# Caesy: A Software Tool for Computer-Aided Engineering

Matt Wette
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109
mwette@csi.jpl.nasa.gov

## Abstract

A new software tool, Caesy, is described. This tool provides a strongly typed programming environment for research in the development of algorithms and software for computer-aided control system design. A description of the user language and its implementation as they currently stand are presented along with a description of work in progress and areas of future work.

## Introduction

Over the past few decades, control system design and analysis has become more and more dependent on computers. With the availability of more powerful hardware has come the demand for more performance. Computer-aided control system design tools such as Matlab have been used with some success in control system design since the early 1980's. However, many workers in the development of software and algorithms for control system design have recognized that these tools have limits in both flexibility and efficiency. The forces driving the development of new tools include the desire to make complex system modeling, design and analysis easier; the need for quicker turnaround time in analysis and design; the desire to make use of advanced computer architectures to help in control system design; the desire to adopt new methodologies in control; and the desire to integrate design processes (e.g., structure, control, optics). We have developed Caesy (Computer-Aided Engineering SYstem) as a means toward discovering how these desires can best be satisfied. The first Matlab-type environment for matrix manipulation, called MATLAB, was developed by Cleve Moler in the late 1970's, mostly at the University of New Mexico, with support from the National Science Foundation. Several other Matlab-based environments have been produced for control system design since then. Included in this group of control system design tools is Ctrl-C from Systems Control Technology, Matrix$_X$ from Integrated Systems, Inc. (ISI), Pro-Matlab from the MathWorks, Inc., Mat/C developed at Lawrence Livermore National Laboratory, and SFPACK developed at the University of Waterloo. In this document we will collectively refer to these MATLAB-based environments as Matlab. A new-generation Matlab package has been released from ISI. This package, called Xmath, seems to provide some features for easier use but does not offer all the features we feel are desirable. The success of Matlab as an environment for the design and application of algorithms for control system design implies that a new tool geared toward large order, complex problems should include capabilities provided by Matlab and attempt to maintain in some sense the "flavor" of Matlab.

In the development of Caesy, our goal has been to provide a more advanced environment which can provide the capabilities provided by these packages, but can also provide alternatives to better handle the problems encountered when problems become complex or the system order creates computational bottlenecks. More discussion on the requirements driving the development of Caesy can be found in [1]. Our approach to solving the complexity problem will be to develop an "object-oriented" user interface (e.g., typed variables and overloaded functions and operators). Our approach to the computational problems will be to take advantage of this interface to develop specialized software which can take advantage of any special structure in the models (e.g., symmetry, bandedness, sparseness).

In this paper we will provide an overview of the current capabilities and implementation of Caesy, the work currently in progress, and work planned for the future.

## Current Implementation of Caesy

In this section, we give an overview of Caesy and some of its features. Caesy is, in a concise description, a mix between Matlab and Ada. Caesy contains many of the constructs of Ada but adds features from Matlab and whatever other modifications we felt were needed to provide the required functionality. Ada was chosen as a primary influence because it provides many of the features required and has a procedural syntax familiar to many Matlab users. In addition, the syntax used in the Caesy language is close to that being encouraged as an IFAC and IEEE standard (see [2]).

### Tidbits

Caesy is an interactive shell and thus processes user input on a command by command basis. The prompt '`caesy>`' shown below implies that the input is at the top level of the shell. Caesy is not case sensitive, so '`abc`', '`ABC`', '`AbC`' and '`abC`' are all equivalent. Internally, Caesy treats all symbols as lowercase. Imaginary constants, used heavily in control system analysis, are input using a real constant with a trailing '`i`', '`I`', '`j`' or '`J`' (e.g., `1.0i`, `1.0J`).

## Types in Caesy

Probably the most striking difference between Caesy and other Matlab environments is the fact that Caesy is a typed language. That is, in Caesy all variables have type (e.g., integer, real, string) whereas in Matlab all variables typically have no type (i.e., all variables are the same type, Matrix). The addition of types to a language adds possibilities for making more powerful tools, but may place a burden on the user for declaring types.

In Caesy, we use types but put minimal requirements on users to declare variable types. The only place one is strictly required to declare variable types is when they appear as formal arguments in function and procedure declarations. Otherwise, the type of a new variable (implied by an assignment statement) can be synthesized from the type of the right hand side expression in the assignment. For example, in the statement

```
caesy> abc := 3 + 4;
```

the right hand side has type **integer** so if abc was not previously declared, Caesy automatically declares the variable abc and gives it type **integer**.

