Università degli Studi di Bergamo

SCUOLA DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

ltspice2circuitikz

Progetto di Linguaggi Formali e Compilatori
Documentazione tecnica

Prof. Giuseppe Psaila

Dott. Paolo Fosci

Candidati
Davide Salvetti
Matricola 1057596

Matteo Verzeroli
Matricola 1057926

Anno Accademico 2022–2023

1 Introduzione

Il seguente progetto ha come obiettivo quello di realizzare un'applicazione che consenta di ricavare automaticamente la rappresentazione in LaTeX di un circuito elettronico realizzato con il software di simulazione LTSpice (Analog Devices), utilizzando il pacchetto CircuiTikz che permette la visualizzazione di un circuito in un documento latex. Infatti, LTSpice produce un file con estensione .asc che contiene tutte le informazioni utili alla rappresentazione grafica dei componenti, come ad esempio la posizione assoluta dei vari elementi circuitali, delle eventuali label associate alle connessioni e i valori dei vari componenti. L'idea alla base di questo progetto è quella di utilizzare le informazioni contenute nel file .asc e il pacchetto CircuiTikz per ottenere una rappresentazione in latex del circuito. Una volta ottenuto il file .tex, si utilizza lo strumento `pdflatex` (distribuito con il compilatore MikTex) per generare un documento pdf.

1.1 Tecnologie utilizzate

Per lo sviluppo di questo progetto si sono utilizzate le seguenti tecnologie:

- `ANTLRWorks 1.5.2`, per la generazione del lexer e del parser nel linguaggio Java;
- libreria Java `antlr-complete` versione 3.4;
- pacchetto `CircuiTikz` e compilatore `MikTex`, per la generazione del codice latex e del documento pdf;
- `Eclipse IDE`, per la creazione del progetto Java del lexer e del parser;
- framework Qt, per la creazione dell'app client;
- `GitHub`, per il versioning del codice.

1.2 LTSpice

LTSpice è un software molto utilizzato in ambito accademico e lavorativo per effettuare simulazioni di circuiti elettrici. L'interazione con il programma avviene tramite un'interfaccia grafica che permette di inserire i componenti, assegnare loro un valore e un identificativo e definire i parametri di simulazione (Fig.1).

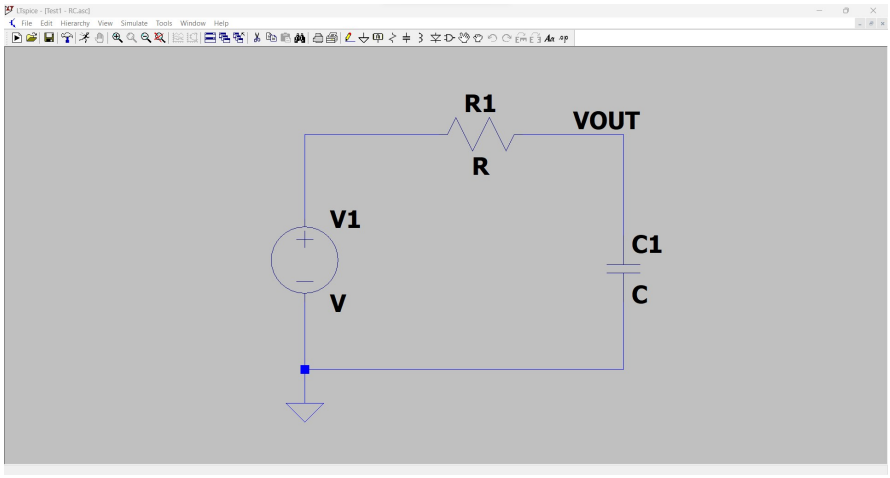

Figura 1: Schermata principale di LTSpice.

Il motore di simulazione è basato sul programma SPICE (*Simulation Program with Integrated Circuit Emphasis*) che riceve in input una `netlist`, ossia una rappresentazione testuale del circuito, e produce in output i risultati di simulazione. LTSpice crea automaticamente la `netlist` a partire dalla rappresentazione grafica del circuito. Per esempio, il circuito in figura 1 genera la seguente netlist:

```
V1 N001 0 V
R1 VOUT N001 R
C1 VOUT 0 C
.backanno
.end
```

Inoltre, il programma salva il circuito in un file con estensione .asc che contiene tutte le informazioni per rappresentare in modo grafico il circuito. Di seguito si riporta il file .asc prodotto dal circuito in figura 1.

```
Version 4
SHEET 1 880 1280
WIRE 160 400 32 400
WIRE 320 400 240 400
WIRE 336 400 320 400
WIRE 32 480 32 400
WIRE 336 496 336 400
WIRE 32 624 32 560
WIRE 336 624 336 560
WIRE 336 624 32 624
WIRE 32 656 32 624
FLAG 32 656 0
FLAG 320 400 VOUT
SYMBOL voltage 32 464 R0
SYMATTR InstName V1
SYMBOL res 256 384 R90
WINDOW 0 0 56 VBottom 2
WINDOW 3 32 56 VTop 2
SYMATTR InstName R1
SYMBOL cap 320 496 R0
SYMATTR InstName C1
```

Come si può notare, in questo file non vengono riportate informazioni sulla simulazione da eseguire ma bensì la posizione dei collegamenti, dei componenti, delle label ed eventuali attributi. In particolare, le principali parole chiave che si possono riconoscere sono:

- VERSION: rappresenta la versione del sistema utilizzato; per il progetto verrà considerata solamente la versione 4;
- SHEET: definisce il numero del foglio di lavoro utilizzato e le relative dimensioni;
- WIRE: indica un filo di collegamento; seguono le coordinate x,y assolute dei relativi capi;
- SYMBOL: rappresenta un componente; di seguito si riporta la tipologia, la posizione assoluta in coordinate x,y e l'eventuale rotazione o mirror; è possibile indicare una rotazione di 0°, 90°, 180°, 270° o un'operazione di mirror di 0°, 90°, 180° e 270°;
- SYMATTR: permette di indicare eventuali attributi riferiti a un simbolo come il nome, il valore, i parametri dei componenti parassiti o indicare il produttore del componente;
- WINDOW: indica la posizione della label associata a un componente (il nome o il valore) nel caso di rotazioni e mirror;
- FLAG: definisce la posizione e il contenuto di eventuali label associati a dei collegamenti in un circuito; viene utilizzata anche per indicare la massa (indicando come nome della label 0);
- IOPIN: indicano una label di ingresso o uscita nel circuito.

