A TRANSLATOR FROM NMF TO SPARK

Jean-Michel NATAF
GISE-EMP/ENPC
La Courtine 93167 NOISY LE GRAND CEDEX, FRANCE
email:jmn@cenerg.enpc.fr

ABSTRACT

In this paper we review the basic concepts of the Neutral Model Format for component description, of the SPARK objet-based simulation environment, and present a translator from NMF to SPARK. The purpose of the above is to document the NMF claim that translation from NMF to any simulation environment is easy, and also get a feeling of the effort needed to obtain a working translator. The translator presented handles most NMF constructs (equations, loops, sums, embedded controls) with the notable exception of events. Working examples of translator-generated code are then presented.

2 INTRODUCTION

Building science has in recent years made heavier use of simulation techniques. Large monolithic programs like DOE2 [3], TARP, BLAST [9], were developed, that could simulate the thermal behavior of a whole building. However the size and intricate code of those made them hardly fit for modification, correction or extension.

To solve this difficulty modular environments were then developed: TRNSYS [20] provided a seminal example, with a library of so called "types" representing subroutines modeling common components, but was still of a procedural kind, with predefined inputs and outputs.

A later generation of programs (SPARK [6], IDA [5], ZOOM [11], TK-Solver, ..., and also the latest version (v. 14) of TRNSYS) resorted to object-oriented user interface, allowing the user to build a simulation by hooking together "objects". These objects essentially represented equational models, thus non oriented: the difference is the one between the assignment $x = y$ and the equation $x = y$. The task of building a global simulation program was automated. The main effort of the user or developer was now to design or use appropriate models for each part of the global system to be simulated.

Model libraries exist, like the one of TRNSYS, the one compiled by the Annex 17 of the International Energy Agency, etc. However they are usually coded in a specified language (like FORTRAN), and with predefined inputs and outputs for all subroutines. Object based environments do not assume any a priori orientation of the models: the simulation problem determines the numerical technique to be used to solve it, and, hence, the direction in which each model is to be used.

Thus was perceived the usefulness of some model description language that would be non oriented, and also contain all needed information. Another advantage of this approach, and probably the main one, was to allow various simulation environments to communicate via a pivot language, thus diminishing the number of potential translators. That pivot language approach is already popular in the field of machine translation of natural language. Model description languages being much simpler than natural languages, the idea of a pivot language was bound to be more easily applicable.

The idea of a neutral model format is not new. In the CAD community the IGES exchange format (Initial Graphics Exchange Specification) appeared in 1979 and became an ANSI standard. The French SET (Standard d' Échange et de Transfert) was developed for the aerospace industry. The American PDDI (Product Data Definition Interface) led to the PDES (Product Data Exchange Specification) which itself merged with the European Union sponsored STEP (STandard for Exchange of Product data) conceptual standard for product models and its EXPRESS formalization for computer use [4]. An example of STEP development is the U.S. NIDDESC (U.S. Navy Industries Digital Data Exchange Standard Comitee) with its Distribution System Model [7], taking into count considerations of the Process and HVAC industry.

For the building community, the PROFORMA [8] was initially proposed, but was intended to be human readable. Quite recently the NMF format [18] was proposed, with emphasis on equational

models and computer readability.

3 THE NMF FORMAT

3.1 History

The NMF was initially proposed in 1989. The grammar was fairly simple. As users started to try it, additions to the grammar proliferated up to the current state of version 3.01 described in [17].

Some bugs were fixed, needed features were added. NMF was at that stage a language for component description, with no pretense to describe a complete system (e.g. a building).

At present there is much thought devoted to the introduction of hierarchical models in NMF. That occurred due to the pressure of "popular demand", since the limitation of NMF scope to terminal models was perceived as the main hindrance to the wide usability of NMF.