Explicit type conversion in Caesy is performed, by convention, by declaring a function with the type name. For example, a function to explicitly convert integers to real is declared as

```
function real(i: integer) return x: real;
```

and then used, for example, as

```
x := sqrt(real(2));
```

## Statements

Caesy supports many of the statements and control structures found in Ada with the multiple assignment form taken from Matlab.

### Assignment Statements

Assignment statements take the normal form as well as the multiple assignment form familiar to Matlab users. Here are examples of the two forms:

```
caesy> abc := 3 + 4;
caesy> { x, y, z } := fctn(abc, 3.0);
```

One of the statement terminators ';' or '?' is required in Caesy. Just hitting the return key will not do. The '?' implies that the result of the assignment should be displayed to the user:

```
caesy> abc := 3 + 4?
  abc := 7;
```

If an expression *expr* appears alone on the command line, an implied assignment of the form

'`_ans_`*type-name* `:=` *expr*`?`'

is interpreted. For example:

```
caesy> 3 + 4;
  _ans_integer := 7;
```

### Flow Control Statements

Control constructs allow the programmer to do branching and looping. In Caesy, the if-then branching is similar to many languages. An example is

```
if i < 0 then
  j := 1;
elsif i = 0 then
  j := 0;
else
  j := 1;
end if;
```

Looping in Caesy is reasonably flexible. There are for-loops, while-loops, etc. Some examples are

```
for i in 1..3 loop j := 3*j; end loop;
while j < 30 loop j := 3*j; end loop;
loop j := 3*j; exit when j >= 30; end loop;
```

## User-Defined Subprograms

Subprograms in Caesy take the form of functions and procedures. (Matlab has no procedures). The following is an example of a user-defined function definition.

```
function myabs(i: integer) return j: integer is
begin
 if i < 0 then j:= -i; else j:= i; end if;
 return;
end myabs;
```

An example of a procedure definition is the following:

```
procedure swap(x, y: in out real) is
  temp: real; -- declaration optional!
begin
  temp := x; x := y; y := temp;
  return;
end swap;
```

### Overloading Functions

One important feature needed in design interfaces for handling complexity is overloading functions and operators. Overloading allows a user to write functions of the same name and purpose to operate on different object types. For example, we could define a function **myabs** similar to the one above, but which operates on reals.

```
function myabs(x: real) return y: real is
begin
 if x < 0 then y := -x; else y := x; end if;
 return;
end myabs;
```

Function overloading is very handy in control system analysis. For example, one could use the expression `freq(G)` to produce a frequency response plot whether `G` represents a transfer function in state-space form, rational form, or anything else. In each case, a different function would be called depending on the argument type.

Caesy does not overload the return argument(s) of a function as Ada does. This is difficult to implement, would prohibit the ability to synthesize expression types and hence would require users to type-declare *all* variables (see above).

Overloading Operators

Caesy also provides the capability to overload most operators. Operator overloading allows users to overload operations such as '`A+B`' where `A` and `B` have user-defined types. For example, given two transfer functions `G1` and `G2` in state-space form or rational form, we could overload the operator '+' to compute a transfer function in state space form for the parallel connection of two transfer functions. The declarations of these functions in Caesy would look like

```
function "+"(G1: st_sp; G2: st_sp)
    return G: st_sp;
function "+"(G1: rat; G2: rat)
    return G: st_sp;
function "+"(G1: st_sp; G2: rat)
    return G: st_sp;
function "+"(G1: rat; G2: st_sp)
    return G: st_sp;
```

If `G1` and `G2` were two transfer functions in either state space (`st_sp`) form or rational (`rat`) form, the expression `G1+G2` would produce a state space representation of the transfer function. An alternative to the expression `G1+G2` is the explicit function call `"+"(G1,G2)`.

Default Input Arguments

Users of Matlab are accustomed to a variable number of input arguments. In the case where input arguments are not given, default arguments must be supplied in the subprogram declaration. In Caesy, a user need not specify all input arguments if default arguments are given in the subprogram declaration. For example, the following function declaration fragment contains a default input argument:

```
function abc(x: real; y: real := ydef)
```

The function could be referenced, for example, as `abc(x1, y1)` or as `abc(x1)`. In the latter case, the value of the global variable `ydef` would be used for the second argument. In Matlab, unspecified inputs are handled within the function body. In Caesy, the user can change the default input variable, and hence the behavior of the function, at any time.