Tutte le coordinate sono espresse attraverso numeri interi e le parole chiave sono interpretate in modo *case insensitive*. Il formato di codifica del file è `ISO-8859-1` e accetta qualsiasi carattere `UNICODE` all'interno delle label e dei nomi dei componenti. Le parole chiave e i relativi parametri sono separati da degli spazi bianchi. LTSpice è insensibile al numero di spazi tra i diversi token e di eventuali caratteri di *newline*. Tuttavia, il file .asc in output a LTSpice è formattato mantenendo un unico spazio tra ogni token e andando a capo prima di ogni parola chiave.

2 ltspice2circuitikz

L'applicazione finale realizzata in questo progetto si suddivide in due principali componenti:

- un'interfaccia grafica, realizzata con il framework Qt;
- un'applicazione Java che realizza le operazioni necessarie per la traduzione, distribuita come un .jar eseguibile.

In figura 2 è mostrato il deploy diagram dell'intera applicazione.

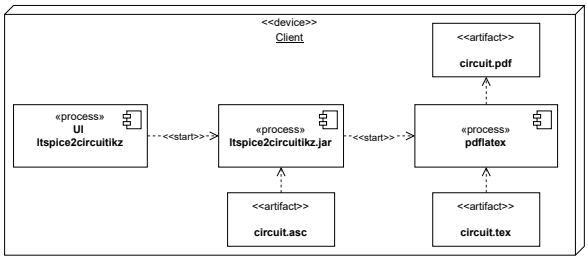

Figura 2: Deploy diagram del sistema.

L'interfaccia utente (`UI ltspice2circuitikz`) avvia l'applicativo jar (`ltspice2circuitikz.jar`), il quale richiede in input il file .asc contente la specifica del circuito realizzata con LTSpice. A sua volta, l'applicazione Java, una volta terminata la traduzione, avvia il processo `pdflatex` che richiede in input il file .tex ottenuto dalla traduzione e restituisce il pdf con la rappresentazione del circuito.
In figura 3 viene mostrato il diagramma delle classi e dei package dell'applicazione Java. Il metodo `main` richiamato all'avvio dell'applicazione è contenuto nella classe `ParserTester`, la quale contiene i riferimenti al lexer e al parser generato da `ANTLRWorks`. Inoltre, il parser contiene il riferimento alla classe `Handler` che fornisce i metodi per eseguire le azioni semantiche. Il package `com.omega.compiler.util` contiene alcune classi utilizzate dalle azioni semantiche per descrivere un componente (classe `Component`), un collegamento (`classe Wire`), una label (classe `Flag`), un nodo (classe `Point`) e gli errori. La classe `LatexConverter` si occupa invece di effettuare le operazioni di traduzione e di creazione del file latex. I metodi contenuti vengono richiamati dalla classe `Handler`.

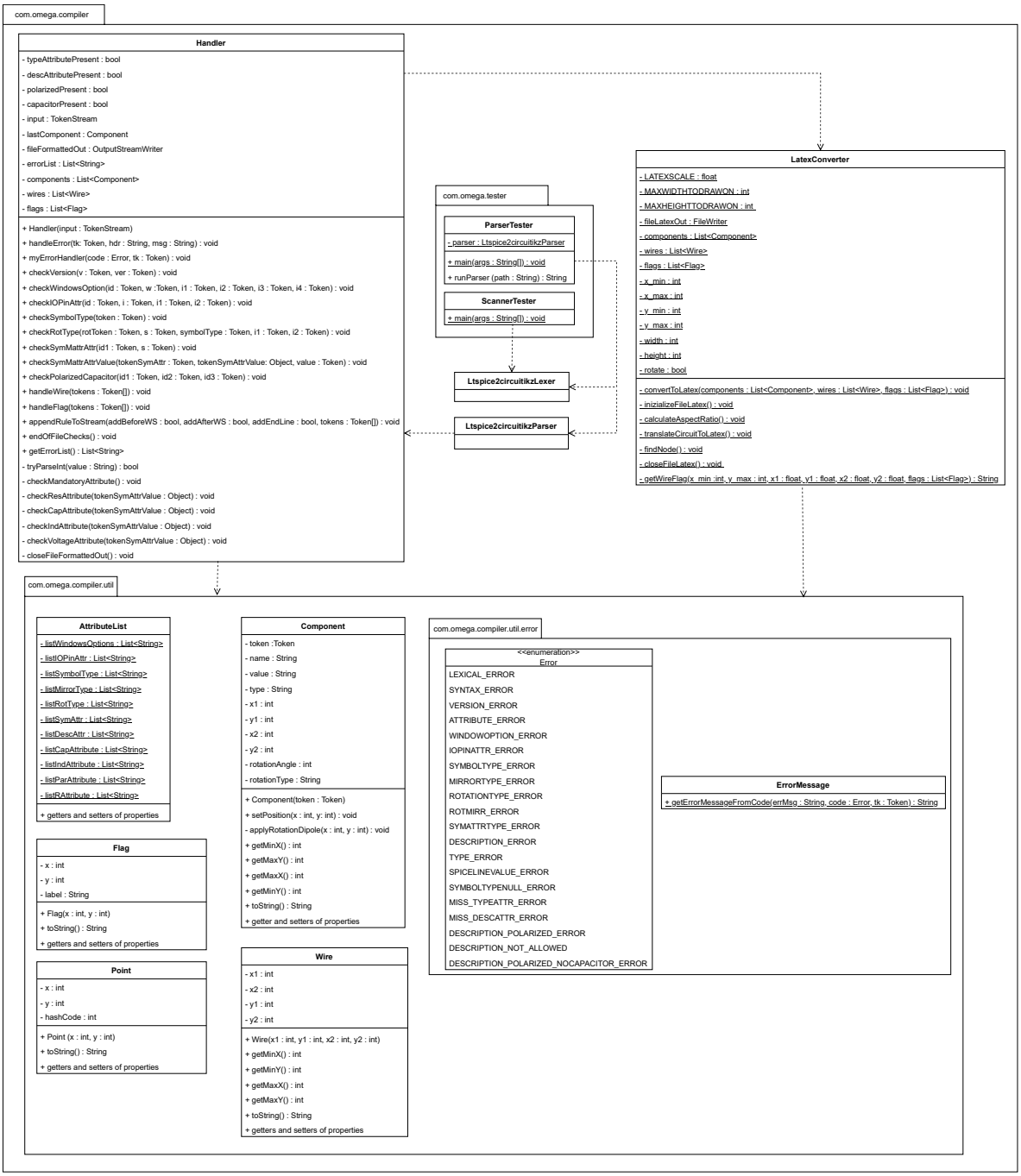

Figura 3: Class and Package Diagram dell'applicazione.

