Universidad Nacional de Rosario

Licenciatura en Ciencias
de la Computación

Tesina de Grado

Aplanado eficiente de grandes modelos Modelica

Autor:
Mariano Botta

Directores:
Federico Bergero
Ernesto Kofman

Legajo: B-5149/7

25 de agosto de 2015

Resumen

Modelica es un lenguaje de modelado de todo tipo de sistemas. Gracias a su metodología de programación orientada a objetos y a la posibilidad de vectorizar los modelos, mediantes arreglos y ecuaciones *for*, Modelica facilita la escalabilidad de modelos.

Las herramientas existentes de compilación de modelos Modelica desaprovechan estas características de los modelos, expandiendo las ecuaciones *for* y eliminando la vectorización del mismo desde las primeras etapas de compilación. Esto reduce gravemente la performance del compilador.

En esta tesina, desarrollamos un algoritmo de aplanado que mantiene la vectorización de los modelos, conservando los arreglos y las ecuaciones *for*. También fue necesario crear un algoritmo de resolución de componentes conexas en un grafo vectorizado para poder resolver las conexiones existentes en los modelos.

Índice general

Capítulo 1

Introducción

1.1. Motivación

A lo largo de la vida, el humano trata de entender cómo funcionan las cosas, por simple curiosidad o para el desarrollo de algún nuevo adelanto tecnológico. Sea cual sea la finalidad, la técnica empleada, generalmente, fue la de observación y experimentación sobre sistemas reales.

Experimentar sobre sistemas reales tiene muchas desventajas: adecuarse a los tiempos del sistema, grandes costos económicos, posibles daños al planeta, entre otros. Por eso, con la innovación en computación apareció un nuevo concepto que simplifica en gran medida estos obstáculos; la simulación.

Gracias a la simulación, cientos de científicos y profesionales de todo el mundo pueden realizar experimentos en modelos computacionales que luego se aplican a la realidad. Así, abaratan costos, no corren peligro, no dañan el ecosistema, tienen el control de los parámetros del sistema, los tiempos del mundo real se escalan a tiempos adecuados, entre otras ventajas.

Todas estas ventajas introduce la simulación en la rama de la investigación, en consecuencia, urge enormemente el desarrollo constante de nuevas herramientas de simulación y modelado para atacar las nuevas inquietudes de la mente humana.

Día a día los modelos que se busca simular van creciendo. Se agregan detalles, se expanden las fronteras, etc. Todos estos cambios hacen que los modelos se vuelvan más complejos y grandes. Con la posibilidad de experimentar sobre sistemas simulándolos, el conocimiento acerca de éstos fue creciendo y así también los modelos asociados. Esto conlleva a que el costo computacional, tanto de modelarlos como de simularlos, se eleve.

En este trabajo se desarrollan algoritmos para tratar eficientemente grandes modelos descriptos en el lenguaje Modelica. Estos algoritmos fueron implementados como parte de una herramienta que tiene el fin de simular modelos Modelica.

1.2. Objetivos

Como mencionamos antes, día a día los modelos se van volviendo más grandes. En general, un modelo grande no es descripto por extensión, sino que se describen sus componentes fundamentales. Luego se los replica y se componen para obtener el modelo final; de esta manera, aunque el modelo sea grande su descripción es pequeña.

Modelica es un lenguaje orientado a objetos que permite desarrollar modelos mediante ecuaciones acausales, es decir, el signo *igual* no tiene el significado de una asignación, sino que genera una igualdad entre ambas expresiones. En estos modelos, el orden de las ecuaciones no altera el significado del mismo. Estos modelos requieren ser procesados antes de poder simularlos. Una de las etapas de este procesamiento es lo que se conoce como "aplanado". En esta etapa muchas estructuras del lenguaje son transformadas en estructuras equivalentes pero más simples.

Las herramientas Modelica disponibles hoy en día no son capaces de lidiar, especialmente, con modelos como los que se mencionó con anterioridad ya que en la etapa de aplanado, expanden el modelo obteniendo una extensa descripción. Consecuentemente, las etapas subsiguientes fallan al tratar de procesar un modelo tan grande.

Por lo expuesto, el objetivo de esta tesina consiste en desarrollar un algoritmo de aplanado de modelos Modelica claro y documentado, poniendo especial atención en que preserve la simplicidad en la descripción del modelo.