3.2 Grammar

The NMF grammar is fairly simple, and consists of sections (introduced by keywords) containing relevant information about

- the variables and parameters used (their name, units, range, type..) in the model,
- the links of the model towards the outside (in a good object oriented fashion),
- and the equations of the model, in symbolic form, with additional constructs like tests, summations, loops

As an example, there comes an exerpt of a sample NMF specification for a ramp model:

```
QUANTITY_TYPES
/* Quantity types are used for
   both variables and parameters.
   For parameters the kind is
   irrelevant. */
/* CROSS = Potential, non-directional */
/* THRU = Flow, directional */
/* GENERAL and THERMODYNAMICS */
/* type name       unit          kind */
   Control         "dimless"     CROSS
/* OTHER APPLICATION FIELDS */
/* type name       unit          kind */
LINK_TYPES
/* type name         variable types... */
   ControlLink       (Control)
   ControlLimit      (Control, Control)
CONSTANTS
/* name   value  unit   comment */
ABS_ZERO -273.16 "Deg-C"
   /*absolute zero temp*/
CONTINUOUS_MODEL CoRamp
ABSTRACT "Actuator ramp with discrete input.
 While control input is 0,
 output remains the same.
 If control input is 1, output ramps up.
 If control input is -1, output ramps down.
 Output stays in the interval (lo,hi)."
EQUATIONS
/* RampOK allows movement when
   outsignal within limits;
   Insignal is one of -1, 0, 1*/
  Level' = slope*Insignal*RampOK ;
/* convert normalized output in (-1,1)
   to specific in (lo,hi) */
  0 = -Outsignal+lo*(1-Level)/2
      +hi*(Level+1)/2;
  RampOK := IF event(Outside,Level*
  Insignal-1) > 0 THEN 0 ELSE 1 END_IF;
LINKS
/*   type        name          variables  */
  controllink  In_sig        Insignal ;
  controllink  Out_sig       Outsignal ;

VARIABLES
/*type  name     role def min max descr.*/
control Outside A_S  -0.5
        ">0 if level outside (-1,1)"
control RampOK  A_S   0  0   1
"State =1 if ramping allowed, =0 otherwise"
control Insignal  IN  0  -1  1
        "Input is -1, 0, or 1"
control Level     OUT 0.5 -1 1
        "Normalized output in (-1,1)"
control Outsignal OUT
        "Specific output in (lo,hi)"
PARAMETERS
/*type  name role def min max descr.*/
generic slop S_P  1. -BIG BIG
         "change of output level"
generic lo   S_P  0. -BIG BIG
         "lower limit of output"
generic hi   S_P  1. -BIG BIG
         "upper limit of output"
END_MODEL
```

A complete description of the NMF grammar is available in [17] in Backus-Naur form. Practically, for the user of the NMF model writer, a look at a NMF specification is a quicker method for grasping what the format is.

3.3 Intended use

The NMF is intended, as mentioned above, to be a pivot description language for models, allowing experience of the model developers to be passed over to the simulation specialists via a neutral format that can easily be translated to any simulation environment.

A partial attempt intended to demonstrate feasibility was performed, and yielded a prototype NMF to SPARK translator [16], via the MACSYMA [15] computer algebra system language: the translator created MACSYMA code from NMF, and the MACSYMA code created SPARK code. Also, as NMF was developed by people involved in the development of the IDA simulation, a NMF to IDA translator was developed.

Last but not least, a separate NMF to IDA translator, called NEUTRAN [12] was proposed. It then was extended to also support creation of SPARK code, thus demonstrating the feasibility of well separating the front end of the translator and its part generating target language code.

This paper presents a newer, direct version of a NMF to SPARK translator, independent of the MACSYMA computer algebra language.

4 SPARK

4.1 General overview

SPARK was created in 1986 [2]. It is an environment that automatically writes simulations programs, based on user specifications. The user creates objects (models) and hooks them together. Then SPARK creates a C program that solves for the global equations system, in an efficient way, taking advantage of the possibilities of substitutions in the equation system. The problem size reduction phase is automatically handled by graph theoretical algorithms [22] [14].