Variable Number of Output Arguments

Caesy supports definition of functions with more than one output argument. In this case, if the function is used in a simple expression, only the first argument is referenced. In the case of a multiple-return assignment (see above), one or more return arguments may be referenced. In the function body, usage of return arguments can be determined via the '`argused`' operator. For example, consider the following function which echoes its input arguments:

```
function echo(i1: integer; i2: integer)
    return { j1: integer; j2: integer } is
begin
  j1 := i1;
  if argused j2 then j2 := i2; end if;
  return;
end echo;
```

Procedures

In Caesy, users may define functions and procedures. Caesy was provided with the ability to define procedures for reasons of efficiency. Procedures allow one to modify variables without creating a duplicate temporary variable. For example, suppose one wanted to write a function to perform a rank-one change on a matrix. This could be done with a function call of the form

```
A = rankfctn(A, x);
```

or with the procedure

```
rankproc(A, x);
```

The difference here is that the function `rankfctn` will create a copy of the matrix `A` to modify and return, and then `A` will be replaced with this matrix. The procedure will never create the copy of the matrix `A`. If the matrix happened to be large, the execution speed between the functional form and procedural form could be quite noticeable.

## Packages

Several Matlab-type packages use the convention of collecting related script files together to form "toolboxes." Caesy formalizes this convention by using the Ada "package" construct. In Caesy, type definitions, functions, procedures and global variables can be collected together in packages. In Matlab, there is no clean way to distinguish between two different functions of the same name in two different toolboxes. This can lead to severe problems and requires that users and toolbox developers take care to avoid the "name-clash" problem. In Caesy, this problem can be avoided easily.

If a user wishes to explicitly refer to an object in one package when there may be a conflict, the user affixes the object name to the specific package name (a concept borrowed from Ada). For example, to explicitly reference the function **xyz** from the package **mypckg**, one would use the construct **mypckg.xyz(*args*)**.

## Matrix Expressions

The ability to create and manipulate matrices is of fundamental importance in control system engineering. In Caesy, matrix expressions, those expressions used for constructing matrices from their parts, are supported in a unique way. An example of a matrix expression is the following:

```
A := [ 1.1, 1.2; 2.1, 2.2];
```

This is a two-by-two real matrix. In Caesy, the type for this matrix is '**ReGeMat**', for *Real General Matrix*. Complex matrices have type '**CoGeMat**'.

```
C := [ 1.0 + 0.1i, 1.2; 2.1i, 2.2];
```

In the future, we plan to have special support for symmetric and Hermitian matrices, sparse matrices, etc. In Caesy, matrices are constructed using overloaded operators. The above expression for defining '**A**' gets internally translated, in essence, to the following sequence of calls:

```
T0v1 := "["(1.1);
T0v2 := ","(T0v1, 1.2);
T0v3 := ";"(T0v2, 2.1);
T0v4 := ","(T0v3, 2.2);
A := "]"(T0v4, 2.2);
```

The variables starting with **T** are temporary variables created by Caesy. When Caesy sees the first '**1.1**' it looks for the function "[" which takes a real argument. (There are also "[" functions defined which take an integer or a complex value as the first argument.) In the Caesy **MATMATH** package, this function is defined, and returns a special type, **regematlist**, for matrix expressions. When the token '**1.2**' is seen, Caesy looks for a function "," which takes arguments of type **regematlist** and **real**. The process continues until the function "]" taking a **regematlist** and returning a **regemat** is called. By implementing matrix expressions with overloaded operators, we have made the matrix expression construct potentially usable in very creative ways.

## Support of Constructors and Destructors

In Caesy, if a procedure of the form

```
procedure constructor(vble: in out Type)
```

exists for some variable of type **ObjType**, then that procedure will be called automatically when the variable comes into scope. Additionally, if a procedure of the form

```
procedure destructor(vble: in out ObjType)
```

exists for some variable of type **ObjType**, then that procedure will be called automatically when the variable comes into scope.

## Implementation

Caesy is written primarily in C. The internal structure is shown in Figure 1. It includes a supervisor to prompt for input, a parser, written in YACC, a byte-code compiler, a byte-code interpreter, a C code compiler, a context (variable, types, etc.) handling module (made of CX/Sys-Ctxt blocks in the figure), an input output interface, and more. The parser outputs LTU (language transfer utility) codes which can be compiled into byte-codes or C code.