Proyecto de Investigación del Grupo

El presente trabajo se enmarca en el proyecto PID-ING386, Modelado, Simulación y Control en Tiempo Real con Aplicaciones en Electrónica de Potencia, UNR y PICT 2012-0077 de la Agencia Nacional de Promoción Científica y Tecnología. Uno de los objetivos del proyecto de investigación es la simulación en paralelo, utilizando los métodos de cuantificación de estado Quantized State Systems [4].

En general, los casos de prueba utilizados en la simulación en paralelo son modelos grandes, ya que en modelos pequeños no habría mucha ganancia. Por ello un tratado eficiente de modelos grandes es fundamental para el proyecto de investigación.

1.3. Organización de la Tesina

En el capítulo 2 se introducen los conceptos necesarios para la compresión del trabajo desarrollado. En los capítulos 3 y 4 queda documentado el desarrollo del algoritmo de aplanado que está dividido en dos etapas. En el capítulo 5 se mencionan cuestiones técnicas sobre la implementación del

algoritmo y, por último, en el capítulo 6 se realizan comparaciones entre el algoritmo de aplanado aquí expuesto y la herramienta OpenModelica.

Capítulo 2

Conceptos Previos

2.1. Modelica

Modelica [2, 11] es un lenguaje orientado a objetos, para el modelado de sistemas complejos, con componentes mecánicos, eléctricos, electrónicos, hidráulicos, térmicos, etc. El lenguaje no tiene restricciones de uso (licencia Modelica V2) y es desarrollado por la asociación sin fines de lucro "Modelica Asociation". Un modelo Modelica es una representación textual, aunque las herramientas de modelado y simulación de Modelica permiten componer el modelo gráficamente sin tener que escribirlo.

Existen diferentes entornos de desarrollo para trabajar con Modelica. Dentro del rubro código abierto, existe el entorno OpenModelica [1] [11] desarrollado por la organización Open Source Modelica Consortium (OSMC). OpenModelica está siendo desarrollado tanto para fines industriales como para fines académicos. Incluye un entorno gráfico para el desarrollo de modelos y una librería con cientos de componentes, de diferentes ámbitos, ya desarrollados. Así, OpenModelica es apropiado para usuarios que desconocen acerca de programación, o bien, no poseen un amplio conocimiento de la misma.

A continuación se presenta un resumen de las características del lenguaje Modelica. En [11] puede verse una especificación detallada del mismo.

2.1.1. Clases

Las clases, como en cualquier otro lenguaje de programación orientado a objetos, representan la unidad fundamental de la estructura de un modelo en Modelica. Las clases proveen la estructura de los objetos a través de variables y ecuaciones que definen el comportamiento de la clase. A partir de éstas, las herramientas de simulación computan la evolución del modelo.

[1] https://openmodelica.org/

```
class circuits
  class Pin
    Real v;
    flow Real i;
  end Pin;
  Pin n,p;
end circuits;
```

Ejemplo 2.1: Ejemplo de clase e instancias.

En el Ejemplo 2.1 se define la clase *Pin* que tiene dos variables, v e i. Observamos que Modelica permite la anidación de clases ya que, como vemos, *Pin* está definida dentro de *circuits*. Por último, en la clase *circuits* se declaran dos instancias de la clase *Pin*.

Modelica incluye restricciones de clases que enriquecen el código, le agregan un sentido semántico además de ciertas restricciones, pero no dejan de ser clases. Permiten que el código sea más fácil de leer y de mantener. Simplemente se cambia la palabra reservada *class*, a la hora de definirla, por otra dependiendo del objetivo de la misma. Los tipos de estas clases son *model*, *record*, *block*, *connector*, *function*, *package*, etc.

Al modelo del Ejemplo 2.1 lo modificamos, agregando dos restricciones de clase y obtenemos el modelo del Ejemplo 2.2. Convertimos a la clase *Circuits* como un paquete, el cual contendrá todas nuestras futuras definiciones, y declaramos a la clase *Pin* como un conector. Esto le da un significado a la clase y anticipa su uso.

```
package circuits
  connector Pin
    Real v;
    flow Real i;
  end Pin;
  model Componente
    Pin n,p;
  end Componente;
end circuits;
```

Ejemplo 2.2: Ejemplo de clase e instancias.