The initial SPARK version, called SPANK, handled only algebraic systems. Dynamics were introduced in 1988 [21] in the version called U.S. Energy Kernel System. The current version, called SPARK, has additional features for problem size reduction, like strong component algorithms [6]. A graphical interface is under way.

4.2 Object Oriented interface

As mentioned above, the user interface is object based: objects, for SPARK, are equations (or collections of objects, i.e. equations systems). Equations are a natural way to describe components of continuous systems. Those equations can be algebraic, if necessary piecewise defined, or even ordinary differential equations (ODE's).

A SPARK elementary object, corresponding to a single (possibly piecewise defined) equation, is a complex containing an object file stating what variables of the equation can be explicitly solved for, and C functions actually solving the equations for their solvable variables. The object communicates with other objects via its variables: two equations with a common variable are represented, in SPARK, by two "linked" objects.

In this way building a simulation amounts to hooking together models. The same applies to macro objects, that is objects containing objects, and representing equations systems.

As an illustration, if `obj1` represents the equation $x = y^2$, `obj2` represents the equation $u + v + w = 0$, then the `obj1` object is represented by the following SPARK object code:

```
define obj1(x,y)
double x,y;
{
x=f(y);
y=g(x);
}
```

with f and g being essentially the following C functions:

```
double f(y)
double y;
{
double x;
x=pow(y,2);
return(x);
}

double g(x)
double x;
{
double y;
y=sqrt(x);
return(y);
}
```

while the equations system

$$a^2 = b$$

$$b^2 = c$$

$$a + b + c = 0$$

is represented by the SPARK macro object code (for example):

```
declare obj1 u,v;
declare obj2 w;
link a(u.y,w.u)
link b(u.x,v.y,w.v)
link c(v.x,w.w)
```

One can see that *obj*1 ("power 2" object) is instantiated twice since there are 2 such equations in the system, while *obj*2 (null triple sum object) is instantiated once. And their common variables are linked under a common global name.

SPARK then takes the numerical implementation and coding task into its hands, and delivers an executable program, for which the user just has to supply the relevant inputs.

It must be known that the task of creating the elementary bricks of any SPARK simulation (that is, the elementary and macro objects) is automated: a preprocessor written in a computer algebra system generates SPARK code from symbolic specification of the equations or the equations systems [23] [16].

4.3 Advantages and limitations

The above approach (i.e. the use of the SPARK simulation environment) makes it easy for a user with no numerical expertise to build a working simulation. Also SPARK performs an exact reduction of the size of the system, thus reducing calculation time with no loss of accuracy.

However, SPARK has several limitations: it can only solve for differential-algebraic equations systems, but not partial differential equations (unless these are discretized).

More seriously, it has a fixed time step: the user can specify any time step he wants, but then the time step cannot vary anymore, thus forbidding use of efficient time integration schemes like Gear method. Asynchronous events likewise cannot be simulated at the exact time of their occurrence. We will see this leads to difficulties when evaluating the code generated by the translator.

Also, completely implicit equations cannot at this stage be handled, since SPARK requires each equation to be solvable in at least one variable. Research versions of SPARK, allowing implicit definition of equation objects and associated C functions, overcome this limitation. The current translator work, however, does not assume this feature to be available in SPARK.

Thus all those particularities have an incidence on the NMF-SPARK translator design, as we will see.

5 THE TRANSLATOR

5.1 History

As seen above, a translator was initially designed that translated NMF specifications into MACSYMA code, that in turn created SPARK code.

The currently presented translator bypasses MACSYMA, for portability and cost reasons.

5.2 Tools used

The tools used to write the translator are standard UNIX tools called Lex and Yacc. Lex [13] is a lexical analyzer generator, while Yacc [10] is a syntactical parser generator. That means that Lex recognizes the elementary "words" of the NMF language, while Yacc recognizes the grammatical constructs of the language.

The Yacc syntax is very close to the Backus-Naur form of formal grammar description, thus making it in principle easy to translate the NMF formal description into a working parser.

