numbering. It also contains routines for returning errors back to the user.

This stream of characters is then filtered using a preprocessing module. Its output is a modified stream of characters where the following set of problems is solved.

- If a line starts with a label then it will be preceeded by a special indicating character. This will enable the lexical analyzer to distinguish these labels from integers, which in turn is needed in the syntactic analyzer.
  Labels after goto's and other jump statements are *not* preceeded by this indicator because the syntactic analyzer does not need to know this.
- All spacing characters except newlines are removed. Many programming languages have a free layout (spacing characters may be inserted freely between tokens), Fortran unfortunately takes this even one step further: spacing characters are insignificant. If you separate the characters within the keyword if with a space (making i f) the compiler should still recognize it as one keyword if instead of two variables i and f.
- Continuation lines are glued together to produce one line.
- Comment lines are deleted.
- Since Fortran officially only defines uppercase letters,the compiler is so friendly to convert lowercase letters to uppercase.
- Note that this preprocessor module breaks the direct correlation between the line numbers in the inputand the output. We want however to be able to go back from the generated SAD desciption to the source code, therefore this correlation needs to be restored. This is solved by the module which reads the file. The semantics desciption is able to refer to the file position of the tokens using a seperate information passing mechanism.

The lexical analyzer for Fortran is written using lex, a tool for generating lexical analyzers using regular expressions. Because the preprocessor has made some changes to the source input it is now possible to tokenize the input. The lexical analyzer not only converts characters into tokens, it also stores the lexical value (lexeme) for later use.

The following irregular behaviour patterns were necessary in the implementation of the lexical analyzer (using *lex*).

- If a label indicator is encountered in the input stream then the next token is a label instead of an integer. The label indicator character is further ignored.
- Within certain contexts, the context-free parser cannot distinguish an assignment to an array element from a function definition statement (this is discussed further on). Therefore the lexical analyzer indicates, on request of the parser, by inserting a special token whether this statement is an assignment or a function definition statement. It can determine this by looking if the identifier in the statement is defined. If the identifier is indeed defined to be an array then we are dealing with an assignment, otherwise with a function definition or an error.
- As said earlier; Fortran considers white spaces as insignificant. Suppose we have the input <u>if foo</u>. This should be tokenized into two tokens, the keyword <u>if</u> and the identifier <u>foo</u>. Unfortunately, because spaces are deleted and *lex* always tries to match the longest token it will consider <u>if foo</u> as one identifier <u>iffoo</u>. By checking prefixes of lexemes we can detect keywords, bypass the ``longest matching token'' rule and consider the remaining characters as part of the next token.

4.3 The semantics and back-end of the*f2sad* compiler

The *yacc* generated syntax checker, is normally used to make a parse tree. A parse tree represents the input source program in a structured way, in the same way the grammar is written down. Yacc does not automate the building of the parse tree, but allows you to execute actions on the nodes of the abstract parse tree. In this way you can build the parse tree yourself.

The translation process of the compiler can be viewed as the rewriting process of the parse tree into a tree representing the output. The output can then be generated by a traversal of the latter tree.

There are tools (e.g. [2]) available which assist in coupling the syntax checker and a parse tree manipulator, but most tools have the disadvantage of defining their own lexical and syntax analyzing languages. These restrict the classes of languages that can be handled and since Fortran parsing needs some special tuning of the lexical and syntax analyzers this presents a severe problem. It is possible to overcome these problems by creating an additional layer or by using a self-made tool.

The rewriting proces of the Fortran parse tree into the SAD formula is not extremely difficult, the only complication is that Fortran may contain goto statements, possibly creating so called spagetti code.

The data dependent control flow contains information by which the control flow can be described in terms of its input parameters. Generating this data dependent control flow information about the program is a more serious problem which still needs a lot of our attention.

Modelling and analyzing data dependent control flow interacts with analyzing data dependency. Therefore techniques used within the IDA (interprocedural dependency analyzer) tool may be very similar to techniques we need to use. Information generated by the IDA tool may enhance the data dependent control flow information. Future interaction between SOTON and the UoA therefore seems fruitful.