# Work in Progress

In this section we discuss features which are partially implemented or are in the process of being implemented.

## User Defined Structured Data Types

Caesy will have the capability for the user to define structured (i.e., record) data types. This is an essential feature needed in the reduction of complexity.

## Exception Handling

Currently, Caesy has some support of exception handling. Internally, every Caesy function returns an exception code to its caller. If the function executes normally, it returns the exception code **NO_ERROR**. On the other hand if some anomaly occurs, a different exception will be raised. For example, if one tries to take the square root of a negative (real) number, a **NUMERIC_ERROR** exception will be raised. Caesy will trap machine exceptions. If a divide-by-zero occurs, a **NUMERIC_ERROR** will be raised. The exception will be passed until it gets to the supervisor level. For example,

```
caesy> a := sqrt(-9.0);
Supervisor caught NUMERIC_ERROR exception.
caesy>
```

This capability will be extended to allow exceptions to be handled in user-defined functions.

## Data Files

Users often have the desire to store and retrieve data. In Caesy, data files will have special binary formats and store data in a machine-independent form. The data files will support user-defined types and hence, must be quite flexible. The stored data files will make use of the Sun XDR (external data representation) [3].

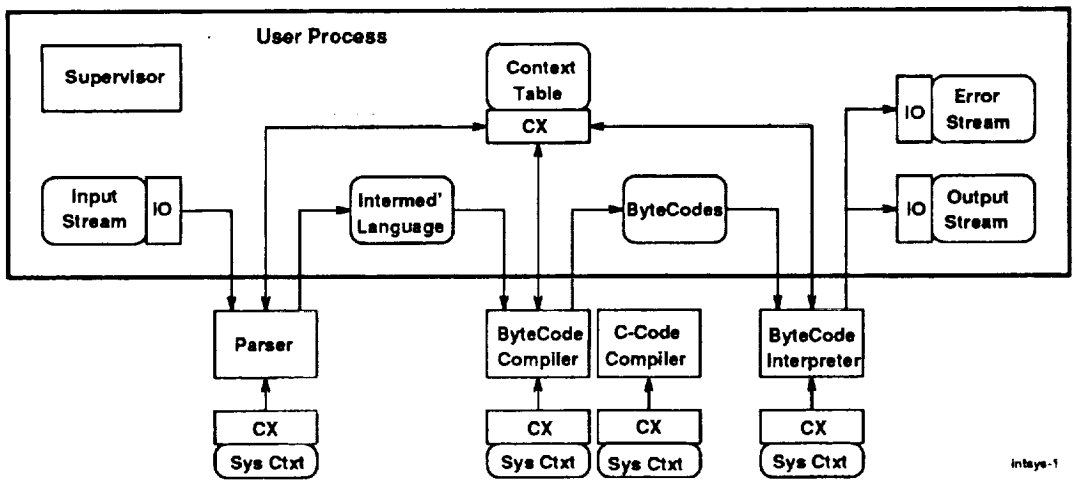


Figure 1: Implementation of Caesy

## Code Generation

Caesy allows users to generate C code from their subprogram scripts. For example, for the following code segment, Caesy will produce the C code shown in Figure 2.

```
use matmath;
function test(A: regemat; B: regemat)
    return C: regemat is
begin
  for i in 1..10 loop
    C := A*B - B*A;
  end loop;
  C := [ C ; [ A, B ] ];
  return;
end test;
```

The C coding capability also makes the prospect of developing stand-alone programs attractive. An often-used application could be converted to C-code and compiled to run as a specialized program, making it more efficient. This feature also has the potential of allowing users to change a routine to make it as efficient as possible. In all, this is a very desirable feature.

## Remote Computing

There is currently work in progress by another team at JPL; the team is implementing a remote computing capability in Caesy. This is being developed using Sun RPC calls [3].

## Matrix Computations

Many of the matrix computations in Caesy are performed using BLAS and LAPACK [4]. These are state-of-the-art FORTRAN libraries in computing solutions to linear equations and computing eigenvalue decompositions. In fact, Caesy could be viewed as a user-friendly interface to LAPACK and, with Caesy's C code compiler, Caesy may be an attractive way to develop code based on both these routines.

## Future Work

In this section we discuss features which are under consideration for future implementations. One object of the Caesy project is to determine what features in an interactive shell are needed for handling large, complex problems.