For example, the recursive definition of an expression would be:

```
expression
        : term
        | expression '+' term
        | expression '-' term
        ;
```

Yacc also allows "actions" to be taken any time a construct is recognized. Typically, in a translator, those actions will be storage of the NMF specification content, and then appropriate code generation for the target language. For example, on can increment the current number of equations that are not assignments by the action:

```
equation
        : expression '='
          expression
          opt_inverse_decl
            {num_equal_equations++;}
        ;
```

It must be emphasized that Lex and Yacc allow to make the front end and the back end of the translator completely modular. The front end is the parser (created by Lex and Yacc). The back end is the code generator, that is essentially a collection of routines invoked by the parser thanks to the Lex and Yacc "actions", on the run or once it has parsed the whole NMF file. Thus it is possible to reuse most of the translator code to create a translator from NMF to another target language than SPARK.

5.3 Implementation

The implementation of the front end was straightforward: one just needed, in principle, to translate the NMF grammar into Yacc form, and then let Yacc create a parser based on the grammar specification. Since the Yacc syntax for grammar description and the Backus-Naur syntax used in the NMF reference manual are very similar, one could envision a painless parser generation. Moreover later modifications of the NMF grammar could easily be implemented in the front end of the translator.

However, unexpected difficulties occurred. Some NMF constructs proved ambiguous, and had to be limited in some way (the most egregious problem was the SUM and FOR construct ambiguity, that made for example `SUM i=1..N-1-x undefined`: is it $(N-1)(-x)$ or $N(-1-x)$? Although pointed out by end 1993, the ambiguity was finally fixed a year later in the informal NMF 1995 Liège meeting, by the introduction of a `OF` keyword between counter and expression). Some semantic distinctions between identifiers in the formal grammar (all the "names" in the grammar) lead to ambiguous Yacc parsers, if one did not exercise care while using the supplied NMF syntax. However little changes to the NMF grammar description (naturally without any changes to the NMF grammar itself!) fixed the problem. All in all, those problems could be overcome relatively easily.

Additional features were added to the grammar rules to make the parser more usable and user friendly, with respect to error recovery. By default when a Yacc parser encounters a syntax error, it terminates immediately. However, adding error acceptance code to the grammar rules in a systematic way leads to a more robust parser. Error recovery was performed using standard recommendations of [19]. This allows the translator to continue when an error is encountered, thus effectively parsing the whole NMF file and reporting errors on the run, the way commercial compilers do.

Thus the first phase, that is the parser generation, was fairly easy, thanks to the adequation of Yacc to the handling of formal grammar descriptions. Grammar ambiguities were spotted while developing the code.

Then came the intermediate phase, the one when the parsed items are stored internally by the translator for further use, semantic checking, and later code generation.

Some difficulties arose when handling test or summation or loop constructs inside the equations. For example,

```
y =
    IF x^2 < 3 THEN
            x^3
    ELSE
            1/x
    END_IF
```

is a valid NMF construct, equivalent to the no less valid:

```
    IF x^2 < 3 THEN
            y = x^3
    ELSE
            y = 1/x
    END_IF
```

Parsing of arithmetic expressions is a well known topic with well known efficient solutions [1]. The introduction of tests inside equations, although not strictly necessary, lead to the establishment of a tree structure representing equations containing IF statements. The nodes of such a tree were either keywords, or expression parts. The handling of those tree structures, and their translation to a more readable piecewise defined representation, was a little heavier than would have been if only plain arithmetic expressions had been allowed. Also some care was to be exercised in the handling of semicolon terminators in embedded constructs.

Once the NMF content had been analyzed and stored internally in a usable form by the translator, arose the problem of code generation for SPARK. The main practical hurdle (that, in our opinion, is not due solely to the SPARK environment) was the clean handling of macro expansion: a typical example is the replacement of

```
FOR i=1..3 x[i]=f(i+1); END_FOR;
```

by

```
x[1]=f(2);
x[2]=f(3);
x[3]=f(4);
```

Owing to the fact that SPARK objects do not support variable number of variables (that is, $x = \sum_{i=1}^{N} f(i)$ with N undefined is not a valid SPARK object: N has to be assigned a known value), macro expansion of loop constructs and sum constructs was needed, with the ensuing complexity of having to introduce an intermediate step in the code generation phase: the translator-generated C code first asks the user what the model parameters (like N in the above example) are, and then generates a C program that generates SPARK code.