2.1.2. Modelo Orientado a Objetos

Todo los sistemas pueden descomponerse en pequeños objetos. Algunos repetidos o semejantes, otros únicos. Pero todos interactúan entre sí para cumplir con la funcionalidad del sistema; ningún objeto queda aislado. Los sistemas pueden verse como la composición de objetos indivisibles o desde un nivel jerárquico más amplio, como la composición de subsistemas. Un subsistema está compuesto por otros subsistemas u objetos indivisibles. La

idea principal es que éstos sean tratados como nuevos objetos.

Como ya mencionamos, Modelica está orientado a objetos. Cada clase puede ser vista como un objeto de la realidad; poseen variables y ecuaciones que le dan propiedades y funcionalidades para actuar como el objeto de la realidad. A través de la definición de variable, un objeto puede almacenar y hacer uso de otros objetos. Así, decimos que un objeto está compuesto por otros objetos y semánticamente podemos observar este objeto como un subsistema.

Por otro lado, este paradigma orientado a objetos incluye el concepto de herencia. La herencia es una técnica para enriquecer semánticamente el modelo y/o reutilizar código. Cuando un objeto hereda de otros, adquiere las variables y ecuaciones de estos y así posee las cualidades de los demás. De esta forma, se modela un objeto y luego se puede ir refinando en diferentes versiones, pero siempre manteniendo las cualidades principales.

Dentro de Modelica, una clase puede heredar a otras mediante la sentencia especial *extends*. Al paquete *Circuits* que venimos desarrollando le vamos a agregar dos nuevos modelos. Podemos observar las modificaciones en el modelo del Ejemplo 2.3.

```
1  package Circuits
2
3    ...
4
5    model OnePort
6      Pin p;
7      Pin n;
8      Real v;
9      Real i;
10   equation
11     v = p.v - n.v;
12     i = p.i;
13     i = -n.i;
14   end OnePort;
15   model Capacitor
16     extends OnePort;
17     parameter Real C = 1;
18   equation
19     C * der(v) = i;
20   end Capacitor;
21 end Circuits;
```

Ejemplo 2.3: Clases y herencias.

Primero agregamos un modelo básico, llamado *OnePort*, que representa un componente electrónico que posea dos pines (*Pin* positivo y *Pin* negativo) pero no realiza ninguna modificación a la corriente ni al voltaje.

Por otro lado, el modelo *Capacitor*, hereda de *OnePort*. Así el *Capacitor* obtiene las definiciones de este último, posee dos *Pines* y las ecuaciones

de voltaje y corriente. Para diferenciarse de la clase *OnePort*, agrega una ecuación que establece una relación entre el voltaje y la corriente por el *Capacitor*.

En el Ejemplo 2.4 se observa una definición equivalente del modelo *Capacitor* sin usar herencia.

```
1  package Circuits
2    model Capacitor
3      Pin p;
4      Pin n;
5      Real v;
6      Real i;
7      parameter Real C = 1;
8    equation
9      v = p.v - n.v;
10     i = p.i;
11     i = -n.i;
12     C * der(v) = i;
13   end Capacitor;
14 end Circuits;
```

Ejemplo 2.4: Modelo *Capacitor* sin herencia.

2.1.3. Tipos

Dentro de Modelica existen los tipos básicos: *Real*, *Integer*, *Boolean* y *String*. Además de los tipos básicos, las clases definen nuevos tipos.

A partir de los tipos mencionados, se pueden construir nuevos tipos a través de los sinónimos de tipos. Estos modifican o agregan nuevas características al tipo.

```
1  package Circuits
2    type Current = flow Real;
3    type Voltage = Real;
4    connector Pin
5      Voltage v;
6      Current i;
7    end Pin;
8    type TenPin = Pin[10];
9  end Circuits;
```

Ejemplo 2.5: Sinónimo de tipo.

En el Ejemplo 2.5 vemos como modificamos los tipos de las variable dentro del connector Pin. Con una simple lectura del código queda claro el objetivo del modelo y de sus variables. El conector posee dos variables, una representa la corriente y otra el voltaje del circuito. Por supuesto esta nueva definición es equivalente a la anterior.

Además de crear sinónimos de tipos, a estas definiciones se le pueden agregar otras modificaciones:

- Prefijos de tipo: el tipo *Current* está definido como una flujo (prefijo *flow*).
- Arreglos: el tipo *TenPin* es un sinónimo de un arreglo de diez *Pin*.
- Modificaciones de tipo: las modificaciones de tipo son útiles para generar variaciones de un tipo entre línea y evitar definir un nuevo tipo cada vez que se tenga que usar. Serán explicadas en la sección 2.1.5.

