# Object Oriented Microwave Circuit Simulation *

Carlos E. Christoffersen†, Usman A. Mughal‡, Michael B. Steer§

March 27, 2000

**Abstract**

An object-oriented microwave circuit simulation environment is described. The design of the program is intended to offer flexibility without sacrifying efficiency. Recent developments in object-oriented techniques and in C++ compilers are used to obtain a flexible and robust system ideally suited to the development of a global modeling strategy for the integration of circuit, field, thermal and mechanical analyses. The simulation of spatial power combining systems is used as a vehicle to illustrate the architectural developments of the system.

*Keywords:* Circuit simulation, microwave CAE, object-oriented programming, circuit field interaction, global modeling.

# 1 Introduction

The rapid rate of innovation of microwave and millimeter wave systems requires the development of an easily extensible and modifiable microwave computer aided engineering (CAE) environment. While great strides have been made in the flexibility of commercial CAE tools, these sometimes prove inadequate in modeling advanced systems. As with virtually all aspects of electronic engineering the abstraction level of RF and microwave theory and techniques has increased dramatically. In particular, large systems are being designed with attention given to the interaction of components at many levels. One of the most significant developments relevant to microwave computer aided engineering is the rise of object oriented (OO) design practice [1, 2, 3, 4, 5, 6][1]. While it is normal to think of OO-specific programming languages as being the main technology for implementing OO design, good OO practice (with limitations) can be implemented in more conventional programming languages such as C. However OO-specific languages foster code reuse and have constructs that facilitate

*This work was supported by the Defense Advanced Research Projects Agency (DARPA) through the MAFET Thrust III program as DARPA Agreement Number DAAL01-96-K-3619.

†C. E. Christoffersen is with the Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7914, U.S.A., (e-mail: c.christoffersen@ieee.org).

‡U. A. Mughal is with Intel Corp. PTD, Portland, Oregon 97124, (e-mail: uamughal@ieee.org)

§M. B. Steer is with the Institute of Microwaves and Photonics, School of Electronic and Electrical Engineering, University of Leeds, Leeds, United Kingdom LS2 9JT, (e-mail: m.b.steer@ieee.org).

[1]These references are available on line at http://www.objectmentor.com/

object manipulation. The OO abstraction is well suited to modeling electronic systems, for example, circuit elements are already viewed as discrete objects and at the same time as an integral part of a (circuit) continuum. The OO view is a unifying concept that maps extremely well onto the way humans perceive the world around them. Non-OO circuit simulators always become complicated with many layers of special cases. Referring to circuit elements again, traditional simulation implementations have many "if-then" like statements and individually identify every element in many places for special handling.

Traditionally papers on advances in circuit simulation technology have presented algorithmic developments which usually resulted in more robust and speedier circuit modeling technologies. In some cases additional capability is presented in that strategies for simulating electronic systems that could not be modeled previously are presented. In contrast, what this paper does more than anything else is present a level of abstraction higher than that previously reported for modeling microwave circuits and systems. This is not a paper on how to write a computer program to implement circuit simulation. Rather it is the latest contribution to a theoretical OO circuit modeling framework to which many have contributed. This is a traditional model for reporting research results. Specifically, the material presented here is the result of experimentation in implementing a circuit-focused global modeling strategy integrating apparently disparate analyses [7]. The CAE environment described in this paper is intended to facilitate research in modeling very large systems integrating electromagnetic, thermal, physical device models and various other analysis types. A capability not previously supported for RF and microwave circuits and systems.

There are a few key premises that drove the work reported here. One of these is the adoption of a very strong OO paradigm throughout to obtain a modular design. Also, an integral part of the various high performance computing initiatives is the separation of the core components embodying numerical methods from the modeling and solver formulation process with the result that numerical techniques developed by computer scientists and mathematicians can be formulated using formal correctness procedures. Thus, what is adopted here, is that the circuit abstraction is adapted so that highly reliable and efficient pre-developed libraries can be used.

C++ was once considered slow for scientific applications. Advances in compilers and programming techniques, however, have made this language attractive and in some benchmarks C++ outperforms Fortran [8, 9]. Several OO numerical libraries have been developed [10]. Of great importance to the work described here is the incorporation of the standard template library (STL) [11]. The STL is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template. The current ISO/ANSI C++ standard [12] has not been fully implemented and C++ compilers support a variable subset of the standard. The biggest areas of noncompliance being the templates and the standard library.

The circuit simulator implementing the ideas presented here is called Transim. We present for the first time a circuit simulator using some of the recently developed formal OO techniques and C++ features. The design intent was to to combine the advantages of previous OO circuit simulators with these new developments as well as expanding capability. Transim uses C++ libraries [13, 14] and several written in C or Fortran [15, 16, 17]. In the following OO circuit simulators are first reviewed and the specific OO programming construction used

in the current work is described. Then an example is presented integrating electromagnetic and circuit analysis in modeling a microwave CPW active antenna.

# 2 Background

*APLAC*[2] [18, 19] is a significant achievement in the development of object-oriented circuit simulators with the object orientation implemented in the standard C language using macros. The most important feature of APLAC is that every circuit element is modeled internally using independent and voltage-controlled current sources. Since all models in APLAC are eventually mapped to current sources, the simple nodal linear DC analysis, $\mathbf{Gu} = \mathbf{j}$, is all that is required to realize nonlinear DC, AC, transient and harmonic balance analyses. Here $\mathbf{G}$, $\mathbf{u}$ and $\mathbf{j}$ denote conductance matrix, nodal voltages and the independent source currents, respectively. The cost of this approach is reduced speed. In part this is because the C language is not optimum for OO applications but also because the high level of abstraction introduces overhead. However the objective of providing great functionality to enable experimentation with new element types and analysis techniques was achieved. The current version of the program incorporates many advanced features including electro-thermal analysis and is commercially available.

Other OO circuit simulators are *CODECS* [20], *ACS*[3] [21] and *Sframe* [22] and these adopted a common interface for all the circuit elements. In this way, all the code related to one element is separated from the rest of the program. In other words, the main program does not have dependencies on individual elements. The result is that the programming effort required to add new elements and algorithms is greatly reduced. ACS and Sframe are written in C++ and among other features, they both allow one element to be composed of other basic elements. The underlying algorithms in ACS are the same as those in Spice. As well as the flexibility introduced by the OO design there are memory savings in storing the circuit.

Sframe incorporates several novel features including automatic differentiation. In this simulator C++ is used as the circuit description language rather than, say, a Spice netlist. This arrangement yields a level of flexibility difficult to achieve using a netlist parser or graphical interface. On the other hand, the netlist must be compiled and the simulator linked for each circuit, and the user must be aware of the subtle details of the C++ syntax.

Other considerations about the OO design of circuit simulators are presented in [23].

# 3 Structure of the Program

In this Section the focus is on the design of the object-oriented structure of the program. The goal in design was to obtain speed in development, to use 'off the shelf' advanced numerical techniques, and to allow easy expansion and testing of new models and numerical methods.

[2]http://www.aplac.hut.fi/aplac/main.html

[3]http://www.geda.seul.org/dist/

## 3.1 The Network Package

The network package is the core of the simulator. All the elements and the analysis classes are built upon it as shown in the class diagram of Figure 1. (See Appendix 1 for a description of the class diagram syntax.) Following the suggestion made in [21], there is a **NetListItem** class that is the base for all classes of objects that appear in the input netlist. This is the base class that handles parameters. Figure 2 shows some of the methods provided by this class. All the netlist items share a common syntax so that the element model developer does not need to worry about the details of element parsing and there is no need to modify the parser to add new elements. For compatibility reasons, Spice-type syntax (which does not have a consistent grammar) is supported by the parser outside the network package.

The **Element** class contains basic methods common to all elements as well as the interface methods for the evaluation routines. Some of the methods of this class (Figure 3) need to be overridden by the derived classes. For example, in class **Diode**, `svTran()` is intended to contain the code to evaluate the time domain response of a diode. This function is used by DC and transient analyses. The same happens with `svHB()` and `fillMNAM()`. The overhead imposed by these virtual functions is small compared to the time spent evaluating the functions themselves and so this approach is a good compromise between flexibility and efficiency. This idea has been used in [20, 21, 22]. Transim also offers a more elaborate mechanism for nonlinear element evaluation functions which will be described in Section 3.4.

The **ElementManager** class is mainly responsible for keeping a catalog of all the existing elements. Note that this class is the only one that 'knows' about each and every type of element but this dependency is weak. The element list is included from an automatically generated file. **ElementManager** is also used to automatically generate the element catalog documentation in html format, see (Figure 4). The **Circuit** class represents either a main circuit or a subcircuit as a collection of elements and terminals. It provides methods to add, remove and find elements and terminals using different criteria and it also provides methods related to circuit topology. More details about this class are given in Figure 5. All the **Element** and **Terminal** instances must be stored in data structure inside the **Circuit** instance and the *map* container of the STL [11]. The map is a *Sorted Associative Container* that associates objects of type *Key* with objects of type *Data*. Here *Data* is either **Element** or **Terminal** and *Key* is *int* (the ID number). This is an example of where the features of C++ are used to reduce development time. This is achieved at no overhead as an optimum implementation of these concepts is embedded in the compiler.

Subcircuit instances are represented by the **Xsubckt** class, see Figure 6. The method `attachDefinition()` is used to associate a **Circuit** instance where the actual subcircuit is stored to the particular **Xsubckt** instance and `expandToCircuit()` takes a **Circuit** pointer as argument and expands the subcircuit. Note that before expansion the complete hierarchy of a circuit is available in memory, so this engine could eventually be used to perform hierarchical simulation.

## 3.2 Example of Element Implementation: CPW Transmission Line

The flexibility for element creation is illustrated by showing one implemention of the CPW transmission line element, Figure 7. Other implementations are also possible. Since the CPW

model is similar in concept to the physical transmission line model (Tlinp4 element) shown in Figure 4, the CPW element can be implemented as shown in Figure 8. The **Element** class (Figure 3) provides a default `init()` function which is executed after all the parameters values and the terminal connections are set. This method is *overridden* by the **CPW** class (this means the default function is replaced by a custom function for the **CPW** class) as follows. After all the CPW parameters are set, the CPW `init()` function calculates the equivalent parameters for a Tlinp4 element (see Figure 4 for a description of parameters) and inserts it in the circuit. The actual transmission line equations are then handled by the underlying Tlinp4 element and this relationship applies to nearly every other aspect of the model such as the information to check local reference nodes [24, 25].