4.4 Status

The lexical- and syntax desciptions, the file reader and the preprocessor for Fortran are finished. The tool for building and rewriting the parse tree is nearing completion (in the testing phase). The description of the semantics of Fortran is currently being constructed. All finished parts are tested, but can only be tested thoroughly when the entire tool has been completed.

Bibliography

[1] Alfred V. Aho and Ravi Sethi and Jeffrey D. Ullman *Compilers, Principles, Techniques and Tools* Addison-Wesley 1986
[2] *The TXL programming language, Syntax and informal semantics, Version 7*

APPENDIX

Abstracting the SAD Formula from a Fortran 77 kernel by hand

In this appendix the SAD formalism describing the first two levels of the functional hierarchy of Fortran 77 (or C) will be derived using a scientific Fortran 77 kernel extracted from the so called Livermore Loops. This example will clarify the SAD expression that is chosen to describe the functional structure of Fortran 77 programs. Several F77 examples have been transformed to SAD by hand and these have shown the direction in which the F2SAD tool currently is being developed.

The Livermore Loop we're handling as an example of F2SAD by hand has the following form:

```
                PROGRAM SADTST
                PARAMETER (NN = 100, LOOP = 100)
                DIMENSION X(NN), V(NN)
BLOCK(1)        N = 1000
                DO 23 J = 1,  NN
BLOCK(2)           V(J) =  EXP ((REAL(J) - 50.)**2) / 5.0
          23    X(J) =  EXP ((REAL(J) - 50.)**2)
                DO 200 L = 1, LOOP
BLOCK(3)               II = N
                       IPTNP = 0
          222       IPNT = IPNTP
BLOCK(4)               IPNTP = IPTNP + II
                       II = II / 2
                       DO 2 K =  IPNT + 2,  IPNTP, 2
BLOCK(5)                     I = I  + 1
            2                X(I) = X(K) - V(K) * X(K-1) -
BLOCK(6)               IF( II. GT. 1) GO TO 222
          200 CONTINUE
                 END
```

The Livermore Loop under consideration descends from an ICCG (Incomplete Cholesky Conjugate Gradient). The basic statement blocks (level 1) are pointed out. Writing out the SAD expression by hand leads to:

$$SAD(program) = S[Block(1)] + X_1 S[Block(2)] + X_2\{S[Block(3)] + X_3\{S[Block(4)] + X_4\{S[Block(5)]\} + S[Block(6)]\}\}$$

or

$$SAD(program) = S[Block(1)] + X_1 S[Block(2)] + X_2 S[Block(3)] + X_2 X_3\{S[Block(4)] + S[Block(6)]\} + X_2 X_3 X_4 S[Block(5)]$$

Where *S[Block(i)]* denotes the cumulation of the atomic time complexities in the statement blocks. For example: $S[Block(1)] = T_{aic}$: That is the time to assign an integer constant.

The other statement blocks have the same form though of course in large statement blocks much more different atomic times are involved.
Further:

X_1 corresponds to the loop boundary of the DO 23 -- 23 loop
X_2 corresponds to the loop boundary of the DO 200 -- 200 loop
X_3 corresponds to the loop boundary of the IF(II.GT.1) GOTO 222 loop
X_4 corresponds to the loop boundary of the DO 2 -- 2 loop

We see that loop boundaries (originating from *do* loops or from *if..goto* loops) are parameterized in the SAD expression. Further investigations of generic Fortran 77 constructs have shown that an equivalent parameterization can be applied for handling the presence of if..then...elseif ....endif constructs in Fortran 77. The various branching directions offered in a branch construct are modelled by chances that describe the probability that a branch in a certain direction is realised. It turns out that an analogous nested expression arises as in the above expression. Furthermore it became clear that the probability and the loop boundary parameters could be handled as normal factors in a factorized expression. Therefore after having done several F2SAD actions by hand we can state that as long as the code is structured (no spaghetti) a symbolic expression for such programs can be realized and furthermore the general form for such a SAD expression is:

$$SAD = \sum_{i=1}^{N} \prod_{m_i}^{M_i} \prod_{k_i}^{K_i} P_{k_i} X_{m_i} S[Block(i)]$$

Where : P_{k_i} describes the k_i-th nested branch probability of the total of K_i branches in which *S[Block(i)]* is nested**.** Analogously X_{m_i} describes the boundary of the m_i-th nested loop of a total of M_i loops in which *S[Block(i)]* is nested.