2.1.4. Definición de Variables

A lo largo de los distintos ejemplos vimos varias definiciones de variables. Definiciones básicas de tipo real, definiciones con prefijos y definiciones de instancias de clases. En su totalidad una declaración de variable se compone por:

1. **Prefijos de tipos**: *flow*, *constant*, *parameter*, *discrete*, *input* y *output*.
2. **Tipo**: nombre del tipo de la variable. Puede ser un tipo básico, una clase o un sinónimo de tipo. Ejemplo: *Real*, *String*, *Pin*, *TenPin*.
3. **Nombre de la variable**. Ejemplo: *auto*, *habitación*, *casa*, etc.
4. **Dimensión**: Modelica permite la definición de arreglos.
5. **Modificaciones**: sirven para agregar pequeños o hasta grandes cambios al tipo sin tener que definir toda una nueva clase.

2.1.5. Modificaciones

Las modificaciones pueden ocurrir dentro de 3 contextos: declaración de variable, sinónimo de tipo y definiciones de herencia. La modificación más básica cambia el valor por defecto de una variable (Modificación de Asignación). Notar que el signo *igual* funciona como una asignación, determinando el valor de inicio de la variable. En el Ejemplo 2.6 se define una variable de tipo *Integer* con valor inicial 3 y un *Capacitor* con la modificación de su parámetro de configuración.

```
1 Integer i = 3;
2 Capacitor cap(C = 2);
```

Ejemplo 2.6: Modificación básica.

Modelica también nos permite tener modificaciones anidadas, es decir, agregar nuevas modificaciones a una variable interna. En el Ejemplo 2.7 vemos como se aplica esta modificación. Definimos una única vez al circuito *CircuitOne* pero lo instanciado dos veces con diferente tipo de *Capacitor*, es decir, alteramos de diferente forma el capacitor que se usa. Esta modificación nos permite realizar alteraciones en diferentes niveles de anidamiento de clases.

```
1  package Circuits
2    model CircuitOne
3      Capacitor cap;
4      Resistor res;
5      ...
6    equation
7
8    ...
9
10   end CircuitOne;
11   model MainCircuit
12     CircuitOne co1 (cap(C = 10));
13     CircuitOne co2 (cap(C = 15));
14   end MainCircuit;
15 end Circuits;
```

Ejemplo 2.7: Modificación anidada.

También existen modificaciones más complejas. Se puede redeclarar una variable asignándole un nuevo tipo, nuevos prefijos, cambiando la dimensionalidad y/o agregando nuevas modificaciones. Esta modificación comienza con la palabra clave *redeclare*. Además de las variables, los tipos también pueden ser redeclarado.

```
1  class A
2    type RealInput = input Real;
3    RealInput x;
4  end A;
5  class B
6    Real b;
7  end B;
8
9  parameter Integer N = 10;
10 A a1 (redeclare B x (b = 2));
11 A a2 (redeclare RealInput = B[N]);
```

Ejemplo 2.8: Modificaciones de tipo y variable.

En el Ejemplo 2.8 hay dos modificaciones. La primera modificación cambia el tipo de la variable x al tipo B y además, le agrega una modificación de asignación, cambiando su valor inicial a dos. En la segunda variable, la modificación que presenta cambia la definición del tipo RealInput por un

arreglo de dimensión N de tipo B.

2.1.6. Ecuaciones

A diferencia de otros lenguajes de programación, en Modelica las ecuaciones no representan una asignación, sino igualdades. Las ecuaciones pueden tener expresiones complejas de ambos lados de la igualdad y expresan una relación entre las variables. Además, el orden en el que están definidas no influye en el resultado final. En el ejemplo 2.9 podemos ver las ecuaciones definidas en el modelo *OnePort*.

```
1 v = p.v - n.v;
2 i = p.i;
3 i = -n.i;
```

Ejemplo 2.9: Diferentes ecuaciones.