This kind of code reuse could also be achieved using the conventional procedural paradigm of programming. The advantage of the OO apprach is that all the data, as well as all the functions needed to implement the physical transmission line element are contained in one class, thus making it easier to handle the elements as independent 'blocks'. This is known as the *encapsulation* mechanism which provides modularity of the code (see Appendix 2).

The concept of this example can also be used to create elements composed of a set of other elements, or 'hardwired subcircuits', although no elements of this type are currently implemented.

## 3.3 The Analysis Classes

Figure 9 shows the relation between the network package, the elements and the analysis classes. Each of these classes stores analysis-specific data that would traditionally be global. The concept goes farther than this as all the analysis code is encapsulated inside each class. This leads to the key desired attribute of flexibility in incorporating a new type of analysis, or even different implementations of the same analysis type. Examples are the **SVTr** and **SVHB** classes which contain the state-variable- convolution transient and harmonic balance analyses described in [26] and [27] respectively.

There are some components common to two or more analysis types. The natural way of handling these components is by creating a class which is shared by the different analysis types. For example, the **FreqMNAM** class handles a Modified Nodal Admittance Matrix (MNAM) in the frequency domain. In a microwave simulator, the frequency domain admittance matrix is a key element for most analysis types. Since it is used so often, special care was taken to optimize efficiency. The elements fill the matrix directly without the need for intermediate storage of the element stamp. They do that by means of in-line functions to reduce function call overhead. Elements can fill the source vector in a similar way. The elements depend on the **FreqMNAM** methods, but this is not a problem since the interface is very unlikely to change. The current implementation of MNAM uses the Sparse library, described in the next Section, and is completely encapsulated inside the **FreqMNAM** class. In the same way, the **NLSInterface** class encapsulates the nonlinear solver routines. Therefore it is possible to replace the underlying libraries, if that is desired, without the need for any code modification outside the wrapper classes. A final observation is that it is possible to add any kind of analysis type provided that the appropriate interface is defined and the member functions are written for each element type.

## 3.4 Nonlinear Elements

Nonlinear elements often use service routines provided by the analysis classes. In order to maximize code reuse and to avoid the dependence of the element code on a particular analysis routines, in Transim elements depend on interface classes (Figure 10). The concept is similar to the *dependency inversion* [2]. This is a technique which relies on interface classes (normally implemented using abstract classes in C++) to make different parts of a program independent of each other. They only depend on the interfaces. To achieve greater efficiency, in Transim the dependency inversion is implemented using a concrete class with heavy in-lining and pass-by-reference.

In this way, the element routines and the analysis depend on an interface class, **TimeDomainSV** (not shown in Figure 1 for clarity). **TimeDomainSV** is a class that is used to exchange information between an element and a state variable based time domain analysis. It also provides some basic algorithms such as time differentiation methods. This approach enables the element routines to be reused by several analysis types without the need to modify the element code (as long as the new analysis is state variable-based). For example, the **DC** analysis uses the same interface element as the **SVTr**.

Transim offers a more refined way to implement nonlinear elements, provided the element equations can be expressed in the following parametric form [28]

$$\mathbf{v_{NL}}(t) = u[\mathbf{x}(t), \frac{d\mathbf{x}}{dt}, \ldots, \frac{d^n\mathbf{x}}{dt^n}, \mathbf{x}_D(t)] \quad (1)$$

$$\mathbf{i_{NL}}(t) = w[\mathbf{x}(t), \frac{d\mathbf{x}}{dt}, \ldots, \frac{d^n\mathbf{x}}{dt^n}, \mathbf{x}_D(t)] \quad (2)$$

where $\mathbf{v_{NL}}(t)$, $\mathbf{i_{NL}}(\mathbf{t})$ are vectors of voltages and currents at the element ports, $\mathbf{x}(t)$ is a vector of state variables and $\mathbf{x}_D(t)$ a vector of time-delayed state variables, *i.e.*, $x_{Di}(t) = x_i(t - \tau_i)$. All vectors in (1) and (2) have the same size $n_d$ equal to the number of state variables of the element.

Given these conditions, by implementing the parametric equations (1) and (2) using a special syntax in only one function, Transim can obtain the analysis functions `svHB()`, `svTran()` and derivatives automatically. This mechanism is termed *generic evaluation*.

Figure 11 shows the class diagram for an element using this feature. Note that the **Diode** class is derived from a class (**AdolcElement**) which itself provides the analysis routines and deals with the analysis interfaces. The **Diode** class only needs to implement the `eval()` function with the parametric equations.

**AdolcElement** uses the Adol-C library (see Section 4.5 for a detailed explanation) to evaluate the parametric function and derivatives, but the concept is independent of automatic differentiation. If automatic differentiation were not used, then the derived class (*e.g.* the **Diode** class) would have to provide the Jacobian for the parametric equations (1) and (2).

The idea is in a way similar to the one used in [19], *i.e.* the primitive equations are 'wrapped' in analysis-specific generic functions and so there is no need to write a separate routine for each analysis type. In the current work there are two additional features. The first is that the generic evaluation is combined with the state variable concept and automatic differentiation. This provides unprecedented simplicity to create nonlinear element models. The second is that a single mechanism is not mandatory. There are cases where it may not

be practical to use this approach, or the overhead involved (which is completely acceptable even for simple models such as a diode) may not be properly amortized. In those cases, the element can implement the analysis functions directly, without using the **AdolcElement** class.

It is important to remark that generic evaluation is implemented efficiently so there are no superfluous calculations. The current implementation supports elements with any number of state variables. Each element selects the input variables as a subset of the following: the state variables, the first derivatives, the second derivatives and a time delayed version of the variables (the delay may be different for each). No derivation, time delaying nor transformation is performed on the unselected inputs.

For example, an element with only an algebraic nonlinearity (such as the VCT: voltage controlled transducer) only selects the state variables without any derivatives or delays. As another example, the **MesfetM** class implements the Materka-Kacprzac model for a MESFET. It requires two state variables, but only one of them needs to be delayed.

A consequence of having a library of elements using generic evaluation is that it is possible to add a new analysis type by just adding the appropiate evaluation routines to the **AdolcElement** class. Thus, the maintainance and expansion of the simulator is simplified.

### 3.5 Example: Use of Polymorphism

The previous scheme constitutes a good example of the use of polymorphism to solve a complex problem. Consider the segment of code of Figure 12. This code corresponds to the evaluation of the nonlinear element functions in the state variable convolution transient. `elem_vec` is a vector of Element pointers implemented using the vector container of the C++ STL. There is no need to keep the size of the vector in a separate variable as `elem_vec.size()` returns the size of the vector. Also, the memory management of the vector is dynamic and automatic; and `elem_vec[k]` returns the Element pointer at position k in the vector.

Each pointer inside `elem_vec` points to different kinds of elements. For the transient routine the actual type of each element does not matter . The line containing
`elem_vec[k]->svTran(tdsv)` will call the appropriate evaluation routine depending on the actual type of Element pointer. The element may implement the routine directly or through generic evaluation, but it makes no difference for the analysis routine.

The resulting code is therefore simple and there is no need for lists of "if-then" statements. This would be very difficult to maintain, because each time an element is added or removed, all the lists would have to be updated.

## 4 Support Libraries

A large number of support libraries are available (many of them freely) and some of these are used in Transim. The various libraries, which should be of general interest to the microwave modeling community, are described below.

## 4.1 Modified nodal admittance matrix representation

*Sparse 1.3*[4] [16] is a flexible package of subroutines written in C used to quickly and accurately solve large sparse systems of linear equations. The package is able to handle arbitrary real and complex square matrix equations. Besides being able to solve linear systems, it is also able to quickly solve transposed systems, find determinants, and estimate errors due to ill-conditioning in the system of equations and instability in the computations. Sparse also provides a test program that is able to read matrix equation from a file, solve them, and print useful information about the equation and its solution. Sparse was originally written for use in circuit simulators and is well adapted to handling nodal- and modified-nodal admittance matrices.

## 4.2 Vectors and matrices

Most of the vector and matrix handling uses *MV++*[5] [14]. This is a small set of concrete vector and simple matrix classes for numerical computing written in C++. It is not intended as a general vector container class but rather designed specifically for optimized numerical computations on RISC and pipelined architectures which are used in most new computer architectures. The various MV++ classes form the building blocks of larger user-level libraries. The MV++ package includes interfaces to the computational kernels of the Basic Linear Algebra Subprograms package (BLAS) which includes scalar updates, vector sums, and dot products. The idea is to utilize vendor-supplied, or optimized BLAS routines that are fine-tuned for particular platforms. More complete matrix packages such as *Blitz++*[6] or linear algebra packages such as the *Matrix Template Library* (MTL)[7] and the *Template Numerical Toolkit*[8] are available but the GNU gcc compiler version 2.8.1 and earlier used at the time to develop Transim are not capable of compiling them. Transim is now being developed using GNU gcc 2.95.

## 4.3 Solution of nonlinear systems

Nonlinear systems are solved using the *NNES*[9] [17] library. This package is written in Fortran and provides Newton and quasi-Newton methods with many options including the use of analytic Jacobian or forward, backwards or central differences to approximate it, different quasi-Newton Jacobian updates, or two globally convergent methods, etc. This library is used through an interface class (**NLSInterface**), so it is possible to install a different library if desired by just replacing the interface (four different nonlinear solvers have already been used).

[4] http://www.netlib.org/sparse/
[5] http://math.nist.gov/mv++/
[6] http://oonumerics.org/blitz/
[7] http://www.lsc.nd.edu/research/mtl/
[8] http://math.nist.gov/tnt/
[9] http://www.netlib.org/opt/

## 4.4 Fourier transform

Fourier transformation is implemented using the *FFTW*[10] library [15]. FFTW is a C subroutine library for computing the Discrete Fourier Transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. The authors of this library believe that FFTW, which is freely available, should become the FFT library of choice for most applications. Benchmarks, performed on a variety of platforms show that FFTW's performance is typically superior to that of other publicly available FFT software. Moreover, FFTW's performance is portable: the program performs well on most computer architectures without modification.