That lead to fairly longish SPARK file names, reflecting their macro expansion origin. That in turn led to difficulties due to the fact that UNIX^{TM} differentiates functions in libraries only if their

names are lower than 14 or 15 characters long. But on the whole, the implementation was straightforward as well. SPARK elementary objects, macro objects, whether steady state or dynamic, could be created (the dynamics being easy to handle, since it only requires, in SPARK, to link an integrator object to the object representing the ODE in algebraic form). The C functions associated with the elementary SPARK objects were automatically implemented by using the Newton-Raphson method, that is as is well known a general method for solving non linear equations in terms of a number of variables. One can mention in passing that the C function generation phase entailed fairly convoluted string manipulation when faced to indexes in variables, FOR loops and SUM constructs.

That final SPARK code generation phase concluded the translator design process.

As a side note a separate option generates equations in "natural" form, based on the NMF constructs. So at this stage there are two target language implemented: a trivial one, which amounts to rephrase the NMF content in a form closer to the internal translator structure, and a simple one, the SPARK object language.

5.4 Limitations

The obtained translator supports most of the NMF syntax as described in the reference document [17]. Some limitations remain at this time, and are the following:

Units are not implemented. In SPARK it is possible to type unknowns and give them a "unit", that means a unit name, a default value and extremal values. Thus it is in principle possible to get for each variable the type it belongs to (according to NMF) and create in the relevant SPARK configuration file the associated unit, as well as typing the variables in the object files generated. That was not done, since in SPARK one usually types variables only in the simulation file, and rarely in object files to avoid hidden incompatibilities.

Events are, at this stage, not handled properly, partly because an event in NMF is really two equations written in one. It would not be a lot of work to generate the associated additional equation in the internal translator data structure to handle this. Numerical difficulties would then be likely to occur at run time, since SPARK, as mentioned above, requires a fixed time step and cannot precisely handle asynchronous events.

Splines are not supported at this stage. The support of LINEARIZE is minimal and little tested.

Also, C functions associated to objects are all solved for numerically, even when an explicit solution exists. That is due to the fact that the translator has no symbolic capability at this stage (this is the one loss compared to the previous version of the translator, that relied on the computer algebra system MACSYMA). The drawback is at best a lack of rapidity, and at worst a lack of robustness of the C functions thus generated.

Last, external functions are supported, but not created: *external* keywords are created in the calling functions, but the in-line code present in NMF is not replicated. That is an easy to fix limitation for C external code. For Fortran code, the task is a bit harder, since there exist no portable way to invoke Fortran from C code.

Further limitations of the translator derive from the fact that not all the NMF content is useful or applicable to the SPARK environment. The distinction between inputs and outputs, the variables directions `POS_IN` or `POS_OUT`, the `CROSS` or `THRU` variables, even the distinction between assignments and equations, are hardly relevant to SPARK (unless one chooses to in-line parameter assignment in the equations needing the above mentioned parameters). That is a very general issue, in the sense that NMF assumes the target simulation environment to have a certain number of properties and thus documents them: a simulation environment can be cruder than what NMF assumes, or, on the contrary, even more general.

6 TESTS

6.1 Sample library items

A collection of components, described as examples in the reference document, were fed to the translator and returned appropriate SPARK code (with the limitations mentioned above, essentially related to the occurrence of events).

As a real SPARK applications, the `VxMix` model (mixing box) was cast into a simulation form, with specified inputs, for stand-alone testing. The translator generated SPARK macro object functioned properly -the problem being essentially trivial, and requiring no strong solving capability from SPARK.