Como en muchos lenguajes de programación, Modelica posee ecuaciones de iteración (*for*). Una forma, poco práctica, de definir N capacitores podría ser usando ecuaciones *for*. Por ejemplo, en el modelo 2.10, definimos N capacitores desde el modelo *Capacitor* expandido (Ver modelo 2.4).

```
1  package Circuits
2    model Capacitor
3      parameter Integer N = 10;
4      Pin p[N];
5      Pin n[N];
6      Real v[N];
7      Real i[N];
8      parameter Real C[N] = 1;
9    equation
10     for i in 1:N loop
11       v[i] = p[i].v - n[i].v;
12       i[i] = p[i].i;
13       i[i] = -n[i].i;
14       C[i] * der(v[i]) = i;
15     end for;
16   end Capacitor;
17 end Circuits;
```

Ejemplo 2.10: Ecuación *for*.

El ejemplo visto no es como se usa en la realidad, pero es de utilidad para presentar las ecuaciones *for*. Una ecuación *for* no es más que una forma reducida de escribir un número definido de ecuaciones. En el Ejemplo 2.10 vemos un *for* que itera 10 veces, éste es equivalente a tener 10 ecuaciones simples. Un fragmento de esta expansión se observa en el Ejemplo 2.11.

```
1  v[1] = p[1].v - n[1].v;
2  v[2] = p[2].v - n[2].v;
3  v[3] = p[3].v - n[3].v;
4  v[4] = p[4].v - n[4].v;
5  v[5] = p[5].v - n[5].v;
6  v[6] = p[6].v - n[6].v;
7  v[7] = p[7].v - n[7].v;
8  v[8] = p[8].v - n[8].v;
9  v[9] = p[9].v - n[9].v;
10 v[10] = p[10].v - n[10].v;
```

Ejemplo 2.11: Ecuación *for* expandida.

Este es el punto donde se pierde parte de la simplicidad en la descripción de grandes modelos. Si tuviésemos que la variable i itera entre 1 y 1000000, o más, expandir esta ecuación *for* sería altamente perjudicial en el tiempo computacional en las siguientes etapas de compilación.

Otro tipo de ecuaciones que incluye Modelica son las condicionales (*if*). Las ecuaciones se computan en la simulación sólo si la condición es verdadera. En el Ejemplo 2.12 se muestra cómo definir una ecuación de este tipo.

```
1 if i == 0 then
2   a + 1 = p * t + 10;
3 end if;
```

Ejemplo 2.12: Ecuación *if*.

Otro tipo de ecuación que es de gran interés es la ecuación *connect*. La ecuación connect genera una relación de conexión entre dos componentes y es muy utilizada, por ejemplo, en modelos electrónicos para conectar diferentes componentes.

Estas ecuaciones pueden ser establecidas únicamente entre dos conectores del mismo tipo, o entre dos clases con prefijo *class* y que reúnan las restricciones para ser un conector.

Un conector es una clase que sólo contiene variables y se suele representar con el prefijo de clases *connector*. Se utiliza justamente para representar alguna conexión en el modelo. Dos conectores son iguales si las variables que poseen tienen igual nombre y equivalente tipo. (En este trabajo nos restringimos a usar solamente clases del tipo *conector*). El Ejemplo 2.13 muestra el uso de una ecuación *connect*.

```
package Circuits

  ...

  model ground
    Pin p;
  equation
    p.v = 0;
  end ground;
  model inductor
    extends OnePort;
    parameter Real L = 1;
  equation
    L * der(i) = v;
  end inductor;
  model LC_circuit
    Capacitor cap(v(start = 1));
    inductor ind(L = 2);
    ground gr;
  equation
    connect(ind.p,cap.p);
    connect(ind.n,cap.n);
    connect(cap.n,gr.p);
  end LC_circuit;
end Circuits
```

Ejemplo 2.13: Ecuación *Connect*.

Un objeto del tipo conector tiene dos grupos de variables:

1. Variables de potencial. Ejemplo: presión, voltaje, etc.

2. Variables de flujo definidas con el prefijo flow. Ejemplo: corriente, caudal, etc.

Cada ecuación Connect genera distintos tipos de relaciones entre las variables internas del conector, dependiendo si son variables de flujo o de potencial.

- Las variables de potencial dentro de una misma conexión deben ser iguales entre sí.

- Las variables de flujo siguen las reglas de Kirchhoff [17]: la suma de los flujos es igual a cero. Para mantener esta regla hay que considerar como flujo positivo aquel que tenga dirección hacia dentro del componente. En caso contrario, será considerado negativo.

```
// Variables de Potencial
ind.p.v = cap.p.v;
ind.n.v  = cap.n.v;
cap.n.v = gr.p.v;

// Variables de flujo
ind.p.i + cap.p.i = 0;
ind.n.i + cap.n.i + gr.p.i = 0;
```

Ejemplo 2.14: Connects tranformados a ecuaciones de igualdad.