## 4.5 Automatic differentiation

Most nonlinear computations require the evaluation of first and higher derivatives of vector functions with $m$ components in $n$ real or complex variables [13]. Often these functions are defined by sequential evaluation procedures involving many intermediate variables. By eliminating the intermediate variables symbolically, it is theoretically always possible to express the $m$ dependent variables directly in terms of the $n$ independent variables. Typically, however, the attempt results in unwieldy algebraic formulae, if it can be completed at all. Symbolic differentiation of the resulting formulae will usually exacerbate this problem of *expression swell* and often entails the repeated evaluation of common expressions.

An obvious way to avoid such redundant calculations is to apply an optimizing compiler to the source code that can be generated from the symbolic representation of the derivatives in question. Exactly this approach was investigated by Speelpenning during his Ph.D. research [29] at the University of Illinois from 1977 to 1980. Eventually he realized that at least in the cases $n = 1$ and $m = 1$, the most efficient code for the evaluation of derivatives can be obtained directly from the evaluation of the underlying vector function. In other words, he advocated the differentiation of evaluation algorithms rather than formulae. In his thesis he made the particularly striking observation that the gradient of a scalar-valued function (*i.e.* $m = 1$) can always be obtained for no more than five times the operations count of evaluating the function itself. This bound is completely independent of $n$, the number of independent variables, and allows the row-wise computation of Jacobians for at most $5m$ times the effort of evaluating the underlying vector function.

Given a code for a function $F : \Re^n \rightarrow \Re^m$, automatic differentiation (AD) uses the chain rule successively to compute the derivative matrix. AD has two basic modes, forward mode and reverse mode [30]. The difference between these two is the way the chain rule is used to propagate the derivatives.

A versatile implementation of the AD technique is *Adol-C*[11] [13], a software package written in C and C++. The numerical values of derivative vectors (required to fill a Jacobian in Harmonic Balance analysis [27], see Figure 13) are obtained free of truncation errors at a small multiple of the run time with little additional memory required. It is important to note that AD is not numerical differentiation and the same accuracy achieved by evaluating analytically developed derivatives is obtained.

[10] http://www.fftw.org

[11] http://www.math.tu-dresden.de/ adol-c/

The `eval()` method of the nonlinear element class is executed at initialization time and so the operations to calculate the currents and voltages of each element are recorded by Adol-C in a *tape* which is actually an internal buffer. After that, each time that the values or the derivatives of the nonlinear elements are required, an Adol-C function is called and the values are calculated using the tapes. This implementation is efficient because the taping process is done only once (this almost doubles the speed of the calculation compared to the case where the functions are taped each time they are needed). When the Jacobian is needed, the corresponding Adol-C function is called using the same tape. We have tested the program with large circuits with many tones, and the function or Jacobian evaluation times are always very small compared with the time required to solve the matrix equation (typically some form of Newton's method) that uses the Jacobian. The conclusion is that there is little detriment to the performance of the program introduced by using automatic differentiation. However the advantage in terms of rapid model development is significant. The majority of the development time in implementing models in simulators, particularly harmonic balance simulators, is in the manual development of the derivatives equations. Unfortunately the determination of derivatives using numerical differences is not sufficiently accurate for any but the simplest circuits. With Adol-C full 'analytic' accuracy is obtained and the implementation of nonlinear device models is dramatically simplified. From experience the average time to develop and implement a transistor model is an order of magnitude less. Note that time differentiation, time delay and transformations are left outside the automatic differentiation block and this is approximately ten times faster than including them inside the block.

# 5 Discussion

The successful development of the global modeling strategy required modification of the way the simulator worked. In particular it required the experimentation with various ways to model the interface of different types of analyses, e.g. circuit, field and thermal analyses. Key results led to the adoption of the local reference node concept [24, 25], and state variable-based harmonic balance and transient analyses [26, 27]. This process of experimentation would have been difficult without the flexibility and inherent code integrity resulting from the OO approach described here.

## 5.1 Example: CPW Folded-Slot Active Antenna Unit Cell

As an example of the type of problem that can be modeled consider the CPW folded-slot active antenna [31] shown in Figure 14. This is a component of a spatial power combining circuit. The unit cell amplifier is excited by an incident horizontally polarized field and radiates an amplified vertically polarized field. Complete analysis of this structure uses electromagnetic characterization as well as circuit simulation. In Figure 14 the two orthogonal folded-slots are connected to each other by a CPW with an inserted MMIC amplifier.The system is modeled using the circuit of Figure 15 and electromagnetic modeling of the structure is discussed in [32, 33, 34]. Note that three different local reference nodes, indicated by the diamond shaped node symbol, are required. EM modeling yields port-based $y$ parame-

ters of the antennas at each frequency of interest. The transfer of data between the EM and circuit simulators (typically a file) includes a header with port grouping information (a port grouping includes terminals associated with a specific local reference node). This is required by the circuit simulator in order to expand the port-based matrix into nodal form and also to check the connectivity of the spatially-distributed circuit. The currents calculated by this simulation are used by an EM program to evaluate the effective isotropic power gain of the amplifier cell. The comparison between the simulated results and the measurements presented in [31] are shown in Figure 16.

### 5.2 Example: 2x2 CPW Folded-Slot Active Antenna Array

The 2x2 antenna array shown in Figure 17 is modeled using the equivalent circuit in Figure 18. A multi-port distributed element is used to model the antenna array so the electromagnetic interactions of the antennas of different cells is taken into account.

The netlist (Figure 19) shows the syntax of subcircuits and of local refeence nodes. The `.model` statements can be used in conjunction with any element type and here it is used for the transmission lines. The output statement (`.out`) provide a calculator to process output data.

The output currents are plotted in Figure 20. As expected, the currents are all different since the system is not symmetric.

## 6 Conclusion

Object oriented techniques offer significant advantages for the design of circuit simulators. Great design flexibility is obtained without compromising efficiency. This is due to advances in both programming techniques and compiler technology. However careful analysis of the problem and programming discipline are required. The use of 'off-the-shelf' libraries permits rapid development, higher quality of the code, and as they implement modern numerical methods which are regularly updated, currency is maintained. The simulator was developed to model spatially distributed circuits, and in particular spatial, power combining systems where electromagnetics, circuit and thermal interactions are important. The simulator architecture presented here is suited to the experimentation of new simulation algorithms and element models. Transim source code is freely available by contacting the authors.

## Appendix 1: Object Oriented Programming Basics

Object oriented programming (OOP) [1] provides a means for abstraction in both programming and design. OOP does not deal with programming in the sense of developing algorithms or data structures but it must be studied as a means for the organization of programs and, more generally, techniques for designing programs.

As the primary means for structuring a program or design, OOP provides *objects*. Objects may model real life entities, may function to capture abstractions of arbitrary complex phenomena, or may represent system artifacts such as stacks or graphics. Operationally, objects control the computation. From the perspective of program development, however,

the most important characteristic of objects is not their behavior as such, but the fact that the behavior of an object may be described by an abstract characterization of its interface. Having such a characterization suffices for the design. The actual behavior of the object may be implemented later and refined according to the need. A *class* specifies the behavior of the objects which are its instances. Also, classes act as templates from which actual objects may be created. An *instance* of a class is an object belonging to that class. A procedure (or function) inside an object is called a *method*. A *message* to an object is a request to execute a method.

*Inheritance* is the mechanism which allows the reuse of class specifications. The use of inheritance results in a class hierarchy that, from an operational point of view, decides what is the method that will be selected in response to a message.

Finally, an important feature of OO languages is their support for *polymorphism*. This makes it possible to hide different implementations behind a common interface.

The notion of flow diagram in procedural programming is replaced in OOP by a set of objects which interact by sending messages to each other.

We will briefly review what are traditionally considered to be features and benefits of OOP. Both *information hiding* (also known as *encapsulation*) and *data abstraction* relieve the task of the programmer using existing OO code, since with these mechanisms the programmer's attention is no longer distracted by irrelevant implementation details. The flexible dispatching behavior of objects that lends them their polymorphic behavior is due to the dynamic binding of methods to messages. For the C++ language, polymorphic object behavior is effected by using *virtual functions*, for which in contrast to ordinary functions, the binding to an actual function takes place at run time and not at compile time.

Encapsulation promotes *modularity*, meaning that objects may be regarded as the building blocks of a complex system. Another advantage often attibuted to the OOP is code reuse. Inheritance is an invaluable mechanism in this respect, since it enables the programmer to modify the behavior of a class of objects without requiring access to the source code.

Although an object oriented approach to program development indeed offers great flexibility, some of the problems it addresses are intrinsically difficult and cannot really be solved by mechanisms alone. For example, it is more likely to achieve a stable modularization when shifting focus from programming to design.

C++ virtual functions [12] can have big performance penalties such as extra memory accesses or the possibility of an unpredictable branch so pipelines can grind to a halt (note however, that some architectures have branch caches which can avoid this problem). There are several research projects which have demonstrated success at replacing virtual function calls with direct dispatches. Note however, that virtual functions are not always bad when it comes to performance. The important questions are: How much code is inside the virtual function? How often is it used? If there is a lot of code (*i.e.* more than 25 flops), then the overhead of the virtual function will be insignicant. But if there is a small amount of code and the function is called very often (*e.g.* inside a loop), then the overhead can be critical.

# Appendix 2: UML diagrams

The Unified Modeling Language (UML) [35, 6] is a language for specifying, visualizing, and constructing the artifacts of software systems as well as for business modeling. The goal of UML is to become a common language for creating models of object oriented computer software. It is used here to graphically illustrate the relationship of classes using what is called a class diagram. A class diagram is a graph of Classifier elements with connections indicating by their various static relationships (Figure 1). (Note that a "class" diagram may also contain interfaces, packages, relationships, and even instances, such as objects and links. Perhaps a better name would be "static structural diagram" but "class diagram" is shorter and its use is well established.) A class is drawn as a solid-outline rectangle with 3 compartments separated by horizontal lines. The top name compartment holds the class name and other general properties of the class (including stereotype); the middle list compartment holds a list of attributes; the bottom list compartment holds a list of operations. Either or both of the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment. If a compartment is suppressed, no inference can be drawn about the presence or absence of elements in it.