Lexer e Parser

L'applicazione realizzata si basa su un lexer e un parser LL(1) generato nel linguaggio Java grazie al tool `AntlrWorks`. Essi permettono di leggere ed interpretare un file .asc e di effettuare la traduzione in un file .tex tramite delle azioni semantiche. Per definire la grammatica utile al parsing del file .asc è stato eseguito un processo di *reverse engeneering*, analizzando il file generato da LTSpice con dei circuiti di esempio. Di seguito si riportano i token identificati e le regole sintattiche ricavate dall'analisi dei file .asc.

```
parseCircuit
  :
    prologueRule
    componentRule*
    EOF
  ;
prologueRule
  : versionRule
    sheetRule
  ;
versionRule
  :
    v=VERSION
    ver = INTEGER
  ;

sheetRule
  :
    s=SHEET i1=INTEGER i2=INTEGER i3=INTEGER
  ;

componentRule
  : wireRule
    | symbol = symbolRule
    | symattrRule
    | flagRule
    | windowRule
    | iopinRule
  ;
wireRule
  :
    w=WIRE i1=INTEGER i2=INTEGER i3=INTEGER i4=INTEGER
  ;

flagRule
  :
    f=FLAG i1=INTEGER i2=INTEGER v=(INTEGER | ID | reservedWordRule)
  ;

windowRule
  :
    w=WINDOW
    i1=INTEGER
    i2=INTEGER
    i3=INTEGER
    id = ID
    i4=INTEGER
  ;
iopinRule
  :
    i=IOPIN
    i1=INTEGER
    i2=INTEGER
    id = ID
  ;
```

```
symbolRule returns[Token symbol]
  :
    s=SYMBOL
    symbolType = ID
    i1=INTEGER
    i2=INTEGER
    rotType = ID
  ;
symattrRule
  :
        s=SYMATTR id1=ID
      ( id2=ID
        ( id3=ID
        | (attrRuleNoId attrRule[id1]*)?)
      | i=INTEGER
      | f=FLOAT
      | r = reservedWordRule)
  ;
attrRuleNoId
  :
    a=ASSIGN
    v=(INTEGER | FLOAT | STRING | ID | reservedWordRule)
  ;
attrRule [Token id1]
  :
    id2 = ID
    a=ASSIGN
    v=(INTEGER | FLOAT | STRING | ID | reservedWordRule)
  ;

reservedWordRule returns [Token word]
  :
    v=(VERSION | SHEET | WIRE | FLAG | WINDOW | SYMBOL | SYMATTR | ASSIGN | IOPIN)
  ;

fragment
LETTER : 'a'..'z'|'A'..'Z';
fragment
DIGIT : '0'..'9';
fragment
EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;
fragment
SPECIALCHAR: '\u0021'..'\u002F' |'\u003A'..'\u003C'|'\u003E'..'\u0040'|'\u005B'..'\
    u0060' //punctuation and symbols. '=' removed for STRING TOKEN
       |'\u007B'..'\u007E'|'\u00A1'..'\u017F' //latin punctuation and symbols
       |'\u0370'..'\u03FF' //Greek alphabet
       |'\u0400'..'\u04FF' //Cyrillic alphabet
       ;

/* case insensitive lexer matching */
fragment A:('a'|'A');
fragment B:('b'|'B');
fragment C:('c'|'C');
fragment D:('d'|'D');
fragment E:('e'|'E');
fragment F:('f'|'F');
fragment G:('g'|'G');
fragment H:('h'|'H');
fragment I:('i'|'I');
fragment J:('j'|'J');
fragment K:('k'|'K');
fragment L:('l'|'L');
fragment M:('m'|'M');
fragment N:('n'|'N');
fragment O:('o'|'O');
```

```
fragment P:('p'|'P');
fragment Q:('q'|'Q');
fragment R:('r'|'R');
fragment S:('s'|'S');
fragment T:('t'|'T');
fragment U:('u'|'U');
fragment V:('v'|'V');
fragment W:('w'|'W');
fragment X:('x'|'X');
fragment Y:('y'|'Y');
fragment Z:('z'|'Z');

INTEGER:    ('-')?DIGIT+;
FLOAT
    :   DIGIT+ '.' DIGIT* EXPONENT?
    |   '.' DIGIT+ EXPONENT?
    |   DIGIT+ EXPONENT
    ;


//KEYWORD
VERSION : V E R S I O N ;
SHEET : S H E E T;
WIRE  : W I R E;
SYMBOL  : S Y M B O L;
SYMATTR : S Y M A T T R;
ASSIGN  : '=';
WINDOW  : W I N D O W;
FLAG  : F L A G;
IOPIN : I O P I N;

WS  :   ( ' '
        | '\t'
        | '\r'
        | '\n'
        )+ {$channel=HIDDEN;}
    ;

STRING  : '"' ~('"')* '"';

ID  : (LETTER | DIGIT | SPECIALCHAR)(LETTER | DIGIT | SPECIALCHAR)*;

ERROR_TK    : . ;
```

In particolare, si è notato che il programma non generava errore all'apertura del file se:

- le parole chiave erano scritte in caratteri minuscoli o in una qualsiasi combinazione di caratteri minuscoli e maiuscoli;
- venivano aggiunti degli spazi bianchi o dei caratteri di terminazione di riga tra le parole;

Per mantenere coerente la grammatica con questo comportamento, è stato considerato come separatore tra i diversi token il carattere di spazio bianco, di terminazione di riga e di tab. Inoltre, le parole chiave sono state rese invarianti rispetto al carattere delle lettere (lower o upper case). La grammatica completa si compone di nove parole chiavi: VERSION, SHEET, WIRE, SYMBOL, SYMATTR, simbolo di assegnamento (=), WINDOW e IOPIN. Ulteriori controlli sulle possibili strutture ammesse sono state inserite nella semantica, discussa successivamente. Inoltre, LTSpice accetta l'inserimento di qualunque carattere `UNICODE` nelle label dei componenti. Per questo motivo sono stati aggiunti i token per il riconoscimento dei caratteri speciali, dell'alfabeto greco e dell'alfabeto cirillico.