6.2 The Hamburg cell

A complete simulation, called the Hamburg cell (that was a benchmark case used to compare the SPARK and ZOOM environment, among others), was also generated with the translator: all SPARK object needed (walls, control, zones) were generated using NMF specifications. The simulation was

then run, and surprisingly enough lead to no numerical difficulty at all, even though all C functions generated by the translator solved for their equations via the general Newton-Raphson method. An explanation to this is, that most dependencies in the Hamburg cell are linear (in which case Newton-Raphson converges in one step), the only real nonlinearity lying in the controller. The results with the NMF translator-generated SPARK objects are the same as the ones previously obtained via MACSYMA generated objects.

6.3 The sample Demand Controlled Ventilation System

The complete simulation presented in the reference NMF document was then implemented. It is a demand controlled ventilation system, with a nonlinear controller, fans, a zone with a leak, and a mixing box. Several components of this model include events, that we have seen are not completely handled by the translator by now.

The translator was thus run on all NMF component specifications. The 5 missing equations corresponding to events were then introduced manually in all objects representing models with `Events` in them. A SPARK simulation was then manually created, that used the translator-generated component objects.

The SPARK simulation thus obtained passes all consistency checks of the SPARK engine (same number of equations and unknowns, existence of a graph theoretically determined resolution method), although it does not converge at this stage, maybe because splines are outrageously simplified in the fan models, or because of the asynchronous events encountered.

Another difficulty is the fact that unlike the hamburg Cell case, implicit resolution of local equations of this problem in SPARK leads sometimes to numerical difficulties.

Also, some models are partially piecewise defined, leaving some variables domains with no expression, thus leading in the resulting SPARK code to functions that are not defined for all possible inputs, thus bringing random garbage into the numerical simulation.

Nevertheless, these difficulties are due either to the NMF specification (incomplete models), or the translator lack of symbolic capabilities (Newton-Raphson being insufficient), or the SPARK weaknesses (discontinuities, etc.). They do not jeopardize the usefulness of the general approach.

7 CONCLUSION

This paper attempted to show the difficulties to be expected when implementing a translator from NMF to an independently developed simulation environment, and also presents a general approach for creating such translators. Although not yet complete, the Lex and Yacc generated parser is modular, general, and reusable for other target languages. It demonstrates the usefulness of NMF as a pivot language for model description, and the efficiency of automatic compiler compiler techniques for translator development.

Further needed work includes final implementation of events (which is straightforward: one needs to add one assignment equation every time and event is noticed by the parser), and introduction of minimum symbolic capabilities (starting with a special handling for assignments).

Other modules for other target languages could then be added.

As a whole, the Lex and Yacc approach is certainly saving time at parser generation time. It is a reasonable tool for someone not too versed in translator writing, and hides most of the difficulties. And the effort that we saw was needed, after the parser generation phase, for creating appropriate code, is unavoidable anyways. Using Lex and Yacc also allows to quickly keep track and implement syntax changes like the ones that have been (and will be?) cropping up, for various reasons, since the introduction of NMF. That fact is an additional reason why we see the Lex/Yacc method as a good choice for translator implementation: it gives modularity, easy control of the grammar accepted by the translator, and needs essentially standard C for functioning (plus UNIX^{TM} if one needs to actually modify the grammar, but there are public domain versions of that environment). Thus the spirit of making NMF usable to a larger community of users and model developers is supported by the approach presented in this paper.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Computer Science and Information Processing. Addison-Wesley, 1977.

[2] Jeffrey L. Anderson. A network definition and solution of simulation problems. Technical Report 21522, Lawrence Berkeley Laboratory, Berkeley, CA 94720, September 1987.

[3] Bruce Birdsall, Walter F. Buhl, Kathy L. Ellington, Ahmet E. Erdem, and Frederick C. Winkelmann. *Overview of the DOE-2 Building Energy Analysis Program, Version 2.1 D.* Lawrence Berkeley Laboratory, Report LBL-19735, February 1990.

[4] Bo-Christer Bjork and Jeff Wix. An introduction to step. Technical report, VTT Technical Centre of Finland and Wix McLelland Ltd., 1991.