Each instance of type **Element**, for example, seems to contain an instance of type **ElementData**. This class relationship is indicated by the joining line. The relationship is composition — indicated by the solid diamond symbol. The arrowhead denotes that the relationship is navigable in only one direction, *i.e.*, **ElementData** does not know anything about **Element**. The inheritance relationship in UML is depicted by the triangular arrowhead and points to the base class. A line from the base of the arrowhead connects it to the derived classes, e.g. **Element** is derived from **NetListItem**.

Other forms of containment do not have whole/part implications and are called association relationships indicated by a line drawn between the participating classes. (This relationship will almost certainly be implemented using pointers unless it is very weak.) If the relationship between two classes is very weak (*i.e.* very little data is shared) then a dashed line is used. For example, in Figure 1, **ElementManager** somehow depends upon **Diode**. (In C++ the weak relationship is almost always implemented using an `#include`.)

An illustration showing examples for the notation is given in Figure 21.

# References


[1] A. Eliëns, Principles of object-oriented software development, Adison-Wesley, 1995.

[2] R. C. Martin. "The dependency inversion principle," C++ Report, May 1996.

[3] R. C. Martin, "The Open Closed Principle," C++ Report, Jan. 1996.

[4] R. C. Martin, "The Liskov Substitution Principle," C++ Report, March 1996.

[5] R. C. Martin, "The Interface Segregation Principle," C++ Report, Aug 1996.

[6] R. C. Martin, "UML Tutorial: Part 1 — Class Diagrams," Engineering Notebook Column, C++ Report, Aug. 1997.

[7] M. B. Steer, J. F. Harvey, J. W. Mink, M. N. Abdulla, C. E. Christoffersen, H. M. Gutierrez, P. L. Heron, C. W. Hicks, A. I. Khalil, U. A. Mughal, S. Nakazawa, T. W. Nuteson, J. Patwardhan, S. G. Skaggs, M. A. Summers, S. Wang, and A. B. Yakovlev, "Global modeling of spatially distributed microwave and millimeter-wave systems," IEEE Trans. Microwave Theory Techniques, June 1999, pp. 830-839.

[8] A. D. Robison, "C++ Gets Faster for Scientific Computing," Computers in Physics, vol. 10, pp. 458-462, 1996.

[9] J. R. Cary and S. G. Shasharina, "Comparison of C++ and Fortran 90 for Object-Oriented Scientific Programming," Available from Los Alamos National Laboratory as Report No. LA-UR-96-4064.

[10] The Object Oriented Numerics Page, http://oonumerics.org/.

[11] Silicon Graphics, Standard Template Library Programmer's Guide, http://www.sgi.com/Technology/STL/.

[12] T. Veldhuizen, Techniques for Scientific C++ - Version 0.3, Indiana University, Computer Science Department, 1999. (http://extreme.indiana.edu/ tveldhui/papers/techniques/)

[13] A. Griewank, D. Juedes, J. Utke, "Adol-C: A Package for the Automatic Differenciation of Algorithms Written in C/C++," ACM TOMS, vol. 22(2), pp. 131-167, June 1996.

[14] R. Pozo, MV++ v. 1.5a, Reference Guide, National Institute of Standards and Technology, 1997.

[15] M. Frigo and S. G. Johnson, FFTW User's Manual, Massachusetts Institute of Technology, September 1998.

[16] K. S. Kundert and A. Songiovanni-Vincentelli, Sparse user's guide - a sparse linear equation solver, Dept. of Electrical Engineering and Computer Sciences, University of California, Berkeley, Calif. 94720, Version 1.3a, Apr 1988.

[17] R. S. Bain, NNES user's manual, 1993.

[18] M. Valtonen and T. Veijola, "A microcomputer tool especially suited for microwave circuit design in frequency and time domain," Proc. URSI/IEEE National Convention on Radio Science, Espoo, Finland, 1986, p. 20,

[19] M. Valtonen, P. Heikkilä, A. Kankkunen, K. Mannersalo, R. Niutanen, P. Stenius, T. Veijola and J. Virtanen, "APLAC - A new approach to circuit simulation by object orientation," 10th European Conference on Circuit Theory and Design Dig., 1991.

[20] K. Mayaram and D. O. Pederson, "CODECS: an object-oriented mixed-level circuit and device simulator," 1987 IEEE Int. Symp. on Circuits and Systems Digest, 1987, pp 604-607.

[21] A. Davis, "An object-oriented approach to circuit simulation," 1996 IEEE Midwest Symp. on Circuits and Systems Dig., 1996, pp 313-316.

[22] B. Melville, P. Feldmann and S. Moinian, "A C++ environment for analog circuit simulation," 1992 IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors.

[23] P. Carvalho, E. Ngoya, J. Rousset and J. Obregon, "Object-oriented design of microwave circuit simulators," 1993 IEEE MTT-S Int. Microwave Symp. Digest, June 1993, pp 1491-1494.

[24] C. E. Christoffersen and M. B. Steer "Implementation of the local reference concept for spatially distributed circuits," Int. J. of RF and Microwave Computer-Aided Eng., vol. 9, No. 5, 1999.

[25] A. I. Khalil and M. B. Steer "Circuit theory for spatially distributed microwave circuits," IEEE Trans. on Microwave Theory and Techn., vol. 46, Oct. 1998, pp 1500-1503.

[26] C. E. Christoffersen, M. Ozkar, M. B. Steer, M. G. Case and M. Rodwell, "State variable-based transient analysis using convolution," IEEE Transactions on Microwave Theory and Techniques, Vol. 47, June 1999, pp. 882-889.

[27] C. E. Christoffersen, M. B. Steer and M. A. Summers, "Harmonic balance analysis for systems with circuit-field interactions," 1998 IEEE Int. Microwave Symp. Dig., June 1998, pp. 1131-1134.

[28] V. Rizzoli, A. Lipparini, A. Costanzo, F. Mastri, C. Ceccetti, A. Neri and D. Masotti, "State-of-the-art harmonic-balance simulation of forced nonlinear microwave circuits by the piecewise technique," IEEE Trans. on Microwave Theory and Techn., Vol. 40, Jan 1992, pp 12-27.

[29] B. Speelpenning. "Compiling Fast Partial Derivatives of Functions Given by Algorithms," Ph.D. thesis (Under the supervision of W. Gear), Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, Ill., January 1980.

[30] T. F. Coleman y G. F. Jonsson, "The Efficient Computation of Structured Gradients using Automatic Differentiation," Cornell Theory Center Technical Report CTC97TR272, April 28, 1997

[31] H. S. Tsai, M. J. W. Rodwell and R. A. York, "Planar amplifier array with improved bandwidth using folded-slots," IEEE Microwave and Guided Wave Letters, vol. 4, April 1994, pp. 112-114.

[32] M. B. Steer, M. N. Abdullah, C. Christoffersen, M. Summers, S. Nakazawa, A. Khalil, and J. Harvey, "Integrated electro-magnetic and circuit modeling of large microwave and millimeter-wave structures," Proc. 1998 IEEE Antennas and Propagation Symp., pp. 478–481, June 1998.

[33] M. N. Abdulla, U.A. Mughal, and M B. Steer, "Network Charactarization for a Finite Array of Folded-Slot Antennas for Spatial Power Combining Application," Proc. 1999 IEEE Antennas and Propagation Symp., July 1999.

[34] U. A. Mughal, "Hierarchical approach to global modeling of active antenna arrays," M.S. Thesis, North Carolina State University, 1999.

[35] Rational Software, UML Resources, http://www.rational.com/.

# List of Figures


1 The network package is the core of the simulator. . . . . 18
2 The **NetListItem** class. . . . . 19
3 The **Element** class. . . . . 20
4 Documentation generated for an element. . . . . 21
5 The **Circuit** class. . . . . 22
6 The **Xsubckt** class. . . . . 23
7 A CPW transmission line . . . . . 24
8 Implementation of an element using another element as a building block . . . 25
9 The analysis classes. . . . . 26
10 Dependency inversion was used to make the elements independent of the analysis classes. . . . . 27
11 Class diagram for an element using generic evaluation. . . . . 28
12 Nonlinear element function evaluation in convolution transient. . . . . 29
13 Implementation of automatic differentiation. . . . . 30
14 Unit cell of the CPW antenna array. . . . . 31
15 Circuit model of the unit cell. The diamond symbol indicates a local reference node. . . . . 32
16 Effective isotropic power gain as a function of frequency. . . . . 33
17 2x2 CPW antenna array. . . . . 34
18 2x2 CPW antenna array equivalent circuit. . . . . 35
19 Netlist for a 2x2 CPW antenna array. . . . . 36
20 Output currents for the 2x2 antenna array. . . . . 37
21 Notation for a class diagram. . . . . 38