## Function and Operator "Tagging"

The semantics of creating matrices using the '[', ',', ';' and ']' operators have a drawback. It may be nice to be able to use the construct to input, for example, sparse matrices. A sparse matrix could be input as

```
Asp := [ 1,1,1.1;  2,1,2.1 ];
```

where `1, 1, 1.1` indicates the element in row 1, column 1, is `1.1` and so on. This type of flexibility cannot be supported by the current semantics of the language. There is no way that Caesy knows that a sparse matrix definition is intended. One work-around could be to "tag" the first value with an explicit type cast:

```
Asp := [ spmatelt(1,1,1.1);  2,1,2.1 ];
```

Here we have a special "sparse matrix element" type that tags the first element to allow Caesy to find the function "[", which takes a sparse matrix element type.

Another work-around, which would provide more flexibility to the Caesy language overall, would be to add *tagged* functions and operators to Caesy. In this option, function and operators could have tags to indicate that special operators should be used. The tag would take the special form *function:tag* in both its declaration and use. The tagging approach would allow the user to use the following to generate a sparse matrix (of type `ReSpMat`):

```
Asp := [:sp 1 1, 1.1;  2, 1, 2.1 ];
```

The declaration for the operator [ may have been

```
#include "kc.h"

kcXcode
_U1_1test(
     MATMATH_regemat* a,
     MATMATH_regemat* b,
     MATMATH_regemat* c)
{
  MATMATH_regemat T1v0;
  MATMATH_regemlist* T1v1;
  MATMATH_regemlist T1v2;
  MATMATH_regemat T1v3;
  MATMATH_regemlist* T1v4;
  MATMATH_regemlist T1v5;
  {
    int i;
    MATMATH_regemat T2v0;
    MATMATH_regemat T2v1;
    MATMATH_regemat T2v2;
    {
      i = 1;
      for (;;)
        {
          if (i>10) break;
          {
            MATMATH_1MUL(a, b, &T2v1);
            MATMATH_1MUL(b, a, &T2v2);
            MATMATH_1SUB(&T2v1, &T2v2, &T2v0);
            MATMATH_1ASSN(c, &T2v0);
          }
          i = i + 1;
        }
    }
    MATMATH_1destructor(&T2v2);
    MATMATH_1destructor(&T2v1);
    MATMATH_1destructor(&T2v0);
  }
  MATMATH_2MST(c, &T1v2);
  MATMATH_2MST(a, &T1v5);
  MATMATH_2MEL(&T1v5, b, &T1v4);
  MATMATH_1MND(T1v4, &T1v3);
  MATMATH_2MRO(&T1v2, &T1v3, &T1v1);
  MATMATH_1MND(T1v1, &T1v0);
  MATMATH_1ASSN(c, &T1v0);

  MATMATH_3destructor(&T1v5);
  MATMATH_1destructor(&T1v3);
  MATMATH_3destructor(&T1v2);
  MATMATH_1destructor(&T1v0);

  return(kc_X_NO_ERROR);
}
```

Figure 2: Generated C Code

```
function "[":sp(i: integer)
    return list: respmatlist;
```

## Parallel Computation

Since Caesy has been developed primarily to explore ways in which to increase computational throughput for control design tools, it is only natural to pursue the possibilities of parallel computation. Several workstations and hardware co-processor boards with multiple processors are currently available on the market. How to best support these multiprocessor environments will most likely depend on which machine and operating system we run under. Currently, Mach derivatives and Solaris have support for *multi-threaded* programming at the C language level.

## Conclusion

In this report we have provided a glimpse of a new software environment, Caesy, which will be used for the development of algorithms and software for the design of large, complex control systems. The tool provides a "next-generation" approach to control system design by taking advantage of some concepts from object-oriented programming languages. The tool should prove to provide a means for easily handling large, complex design problems. It is also being used as a "computational engine" in a tool for integrated (conceptual) design of advanced optical systems.

## Acknowledgment

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

## References

[1] M. Wette, "Caesy: A computer-aided engineering system," in *1992 IEEE Symposium on Computer-Aided Control System Design* (Napa, CA), pp. 232–237, March 17-19, 1992.

[2] M. Rimvall, "Command language standard for cacsd software." Control Systems Laboratory, GE-CRD, Schenectady, NY 12301, 1989.

[3] J. R. Corbin, *The Art of Distributed Applications.* New York: Springer-Verlag, 1990.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide.* SIAM, 1992.