2.1 Semantica

Per gestire le azioni semantiche è stata predisposta la classe `Handler` che racchiude la definizione dei metodi utilizzati per i controlli semantici e le operazioni di traduzione. Di seguito si riporta la grammatica decorata con le operazioni semantiche.

```
parseCircuit
@init {initParser();}
  :
    prologueRule
    componentRule*
    EOF {h.endOfFileChecks();}
  ;
prologueRule
  : versionRule
    sheetRule
  ;
versionRule
  :
    v=VERSION
    ver = INTEGER {h.checkVersion(v, ver);}
  ;

sheetRule
  :
    s=SHEET i1=INTEGER i2=INTEGER i3=INTEGER {h.appendRuleToStream(false, true, true,
    s, i1, i2, i3);}
  ;

componentRule
  : wireRule
    | symbol = symbolRule
    | symattrRule
    | flagRule
    | windowRule
    | iopinRule
  ;
wireRule
  :
    w=WIRE i1=INTEGER i2=INTEGER i3=INTEGER i4=INTEGER {h.handleWire(w, i1, i2, i3, i4
    );}
  ;

flagRule
  :
    f=FLAG i1=INTEGER i2=INTEGER v=(INTEGER | ID | reservedWordRule) {h.handleFlag(f,
    i1, i2, v);}
  ;

windowRule
  :
    w=WINDOW
    i1=INTEGER
    i2=INTEGER
    i3=INTEGER
    id = ID
    i4=INTEGER
    {h.checkWindowsOptions(id, w, i1, i2, i3, i4);}
  ;
iopinRule
  :
    i=IOPIN
    i1=INTEGER
    i2=INTEGER
    id = ID
```

```
    {h.checkIOPinAttr(id, i, i1, i2);}

  ;
symbolRule returns[Token symbol]
  :
    s=SYMBOL
    symbolType = ID {h.checkSymbolType(symbolType); symbol = symbolType;}
    i1=INTEGER
    i2=INTEGER
    rotType = ID  {h.checkRotType(rotType, s, symbolType, i1, i2);}
  ;
symattrRule
  :
        s=SYMATTR id1=ID {h.checkSymMattrAttr(id1, s);}
      ( id2=ID {h.checkSymMattrAttrValue(id1, id2, null);}
        ( id3=ID {h.checkPolarizedCapacitor(id1, id2, id3);}
        | (attrRuleNoId attrRule[id1]*)?)
      | i=INTEGER {h.checkSymMattrAttrValue(id1, "int", i);}
      | f=FLOAT {h.checkSymMattrAttrValue(id1, "float", f);}
      | r = reservedWordRule {h.checkSymMattrAttrValue(id1, "reserved", r);})
    {h.appendRuleToStream(false, false, true);}
  ;
attrRuleNoId
  :
    a=ASSIGN
    v=(INTEGER | FLOAT | STRING | ID | reservedWordRule)
    {h.appendRuleToStream(false, false, false, a, v);}
  ;
attrRule [Token id1]
  :
    id2 = ID {h.checkSymMattrAttrValue(id1, id2, null);}
    a=ASSIGN
    v=(INTEGER | FLOAT | STRING | ID | reservedWordRule)
    {h.appendRuleToStream(false, false, false, a, v);}

  ;

reservedWordRule returns [Token word]
  :
    v=(VERSION | SHEET | WIRE | FLAG | WINDOW | SYMBOL | SYMATTR | ASSIGN | IOPIN)
    {word = v; h.appendRuleToStream(true, false, false, v);}
  ;

fragment
LETTER : 'a'..'z'|'A'..'Z';
fragment
DIGIT : '0'..'9';
fragment
EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;
fragment
SPECIALCHAR: '\u0021'..'\u002F' |'\u003A'..'\u003C'|'\u003E'..'\u0040'|'\u005B'..'\
    u0060' //punctuation and symbols. '=' removed for STRING TOKEN
       |'\u007B'..'\u007E'|'\u00A1'..'\u017F' //latin punctuation and symbols
       |'\u0370'..'\u03FF' //Greek alphabet
       |'\u0400'..'\u04FF' //Cyrillic alphabet
       ;

/* case insensitive lexer matching */
fragment A:('a'|'A');
fragment B:('b'|'B');
fragment C:('c'|'C');
fragment D:('d'|'D');
fragment E:('e'|'E');
fragment F:('f'|'F');
fragment G:('g'|'G');
```

```
fragment H:('h'|'H');
fragment I:('i'|'I');
fragment J:('j'|'J');
fragment K:('k'|'K');
fragment L:('l'|'L');
fragment M:('m'|'M');
fragment N:('n'|'N');
fragment O:('o'|'O');
fragment P:('p'|'P');
fragment Q:('q'|'Q');
fragment R:('r'|'R');
fragment S:('s'|'S');
fragment T:('t'|'T');
fragment U:('u'|'U');
fragment V:('v'|'V');
fragment W:('w'|'W');
fragment X:('x'|'X');
fragment Y:('y'|'Y');
fragment Z:('z'|'Z');

INTEGER:    ('-')?DIGIT+;
FLOAT
    :   DIGIT+ '.' DIGIT* EXPONENT?
    |   '.' DIGIT+ EXPONENT?
    |   DIGIT+ EXPONENT
    ;


//KEYWORD
VERSION : V E R S I O N ;
SHEET : S H E E T;
WIRE  : W I R E;
SYMBOL  : S Y M B O L;
SYMATTR : S Y M A T T R;
ASSIGN  : '=';
WINDOW  : W I N D O W;
FLAG  : F L A G;
IOPIN : I O P I N;

WS  :   ( ' '
        | '\t'
        | '\r'
        | '\n'
        )+ {$channel=HIDDEN;}
    ;

STRING  : '"' ~('"')* '"';

ID  : (LETTER | DIGIT | SPECIALCHAR)(LETTER | DIGIT | SPECIALCHAR)*;

ERROR_TK    : . ;
```