[5] Axel Bring. *IDA SOLVER User's documentation.* Kungl Tekniska Hogskolan, January 1992.

[6] Fred Buhl, Ender Erdem, Jean-Michel Nataf, Frederick Winkelmann, and Edward Sowell. Recent improvements in spark: Strong component decomposition, multivalued objects and graphical interface. LBL Report LBL-33906, Lawrence Berkeley Laboratory, August 1993.

[7] Navy Industry Digital Data Exchange Standards Committee. Distribution systems model. Technical report, ISO TC184/SC4/WG1.

[8] Anne Marie Dubois. *Model-Based Computer Aided Modelling: the new perspectives for building energy simulation.* Sophia-Antipolis, France, 4-5 October 1988.

[9] D. Herron, G. N. Walton, and L. Lawrie. *Building Loads Analysis and System Thermodynamics (BLAST) program user's manual volume 1 Supplement (version 3.0)*, March 1981.

[10] S. C. Johnson. *YACC : Yet Another Compiler Compiler*, volume 2. Bell Telephone Laboratories, Inc., Murray Hill, New Jersey 07974, July 31, 1978.

[11] J.-L. Bonin J.-L. Joly, V. Platel, M. Rigal, J.-Y. Grandpeix, and A. Lahellec. Multi-model simulation: the t.e.f. approach. Rome, Italy, June 1989.

[12] Kjell Kolsaker. Recent progress in fire simulations using nmf and automatic translation to ida. In *Proceedings of Building Simulation '93, Adelaide, Australia.* International Building Performance Simulation Association, August 1993.

[13] Mike E. Lesk and E. Schmidt. *LEX - A Lexical Analyzer Generator*, volume 2. Bell Telephone Laboratories, Inc., Murray Hill, New Jersey 07974, 1986.

[14] H. Levy and D. W. Low. *A new algorithm for finding small cycle cut sets.* IBM Scientific Center, Los Angeles, June 1983.

[15] MIT. *MACSYMA Reference Manual, version 10.* Cambridge, MA, 1983.

[16] Jean-Michel Nataf and Frederick Winkelmann. Applications of computer algebra and compiler-compilers for automatic code generation in the simulation problem analysis and research kernel (spark). LBL Report LBL-32815, Lawrence Berkeley Laboratory, September 1992.

[17] Per Sahlin, Axel Bring, and Edward Sowell. The neutral model format for building simulation. Technical report, Swedish Institute of Applied Mathematics, Stockholm and California State University, Fullerton, January 1994.

[18] Per Sahlin and Edward F. Sowell. A neutral format for building simulation models. In *Proceedings of Building Simulation '89, Vancouver, British Columbia*, pages 147–154, P.O. 282, Orleans, Ontario, K1C 1S7, Canada, June 1989. International Building Performance Simulation Association.

[19] Axel T. Schreiner and Jr. H. George Friedman. *Introduction to compiler construction with UNIX.* Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1985.

[20] Solar Energy Laboratory, University of Wisconsin-Madison. *TRNSYS, a Transient Simulation Program*, 1988.

[21] Edward F. Sowell and Walter F. Buhl. Dynamic extension of the simulation problem analysis kernel (spank). In *Proceedings of the USER-1 Conference*, Ostend, Belgium, September 1988. Society for Computer Simulation, La Jolla, CA.

[22] Edward F. Sowell, Walter F. Buhl, and Jean-Michel Nataf. Object-oriented programming, equation-based submodels, and system reduction in spank. In *Proceedings of Building Simulation '89*, pages 141–146, Vancouver, British Columbia, Canada, June 1989. International Building Performance Simulation Association, P.O. 282, Orleans, Ontario, K1C 1S7, Canada.

[23] Edward F. Sowell, Jean-Michel Nataf, and Frederick C. Winkelmann. Radiant transfer due to lighting: An example of symbolic model generation for the simulation problem analysis kernel. LBL Report LBL-28273, Lawrence Berkeley Laboratory, January 1990.