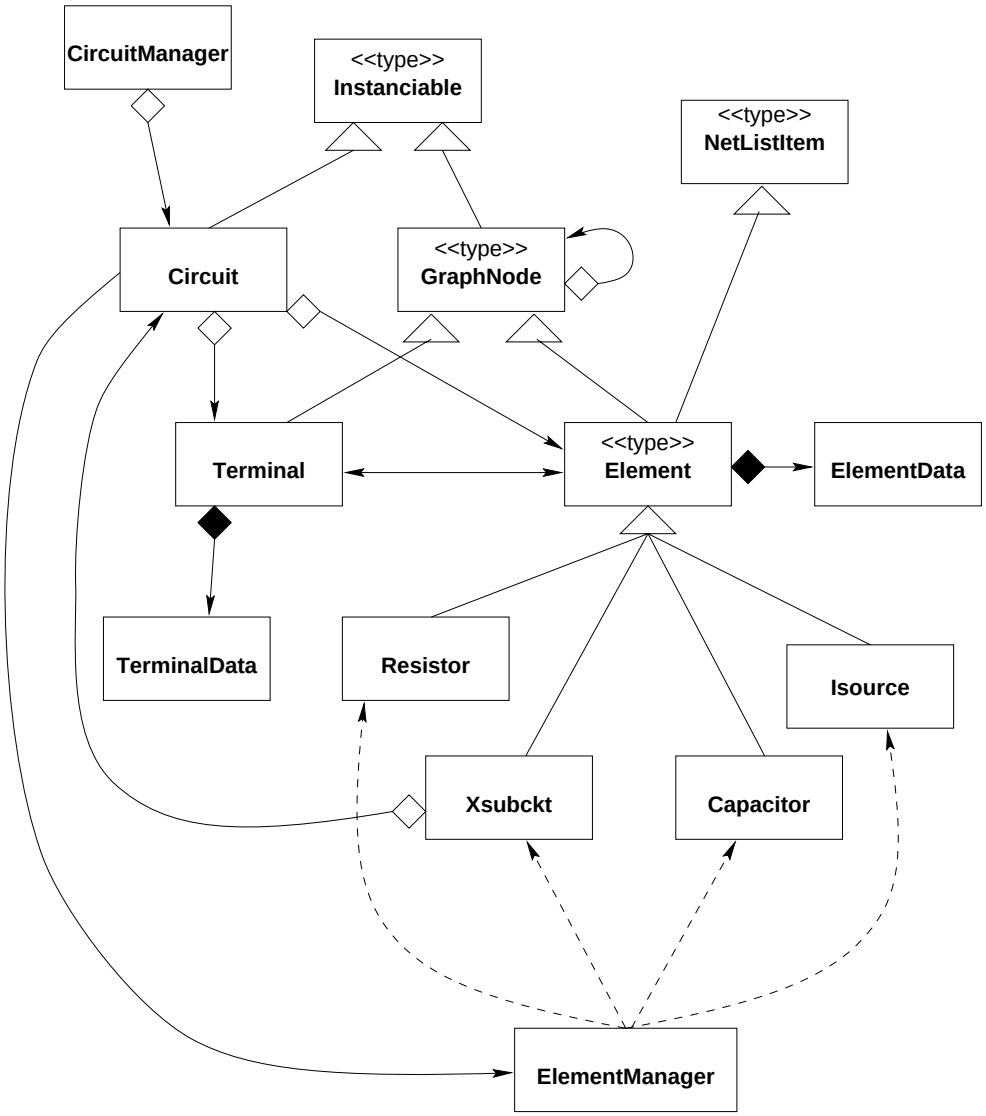


Figure 1: The network package is the core of the simulator.

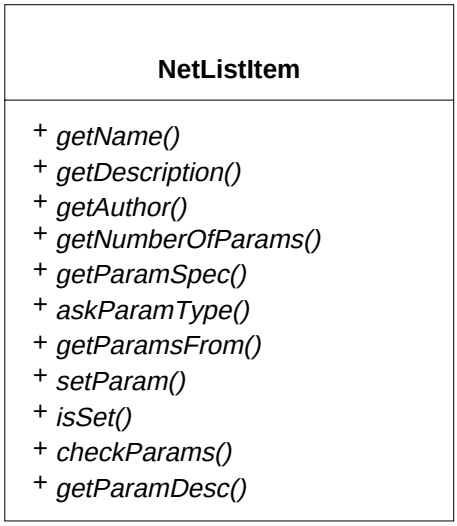


Figure 2: The **NetListItem** class.

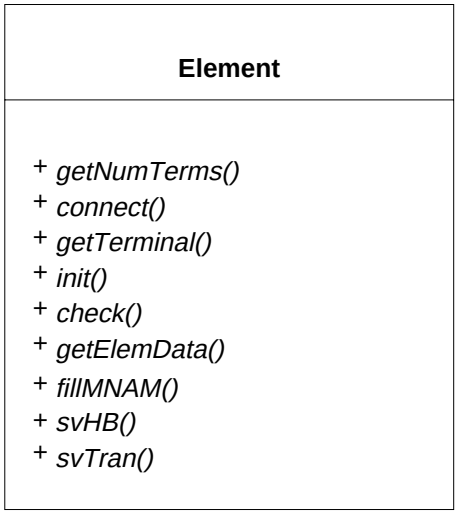


Figure 3: The **Element** class.

## 4 terminal physical transmission line

Authors: Carlos E. Christoffersen, Mete Ozkar

Multi-referenced element.

**Usage:**

**tlinp4**:<instance name> **n1 n2 n3 n4** <parameter list>

| Parameter | Type | Default value | Required? |
|---|---|---|---|
| **k:** Effective dielectric constant | DOUBLE | 1 | no |
| **alpha:** Attenuation (dB/m) | DOUBLE | 0.1 | no |
| **z0mag:** Magnitude of characteristic impedance (ohms) | DOUBLE | n/a | yes |
| **fscale:** Scaling frequency for attenuation (Hz) | DOUBLE | 0 | no |
| **tand:** Loss tangent | DOUBLE | 0 | no |
| **length:** Line length (m) | DOUBLE | n/a | yes |

Figure 4: Documentation generated for an element.

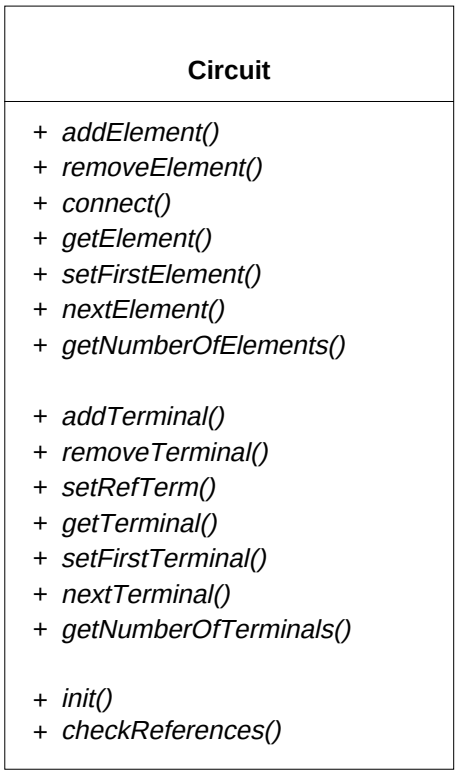


Figure 5: The **Circuit** class.

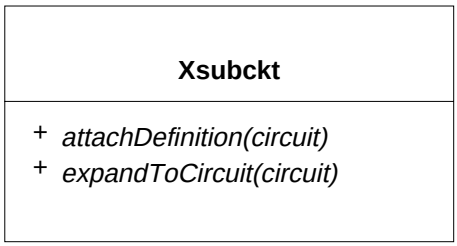


Figure 6: The **Xsubckt** class.