Nel costruttore della classe `Handler` vengono dapprima inizializzate alcune strutture dati che verranno utilizzate per gestire la traduzione. In particolare, vengono create le liste `components`, `wires` e `flags` che servono a memorizzare i componenti letti in modo incrementale durante l'analisi del file in input; viene inizializzata anche una lista che conterrà gli errori lessicali, sintattici e semantici. Inoltre, viene creata una cartella con il nome `circuit_output` e un file denominato `formatted_circuit.asc` che conterrà la descrizione del circuito in formato LTSpice formattato correttamente. Infatti, come già descritto in precedenza, l'inserimento di spazi bianchi e caratteri di terminazione di linea non precludono la possibilità di interpretare correttamente il file. Per questo motivo, ogni qualvolta viene letto un token o un'insieme di token essi sono accodati al file `formatted_circuit.asc` grazie alla funzione `appendRuleToStream` dichiarata nell'handler. Questa funzione ha come argomenti l'insieme dei token da scrivere sul file e alcuni parametri booleani per descrivere come devono essere formattati nel file

(aggiungere uno spazio prima o dopo ogni token e al termine della riga scritta il carattere di \n).
Un altro campo importante nella classe `Handler` è `lastComponent`. Esso è di tipo `Component`, una classe contenuta nel package `com.omega.compiler.util` che contiene tutte le informazioni relative a un componente. Questa variabile è utilizzata per mantenere memoria dell'ultimo componente letto: in questo modo, è possibile poi effettuare dei controlli sulle regole che vengono lette successivamente e che si riferiscono a quel componete. Essa viene inizializzata nel metodo `checkSymbolType` chiamato quando si incontra la parola chiave `SYMBOL` nella regola sintattica `symbolRule`.
Al termine della lettura di tutto il contenuto del file .asc in input, se non ci sono stati errori, viene chiamato il metodo statico `convertToLatex` della classe `LatexConverter` che inizia il processo di traduzione. Inoltre, si procede alla chiusura del file `formatted_circuit.asc`.

2.2 Traduzione

La classe `LatexConverter` contiene i metodi necessari alla traduzione del circuito per la rappresentazione in Latex. Il metodo `convertToLatex`, chiamato dall'handler, riceve in ingresso la lista dei componenti, dei collegamenti e delle etichette ricavate dal file in ingresso. Successivamente, viene chiamato il metodo `calculateAspectRatio` che consente di calcolare i valori massimi e minimi delle coordinate x e y di tutti gli elementi del circuito e viene calcolata l'altezza e la larghezza del circuito nel sistema di riferimento di LTSpice. Se l'altezza è minore della larghezza allora il circuito verrà rappresentato su un foglio in modalità *landscape*. Nella figura 4 sono mostrati i sistemi di riferimento utilizzati in LTSpice e in Latex. Non è conosciuta la posizione assoluta del sistema di riferimento in LTSpice rispetto alla finestra visualizzata.

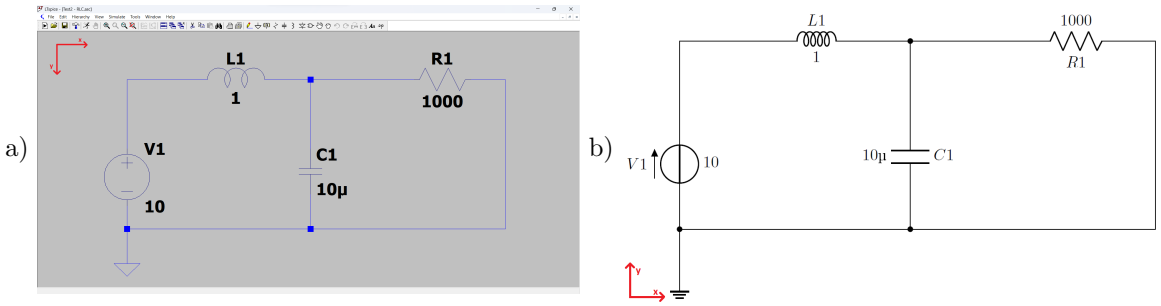

Figura 4: Sistemi di riferimento delle coordinate x e y in LTSpice e in Latex.

Viene quindi creata la cartella `latex_output` e il file `translated_circuit.tex` che conterrà la traduzione in Latex del circuito. Dapprima, viene scritto nel file latex il preambolo che contiene la descrizione del tipo di documento, la dimensione del foglio e l'import del pacchetto `CircuiTikz`. Queste informazioni verranno utilizzate dal compilatore `MikTex` durante la generazione del file pdf.
Il passo successivo consiste nella vera e propria traduzione del circuito in Latex effettuata dalla funzione `translateCircuitToLatex`. Inizialmente, vengono calcolate altezza e larghezza del circuito e nel caso in cui lo schematico sia più largo che alto viene impostata la variabile `rotate` che ruota il circuito in modo da occupare meglio lo spazio. Poi, viene calcolato il fattore di scala (memorizzato nella variabile `LATEXSCALE`) tra le coordinate espresse nel sistema di riferimento di LTSpice e quello di Latex, in modo tale che il circuito tradotto sia contenuto interamente in un unico foglio. Successivamente, vengono tradotti tutti i componenti, i collegamenti e le label inserite nelle liste `components`, `wires` e `flags`; questa operazione richiede di recuperare le coordinate x e y degli estremi di collegamento di ogni componente ed effettuare una rototraslazione per riportare le coordinate dal sistema di riferimento di LTSpice nel sistema di riferimento di Latex: alla coordinata x viene sottratta la x minima calcolata nella funzione `calculateAspectRatio` mentre alla y viene sottratta la coordinata y massima, anch'essa calcolata precedentemente. Le coordinate y vengono poi invertite. Infine, tutte le coordinate vengono

scalate con il fattore di scala `LATEXSCALE` calcolato precedentemente. Il comando Latex utilizzato per disegnare un componente secondo la libreria `CircuiTikz` è `draw` ed è così definito:

- \draw (x1,y1) to[⟨type⟩=⟨name⟩, a=⟨value⟩] (x2,y2), per disegnare un componente; il ⟨type⟩ può essere R, C, eC, L, Do o vsource per indicare una resistenza, un capacitore, un capacitore polarizzato, una induttanza, un diodo e un generatore di tensione; ⟨name⟩ e ⟨value⟩ corrispondono al nome e al valore del componente; le coordinate degli estremi vengono calcolate a partire dalle coordinate x,y del componente indicate nel file .asc che fanno riferimento al centro del componente; a queste coordinate vengono applicati degli offset (ricavati sperimentalmente) per calcolare la posizione dei capi di collegamento in funzione anche della rotazione o operazione di mirror applicata; queste operazioni sono inserite nella funzione `applyRotationDipole` nella classe `Component`;

- \draw (x1,y1) to[short, l=⟨label⟩] (x2,y2), traccia un filo di collegamento dal punto (x1,y1) al punto (x2,y2) e associa eventualmente una label ⟨label⟩ al collegamento; questa label viene ricercata nella lista `flags`, cercando una flag che abbia le stesse coordinate di uno dei capi del collegamento;

- \draw (x,y) to (x,y) node[ground], inserisce un nodo di massa nel punto (x,y); i nodi di massa in LTSpice sono rappresentati da dei flag con valore 0;

- \draw (x,y) to[short, -*] (x,y), permette di rimarcare un nodo come punto di collegamento tra tre fili; essi vengono ricavati cercando l'intersezione di almeno tre collegamenti grazie alla funzione `findNode`;

Dopo aver terminato la traduzione, la funzione `closeFileLatex` esegue le operazioni di chiusura del file latex, dopo aver inserito l'epilogo latex. Successivamente, crea un nuovo processo `pdflatex` al quale passa il nome del file (`translated_circuit.tex`) da cui generare il pdf. Se l'operazione va a buon fine, viene prodotto nella cartella `latex_output` il pdf con la rappresentazione del circuito.
Gli eventuali errori lessicali, sintattici e semantici vengono raccolti nella lista `errorList` dell'handler. Il metodo `main`, contenuto nella classe `ParserTester`, chiama il metodo `parseCircuit` dichiarato nella classe `Ltspice2circuitikzParser` che avvia l'analisi sintattica e le operazioni di traduzione. Al termine, viene verificato se la lista `errorList` è vuota. In tal caso l'applicazione termina. Altrimenti, vengono eliminate le cartelle `circuit_output` e `latex_output` e viene creata la cartella `logs` che conterrà un file sul quale vengono scritti gli errori rilevati.

3 Gestione degli errori

L'applicazione gestisce errori lessicali, sintattici e semantici. Di seguito sono riportati i codici degli errori semantici gestiti dall'applicazione con il loro significato:

- `VERSION_ERROR`: l'applicazione non è in grado di gestire il numero di versione indicato. Al momento, l'applicazione è in grado di gestire solo la versione 4;

- `WINDOWOPTION_ERROR`: l'opzione riguardante la regola `WINDOW` non è corretta. Sono ammesse le seguenti opzioni: `Invisibile`, `Center`, `Left`, `Right`, `Top`, `Bottom`, `VCenter`, `VLeft`, `VRight`, `VTop` e `VBottom`;

- `IOPINATTR_ERROR`: l'attributo associato alla regola `IOPIN` non è corretto. Sono ammessi i seguenti attributi: `In`, `Out`, `BiDir`;

- `SYMBOLTYPE_ERROR`: il simbolo inserito non è stato riconosciuto. I simboli che al momento vengono riconosciuti dall'applicazione sono `res`, `res2`, `cap`, `ind`, `ind2`, `diode`, `schottky`, `zener`, `LED`, `varactor`, `voltage` e `polcap`;

- `SYMBOLTYPE_TO_IMPLEMENT`: il simbolo inserito è riconosciuto ma deve ancora essere implementato per la traduzione corretta. I simboli presenti nella lista `listSymbolToImplement` sono solo alcuni dei simboli riconosciuti da LTSpice che nella versione attuale della applicazioni non vengono ancora tradotti;
- `ROTATIONTYPE_ERROR`: la rotazione definita per il componente non è valida. Le rotazioni accettate sono `R0`, `R90`, `R180` e `R270`;
- `MIRRORTYPE_ERROR`: l'attributo di mirror definito per il componente non è valido. Gli attributi possibili sono `M0`, `M90`, `M180` e `M270`;
- `ROTMIRR_ERROR`: l'attributo inserito non indica ne rotazione ne mirror;
- `SYMATTRTYPE_ERROR`: la parola che segue il token `SYMATTR` non è corretta. Le parole ammesse sono `InstName`, `Description`, `Type`, `Value`, `SpiceLine`;
- `DESCRIPTION_ERROR`: la parola che segue l'attributo `Description` deve essere coerente con il simbolo in questione. Se il simbolo è `schottky`, `zener` o `LED` allora la parola che segue `Description` deve essere `Diode`. Se il simbolo è `polcap`, deve invece essere `Capacitor`;
- `DESCRIPTION_POLARIZED_ERROR`: è un caso particolare dell'errore precedente. Nel caso in cui il simbolo è `cap`, se la descrizione è presente deve essere uguale a `Polarized Capacitor`;
- `DESCRIPTION_POLARIZED_NOCAPACITOR_ERROR`: indica il caso in cui il simbolo è `cap` e ci sia solo la parola `Polarized` come `Description`, mentre l'applicazione si aspetta `Polarized Capacitor`;
- `TYPE_ERROR`: i valori dell'attributo che segue `Type` dipende dal componente in questione. Nel caso in cui il componente è `shottky`, `zener`, `LED` allora bisogna indicare `diode`. Nel caso in cui, invece, è `polcap` il valore accettato è `cap`;
- `SPICELINEVALUE_ERROR`: un attributo che segue la parola `SpiceLine` non è corretto. Gli attributi riconosciuti sono `tol`, `pwr`, `Rser`, `Rpar`, `Cpar`, `Ipk`, `mfg`, `pn`, `V`, `Irms`, `Lser`, `type`;
- `SYMBOLTYPENULL_ERROR`: indica che son presenti degli attributi ma non è stato dichiarato un simbolo in precedenza;

4 Client app

L'applicazione client è stata sviluppata utilizzando le librerie Qt 6.4.3 e l'ambiente di sviluppo Qt Creator 10.0.1. L'applicazione è stata sviluppata per sistema operativo Windows, ma grazie alle librerie Qt con lo stesso codice è possibile generare eseguibili anche per altri sistemi operativi (Linux, MacOs, Android e IOs). In figura 5 è mostrato il diagramma delle classi dell'applicazione. Il suo funzionamento viene descritto nel Manuale Utente.