Additional information to the SAD formula: Data Dependent control
As we consider the parameter set $\{X_1, X_2, X_3, X_4\}$ we notice the following:

X_1= NN ----> Input
X_2 =LOOP---> Input
X_3 =function of N:=f(N) ----> N :----->Input
X_4 =function of a function of N: = g(f(N))--> N:----->Input
So as was stated above: every parameter in the SAD expression depends on the input in a simple or less trivial manner.

Given a specific input set all parameters can exactly be calculated if the functions f and g are relatively simple. In the case of the example above some algebraic effort can determine the forms of the two functions f and g. As an example we'll derive the form for f:

From the program shown above we note that the loop boundary that X_3 describes depends on a function which we call F. This function has the following formwhen it acts on its argument II (double I not Pi).

$$F(II) = \left\lfloor \frac{II}{2} \right\rfloor$$

Where the floor function acting on argument a :$\lfloor a \rfloor$ means: return the truncation of the real value a . So $\lfloor 1.5 \rfloor = 1$, $\lfloor 2 \rfloor = 2$ etc.....

Further analysis by hand shows that the value of X_3 which depends on the initial value of II (= N) can be calculated using the condition:

$$\begin{cases} F^m(II) > 1 \\ F^{m+1}(II) \le 1 \end{cases} \quad \text{where } F^m = F^{m-1}(F) \text{ and m is integer}$$

The value of m that satisfies the above condition is m = $\lfloor {}^2\log(N) \rfloor$ = f(N).

Analagously such types of dependency analyses can be performed for all parameters that have no direct correspondence with an input parameter. Of course very often the back tracing of such control flow dependencies will turn out to be execution in reverse order of the program. That is wat we do not want. How to circumvent such problems is part of work to be done in PARASOL II.

Mathematical manipulations of the SAD formula

Now several possible reduction/compression techniques on the SAD formula will be discussed.

A Simplify action:

Generally the SAD expression for an arbitrary statement block S[Block] is a cumulation of all the atomic time consumptions in the block. What a Simplify action (such as offered by the mathematical manipulation package Mathematica) does is accumulating the times in the expression of the same name. Example:

$$S[Block] = T_1 + T_2 + T_1 + T_1$$
$$Simplify[S[Block]] \rightarrow 3T_1 + T_2$$

and for a general expression we can write

$$S[Block] = T_i + T_j + T_k + \ldots + T_l$$
$$Simplify[S[Block]] \rightarrow \sum_i \alpha_i T_i$$

where T_i, T_j, T_k etc.... denote the various atomic machineparameters .

On control level: Suppose you have an expression

$$P_1 * S[Block(i)] + P_2 * S[Block(j)]:$$

where

$$S[Block(i)] = \sum_i \alpha_i T_i$$
$$S[Block(j)] = \sum_j \beta_j T_j$$

and these statement blocks -although not related in any manner-turn out to satisfy the condition that the difference in their explicit time complexities is less than some small

value ε , then one can compress the total expression to a form in which only one statement block describing the time complexities of both original blocks remains:

$$\text{if:} \sum_j \beta_j T_j - \sum_i \alpha_i T_i << \varepsilon \Rightarrow (p_1 + p_2) S[\text{Block}(i, j)]$$

Another example of a possible reduction: given two statement blocks that are added together; contraction of the prefactors of the atomic times is another compression/reduction step:

$$S[\text{Block}(j)] + S[\text{Block}(i)] = \sum_i \beta_i T_i + \sum_i \alpha_i T_i \rightarrow \sum_i (\alpha_i + \beta_i) T_i$$

Or suppose the expression below arises:

$$X * S[\text{Block}(j)] + S[\text{Block}(i)]$$

When during simulations in PARASOL II X turns out not to be affected significantly by fluctuations $\delta\vec{I}$ of the input vector $\vec{I}$. Then the variable nature of X can be omitted and replaced by a constant. Consecutively one of the reduction techniques that are mentioned above can be used to reduce the expression further.