En el Ejemplo 2.14, observamos las ecuaciones que están implícitas en el modelo del Ejemplo 2.13.

Si queremos reemplazar todas las ecuaciones *connect* por sus ecuaciones equivalentes, debemos saber qué conectores están conectados entre sí. Si pensamos que los conectores son nodos de un grafo y que cada ecuación *connect* une éstos con un arco, encontrar cuáles componentes están conectados es equivalente a calcular las componentes conexas del grafo.

2.1.7. Sentencias

Las sentencias permiten la definición imperativa de algoritmos. No hay ecuaciones, solo asignaciones como en otros lenguajes de programación. En ciertas situaciones la definición de algoritmos en forma de ecuaciones se vuelve muy complicado. Por eso agregaron esta sección dentro del lenguaje que comienza con la palabra reservada *algorithm*.

Las sentencias disponibles para usar son las típicas. Asignaciones (:=), condicionales *if* e iteradores *for*, *when*, *while*. A continuación vemos el Ejemplo 2.15 que hace uso de varias de estas sentencias.

```
class A
  Real a[10],b[10],index;
algorithm
  for i in 1:10 loop
    a[i] := b[i]^2;
  end for;

  index := size(a,1);
  while index >= 1 loop
    if x[index]== 8 then
      break;
    else
      index := index + 1;
    end if;
  end while;
end A;
```

Ejemplo 2.15: Statements y sus variantes.

Cabe aclarar que aquí el símbolo *igual* representa una asignación (a diferencia de la sección de ecuaciones). Por lo tanto, una sentencia no puede tener la forma $a + b := 3;$, es decir, el lado izquierdo de una asignación debe ser una variable.

2.2. Simulación de Modelos Modelica

Un modelo Modelica representa matemáticamente un sistema de ecuaciones dinámicas acausales. Estos modelos pueden ser formulados como Ecuaciones Diferenciales Algebraicas [7] (o DAE por su sigla en inglés). Aunque existe métodos para simular DAEs, generalmente éstas son convertidas en Ecuaciones Diferenciales Ordinarias (ODE por su sigla en inglés) ya que estas últimas son más fáciles de simular.

Para llevar un modelo Modelica a una representación ODE debemos realizar una secuencia de transformaciones:

Aplanado del modelo. Esta etapa convierte al modelo en uno más simple, reemplazando herencia, clases y otras estructuras complejas. Esta etapa es la que tratamos en este trabajo.

Reducción de índices. Ciertos sistemas representados como DAE contienen singularidades estructurales (también llamados de alto orden) que deben ser tratadas en esta etapa.

Ordenamiento y optimización de ecuaciones. En esta etapa las ecuaciones del modelo son ordenadas vertical y horizontalmente. Por horizontal entendemos que las ecuaciones tiene la forma $v_i = f(v_0, v_1, ..., v_{i-1})$, es decir que la variable v_i está despejada de la ecuación y por vertical entendemos que las ecuaciones que definen $\{v_0, v_1, ..., v_{i-1}\}$ aparecen antes de la definición de v_i. Luego de esta transformación el modelo ya se encuentra en una formulación ODE.

Una vez que tenemos el modelo en forma de ODE podemos utilizar diversos métodos de integración numérica para simularlo. Los métodos de integración clásicos hacen uso de **time slicing** para calcular las ecuaciones. Es decir, evalúan el lado derecho de la ODE en pasos discretos de tiempo t_k para computar el elemento x_{k+1}. Estos algoritmos están basados en la discretización temporal. Ver [7] para un estudio más detallado del tema.

Otra aproximación es utilizada por la familia de métodos de QSS (por Quantized State System) [7] los cuales no utilizan discretización del tiempo, en cambio, usan cuantificación de estados. Aproximan la ODE según la ecuación $\dot{x} = f(q(t), t)$ donde $q(t)$ es conocido como el vector de estados cuantificados. Existen diversos métodos de QSS, cada uno de ellos utiliza una función distinta para calcular $q(t)$. En [7,16] se definen formalmente los métodos QSS.