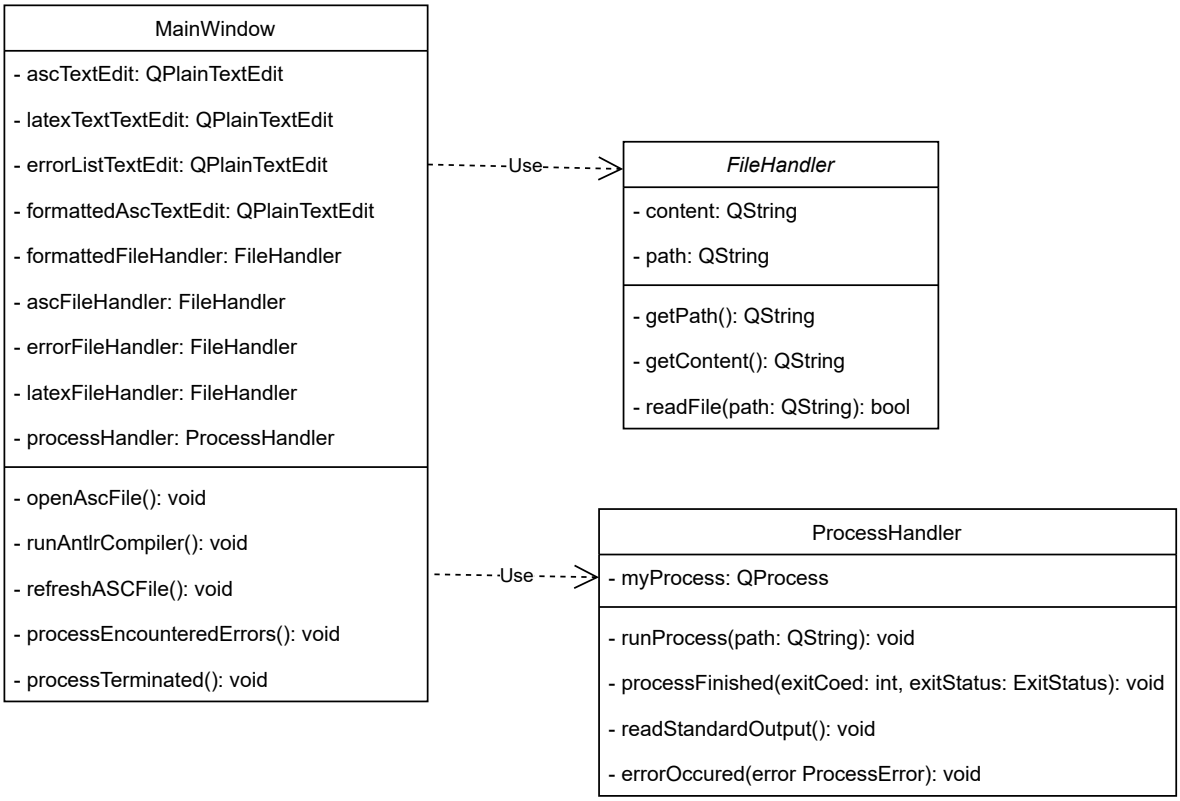

Figura 5: Class Diagram dell'app client.

5 Esempi e sviluppi futuri

Di seguito viene riportata una lista contente gli sviluppi futuri:

- **introduzione di nuovi componenti**: al momento, i componenti che possono essere utilizzati sono resistenze, condensatori, condensatori polarizzati, diodi e generatori di tensione. Ci sono altri componenti che possono essere inseriti all'interno dell'applicazione per generare circuiti più complessi.
- **gestione di formati diversi da A4**: al momento, l'applicazione è in grado di disegnare circuiti scalati in modo corretto basandosi sulle dimensioni di un foglio A4.
- **gestione di fattori di scala molto grandi**: nel caso in cui il circuito sia estremamente grande, l'output generato può contenere dei difetti dovuti al fattore di scala. In tal caso, è consigliabile dividere il circuito su più fogli.

Di seguito viene mostrato come utilizzare l'applicazione *ltspice2circuitikz*.

1. Aprire l'applicazione *ltspice2circuitikz* cliccando due volte l'eseguibile *ltspice2circuitikz.exe*. Si aprirà una schermata come quella mostrata in figura 6.

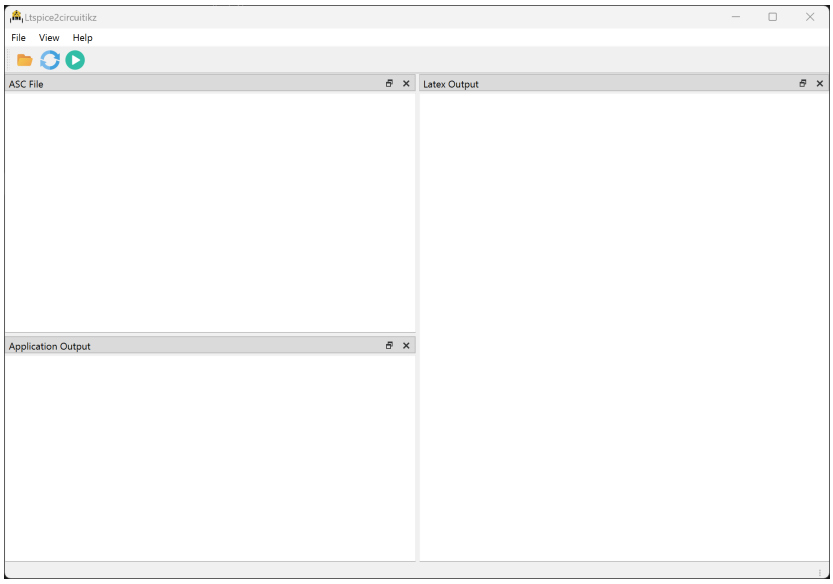

Figura 6: Schermata principale dell'applicazione.