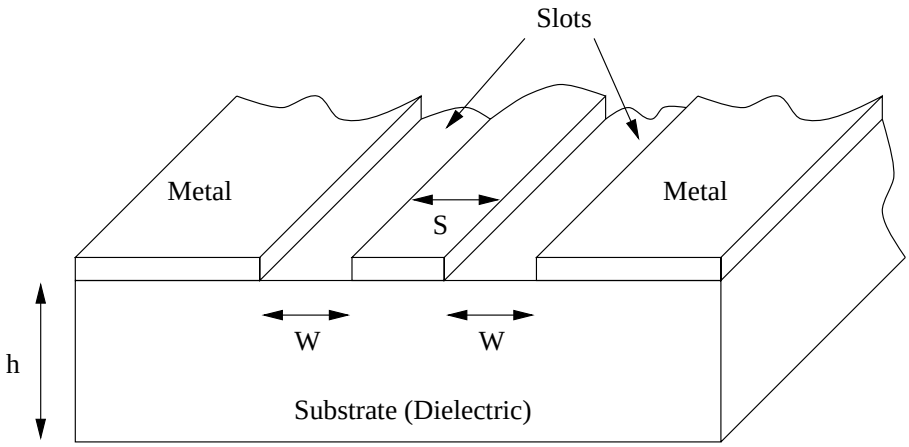


Figure 7: A CPW transmission line

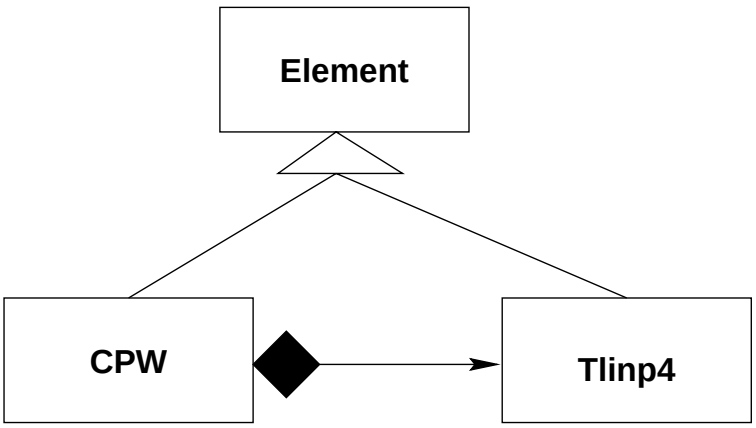


Figure 8: Implementation of an element using another element as a building block

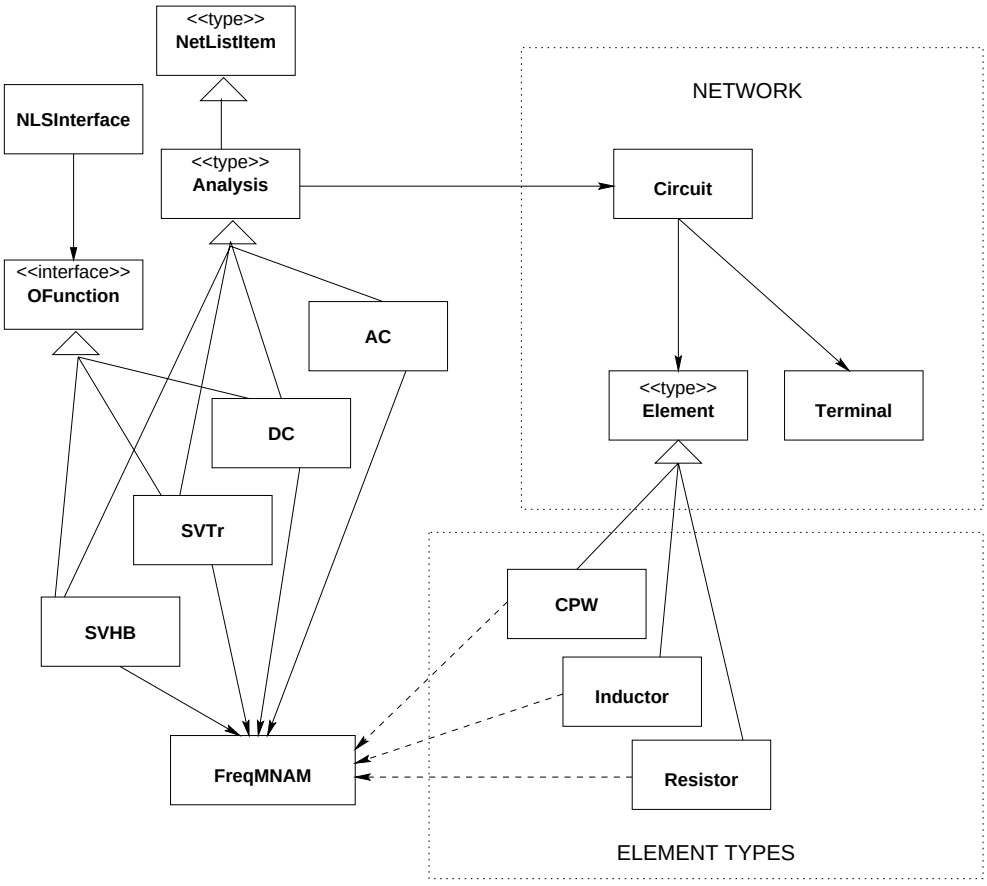


Figure 9: The analysis classes.

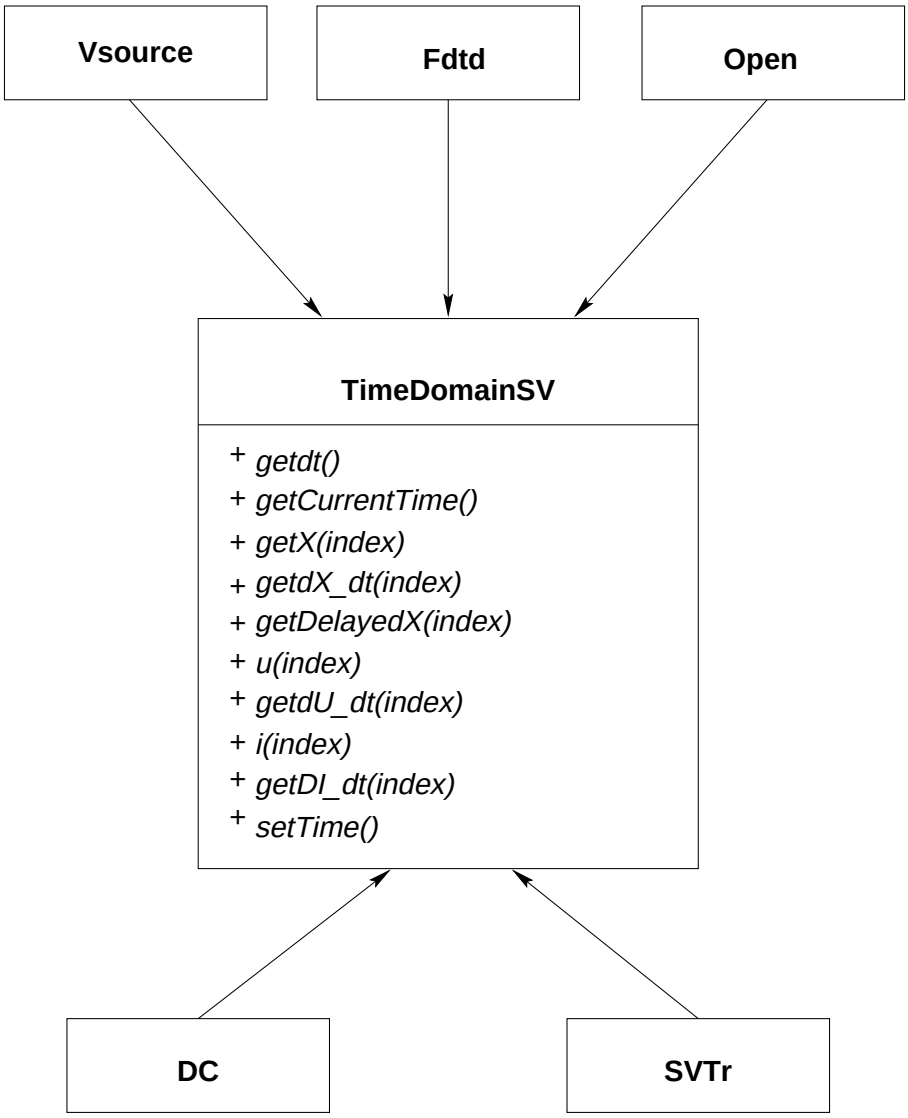


Figure 10: Dependency inversion was used to make the elements independent of the analysis classes.

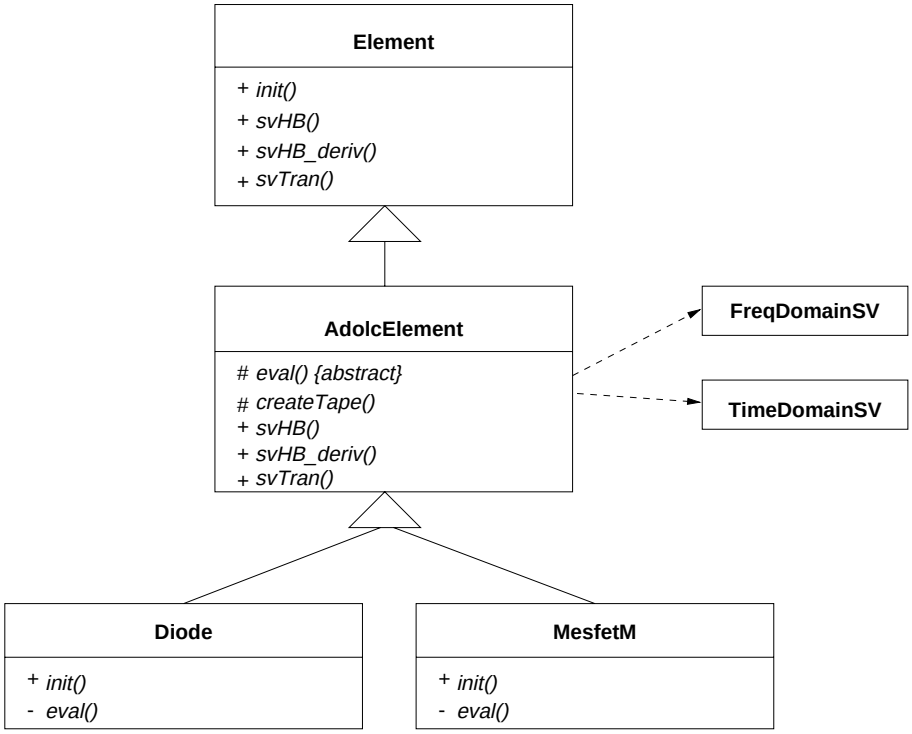


Figure 11: Class diagram for an element using generic evaluation.

```
// Number of elements
int n_elem = elem_vec.size();
int i = 0;
// Go through all the nonlinear elements
for (int k = 0; k < n_elem; k++) {
  // Set base index in interface object
  tdsv->setIBase(i);
  // nonlinear element evaluation
  elem_vec[k]->svTran(tdsv);
  i += elem_vec[k]->getNumberOfStates();
}
```

Figure 12: Nonlinear element function evaluation in convolution transient.

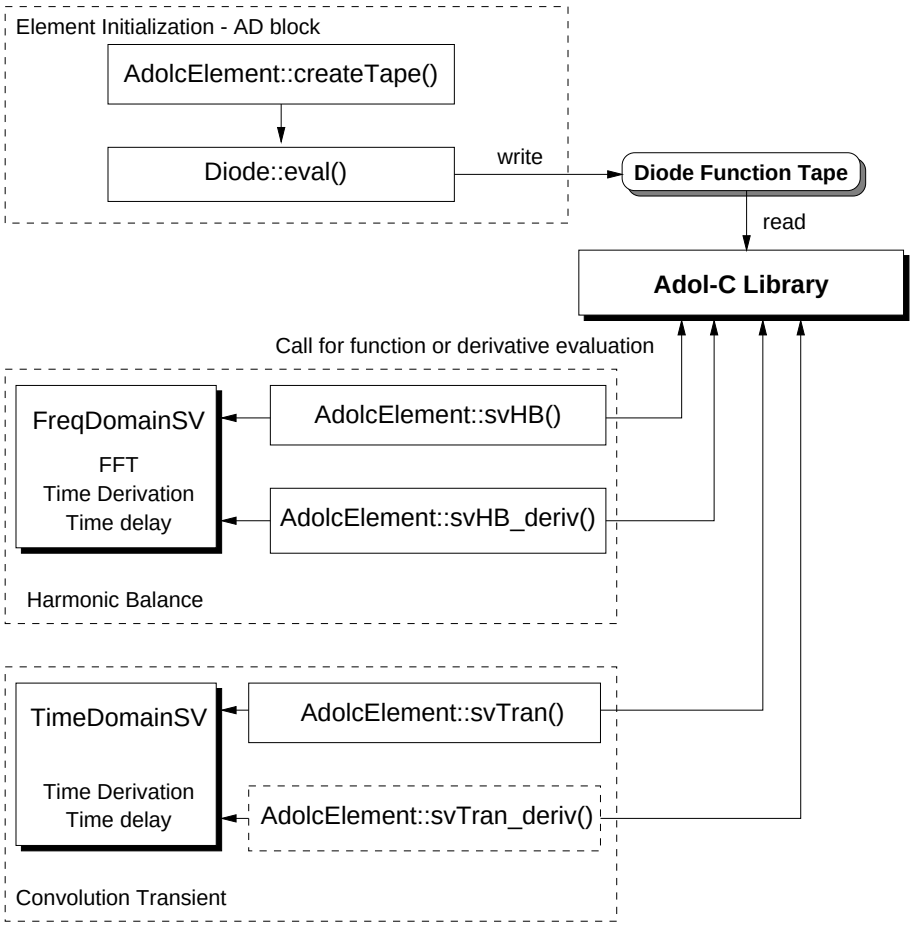


Figure 13: Implementation of automatic differentiation.