2. Cliccare l'icona *"Open File"* e selezionare un file .asc da aprire. Una volta selezionato il file, il suo contenuto verrà mostrato nella finestra in alto a sinistra (figura 7). Il contenuto del file non può essere modificato dall'applicazione ma deve essere modificato da un tool esterno. Nel caso venga modificato è possibile ricaricare il file premendo l'icona di *"Refresh"*.

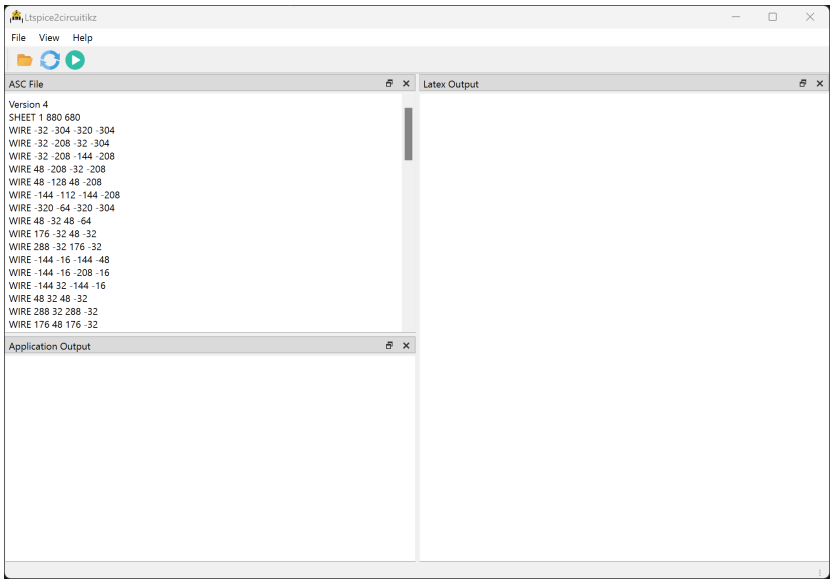

Figura 7: Schermata che mostra il file aperto.

3. Premere il pulsante verde *"Run"* per lanciare il processo di conversione. Se non ci sono errori, viene mostrato un messaggio verde nell'*Application Output* e viene caricato il contenuto del file latex nella colonna di destra (figura 8). Inoltre, viene aperto il file pdf contenente il circuito generato. Se si vuole mostrare anche il file formattato in modo corretto, è sufficiente selezionare la voce *"Formatted ASC File"* nel menù *"View"*.

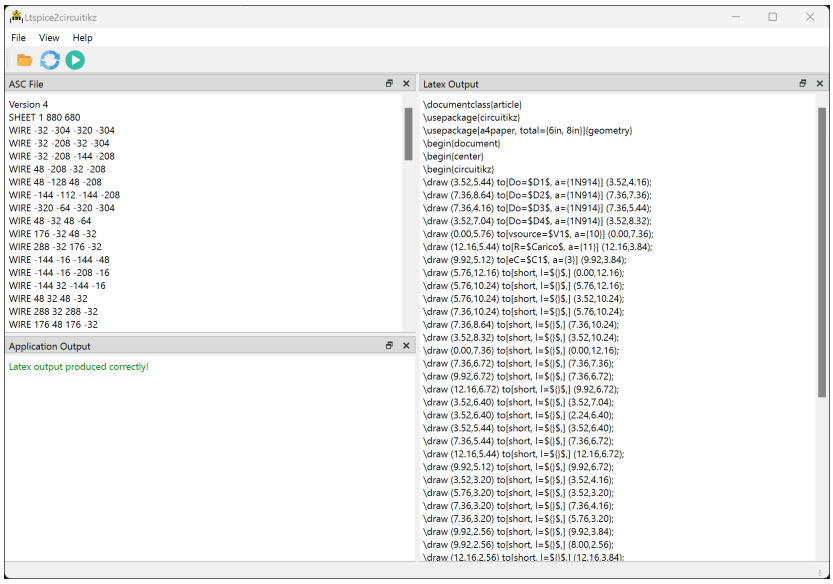

Figura 8: Processo eseguito con successo.

4. Nel caso in cui ci siano errori semantici o sintattici, il processo si interrompe e nell'*Application Output* vengono mostrati gli errori (figura 9).

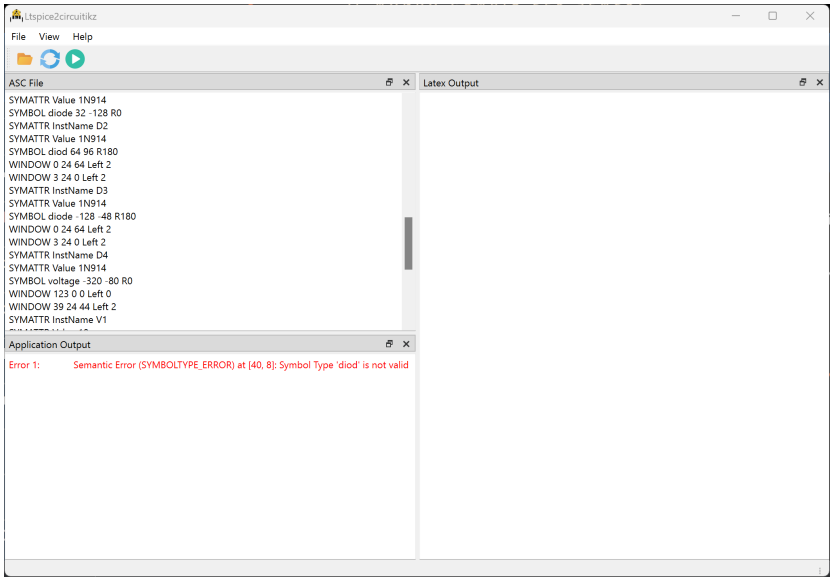

Figura 9: Errori semantici.