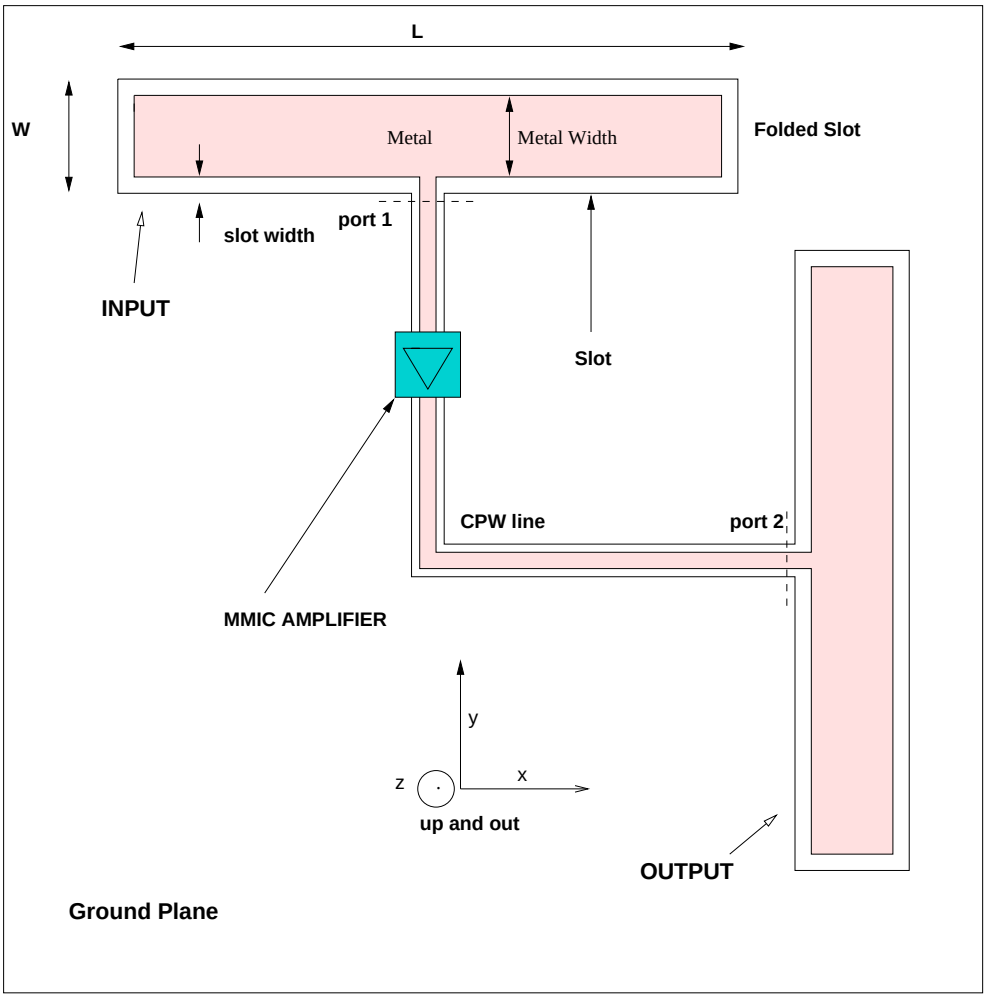


Figure 14: Unit cell of the CPW antenna array.

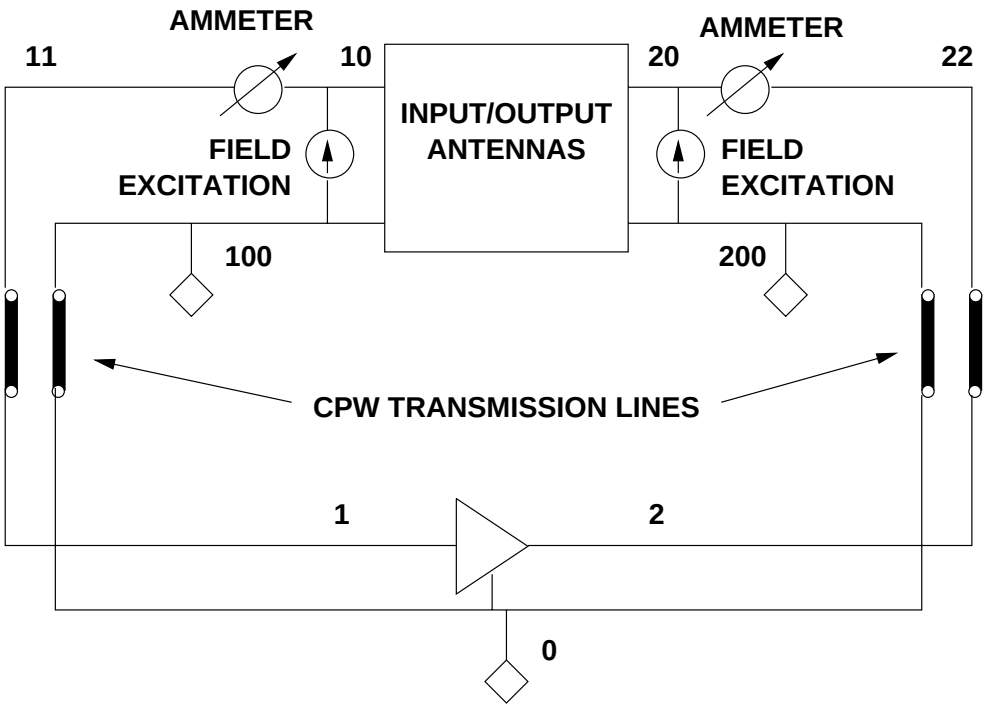


Figure 15: Circuit model of the unit cell. The diamond symbol indicates a local reference node.

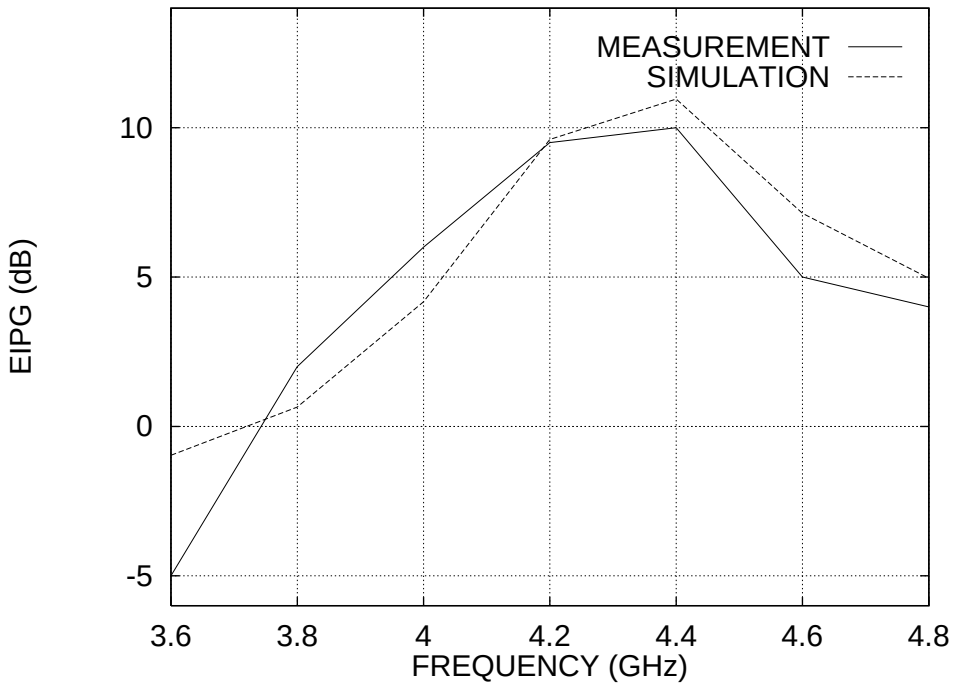


Figure 16: Effective isotropic power gain as a function of frequency.

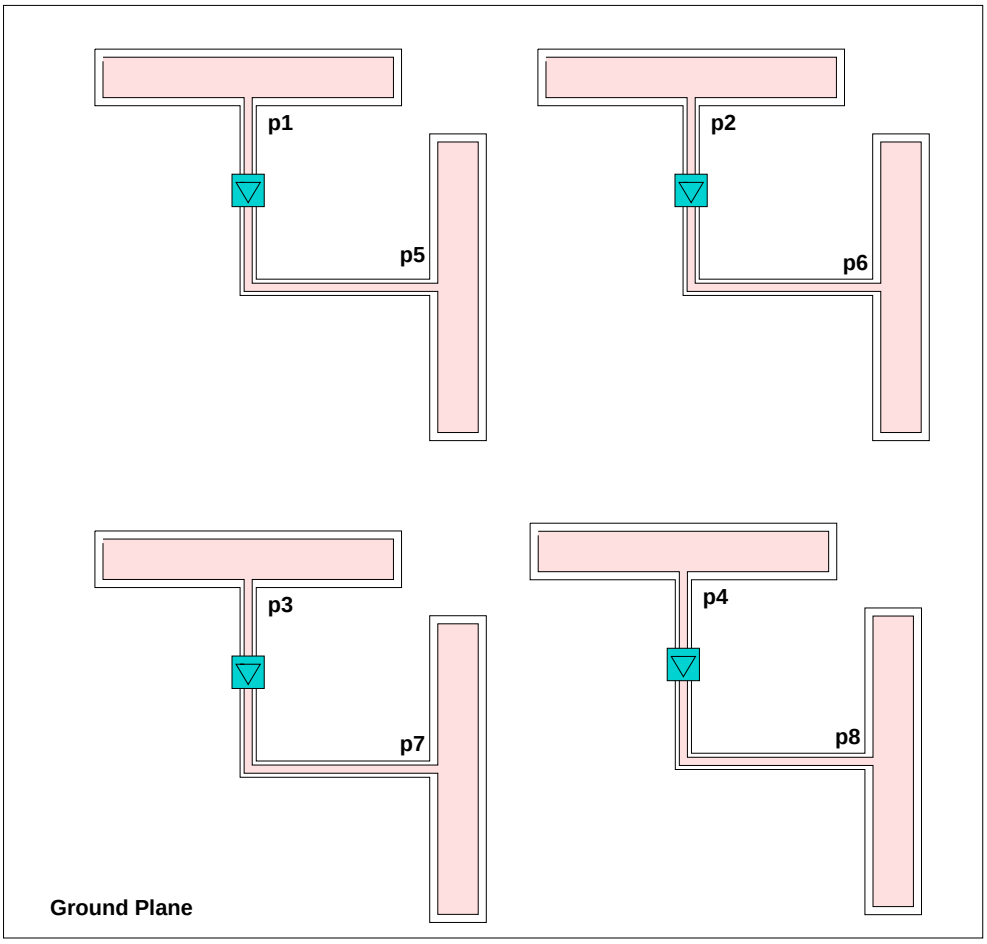


Figure 17: 2x2 CPW antenna array.

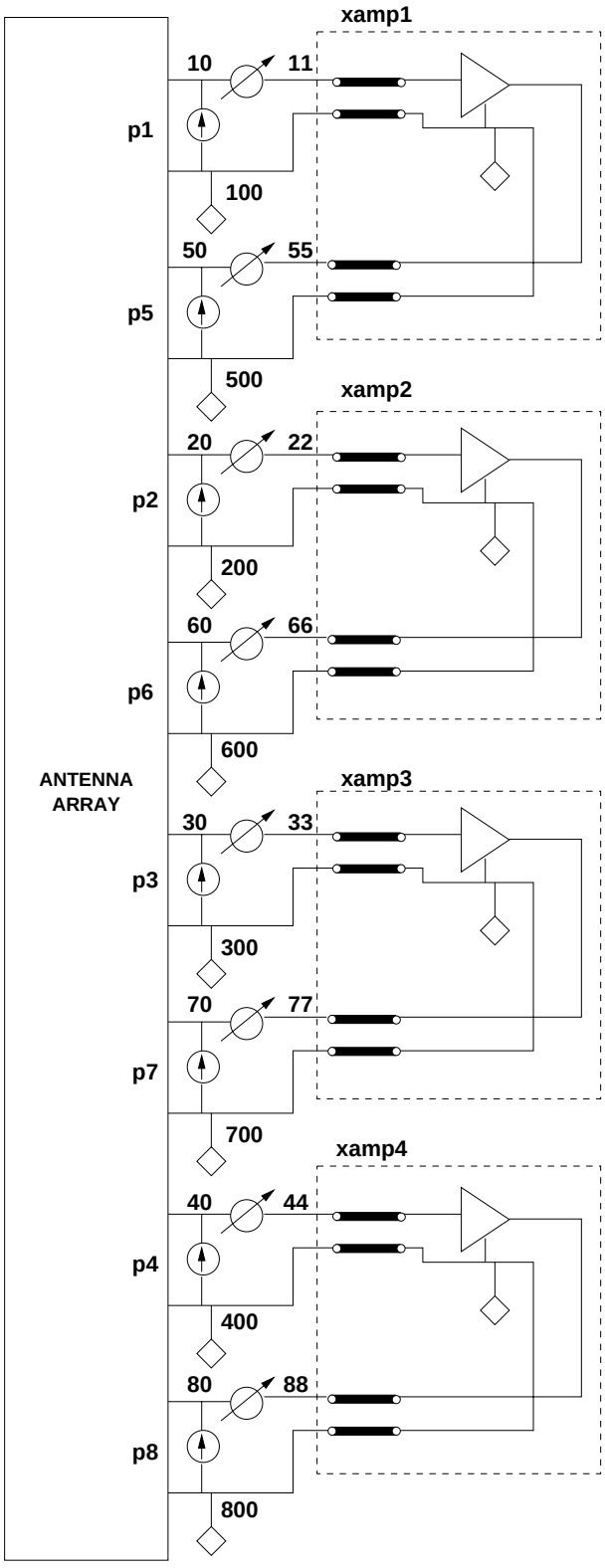


Figure 18: 2x2 CPW antenna array equivalent circuit.

```
.ac start = 3.6GHz stop = 5.0GHz n_freqs = 50
* Subcircuit with cpwlines and amplifier model
.subckt amp_lines 11 100 22 200
.ref "a_gnd"
* Transistor small signal model
nport:amplifier 1 2 "a_gnd" filename = "ne3210s1.yp"
* CPW Transmission lines
.model fsa1 cpw (s=.369m w=1m t=10u er=10.8 tand=.001)
cpw:t1 11 100 1 "a_gnd" model="fsa1" length=8.5m
cpw:t2 22 200 2 "a_gnd" model="fsa1" length=17.5m
.ends
* Local reference nodes
.ref 100
.ref 200
.ref 300
.ref 400
.ref 500
.ref 600
.ref 700
.ref 800
* Antenna array
nport:cpw_2 10 20 30 40 50 60 70 80
+ 100 200 300 400 500 600 700 800 filename = "2x2cell.yp"
* Field excitation
gridex:iin 10 100 20 200 30 300 40 400
+ 50 500 60 600 70 700 80 800 ifilename = "2x2cell.i"
+ efilename = "dummy.e"
* Amplifier instances
xamp1 11 100 55 500 amp_lines
xamp2 22 200 66 600 amp_lines
xamp3 33 300 77 700 amp_lines
xamp4 44 400 88 800 amp_lines
* Current meters
vsource:amp1 10 11
vsource:amp2 20 22
vsource:amp3 30 33
vsource:amp4 40 44
vsource:amp5 50 55
vsource:amp6 60 66
vsource:amp7 70 77
vsource:amp8 80 88
* Plot amplifier output currents
.out plot element "vsource:amp5" 0 if mag in "i5.outm"
.out plot element "vsource:amp6" 0 if mag in "i6.outm"
.out plot element "vsource:amp7" 0 if mag in "i7.outm"
.out plot element "vsource:amp8" 0 if mag in "i8.outm"
.end
```

Figure 19: Netlist for a 2x2 CPW antenna array.

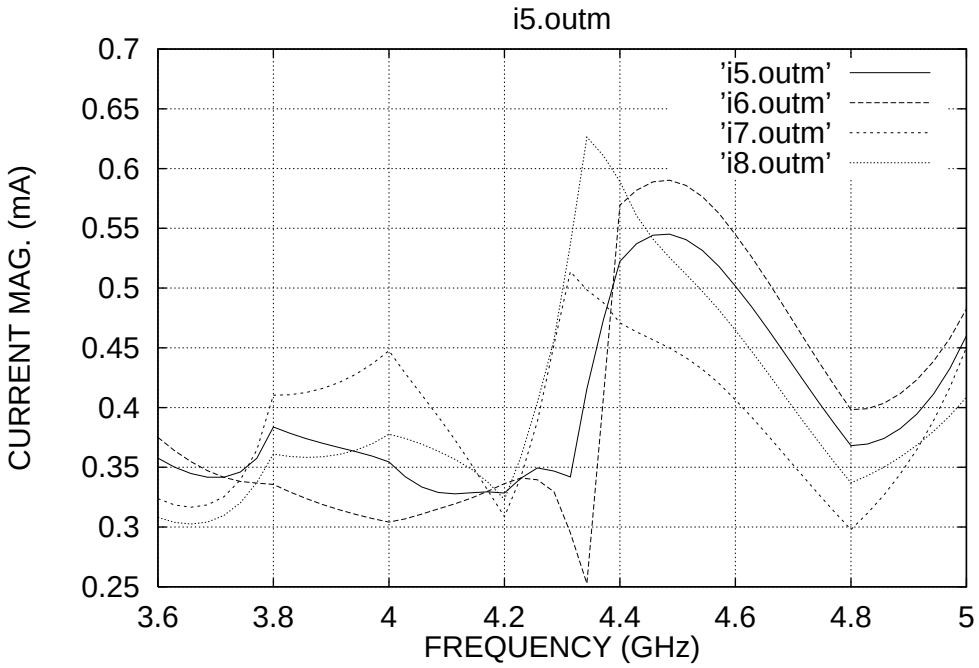


Figure 20: Output currents for the 2x2 antenna array.

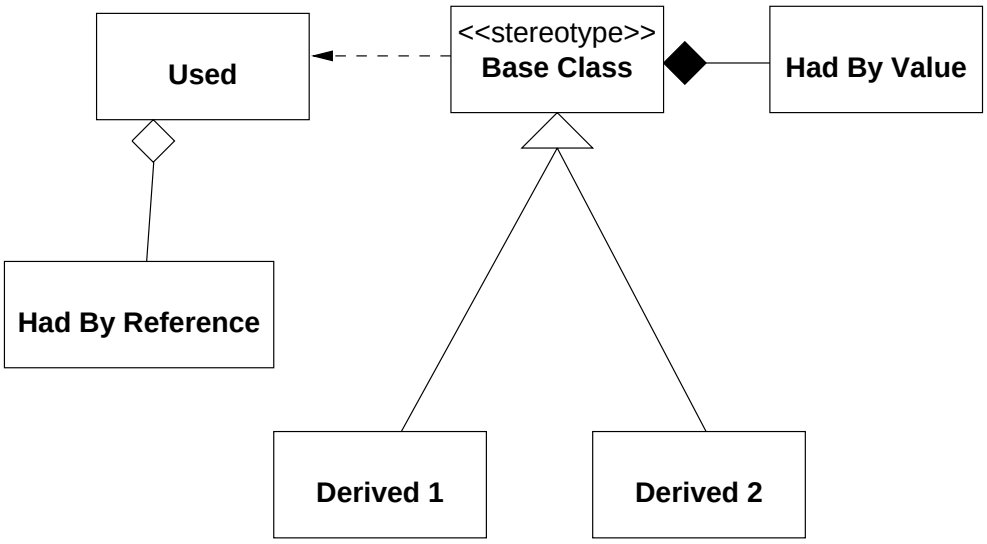


Figure 21: Notation for a class diagram.