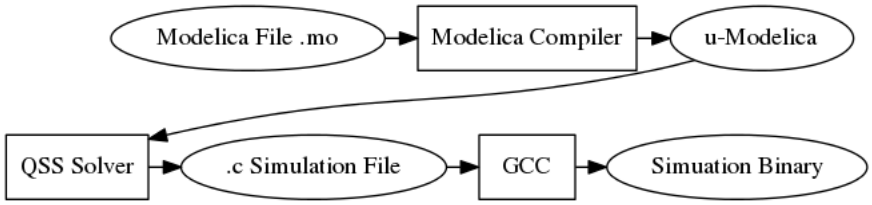

Figura 2.1: Proceso de Compilación.

Estos métodos han demostrado tener ventajas en la simulación de modelos grandes [5], por lo cual el grupo de investigación está desarrollando un conjunto de herramientas para poder simular grandes modelos Modelica.

El trabajo desarrollado en esta Tesina permitirá aplanar modelos grandes sin el costo computacional asociado a la complejidad del mismo y aliviará el trabajo de las etapas siguientes.

2.2.1. Herramientas para Compilación y Simulación

Para que cualquier persona, que no posee conocimiento de este tema, pueda simular un modelo descripto en el lenguaje Modelica aprovechando los métodos de integración QSS; el grupo de investigación está desarrollando una suite de herramientas para tal fin.

Observamos en la Figura 2.1 las diferentes etapas necesarias para la simulación.

1. **Modelica File**: modelo descripto en el lenguaje Modelica (Archivo .mo).
2. **Modelica Compiler**: está compuesto de dos etapas:
   a) Proceso de Aplanado del modelo (herramienta desarrollada en esta tesina).
   b) Transformación a lenguaje μ-Modelica [4] del modelo.
3. **QSS Solver**: [9] generación de archivo de simulación en lenguaje C.
4. **GCC**: genera archivo binario de simulación.

De esta manera, nuestro trabajo permite a usuarios del lenguaje Modelica explotar los beneficios del QSS Solver directamente de la IDE de desarrollo OpenModelica, sin ningún tipo de conocimiento previo sobre QSS.

Dentro de este proceso de compilación, la primer etapa es el Proceso de Aplanado, el cual se va a desarrollar en esta tesina. El QSS Solver no puede trabajar directamente con modelos Modelica, éstos requieren ciertas transformaciones previas que lleven al modelo a un representación más simple del mismo, con mucho menos funcionalidades pero sin perder sus características.

Como vimos antes, en la introducción al lenguaje, Modelica está orientado a objetos. A través de las clases se pueden desarrollar distintas capas de abstracción dentro del modelo. El objetivo del proceso de aplanado es eliminar las distintas capas del modelo, dejando un solo nivel jerárquico ya que los métodos de simulación lo requieren.

2.3. Modelos Grandes

Modelica es un lenguaje muy expresivo, que permite el modelado de diferentes tipos de sistemas y permite mezclarlos para armar modelos muy heterogéneos. Gracias a estas características el campo de aplicación de este lenguaje es enorme. Unos de los grandes desafíos y objetivo de este grupo de investigación, es trabajar con grandes modelos.

El tamaño de un modelo puede ser medido a través de dos parámetros. Por un lado podemos medir "físicamente"qué tan grande es la descripción de un modelo; Cuanto más clases, variables, ecuaciones y sentencias hayan definidas, más grande será el modelo.

La otra forma de medir el tamaño de un modelo es considerar la dimensionalidad del modelo. Cuando empezamos a trabajar con arreglos y ecuaciones *for* el tamaño de un modelo puede variar con tan solo cambiar el valor de un parámetro. Así, el tamaño de la descripción del modelo sigue siendo el mismo, pero la magnitud real del sistema varía considerablemente.

En esta tesina consideramos a la dimensionalidad como el tamaño del modelo y nos enfocamos particularmente en los modelos que están definidos en relación a un parámetro, permitiendo éste, experimentar fácilmente con distintos tamaños del sistema.

2.3.1. Problemas

Un sistema grande con miles de componentes se traducen en miles de variables y ecuaciones en el modelo simplificado. Modelica presenta técnicas de programación para definir en pocas líneas muchas variables a la vez. Esto se hace a través de los arreglos. Además, con las ecuaciones *for* podemos manipular un gran número de componentes iguales en pocas líneas de programación. Así, un modelo que en principio parece que va a ocupar miles de líneas de código, termina ocupando mucho menos de lo que se pensaba.

El problema que se encuentra con los grandes modelos es que hay que mantener las propiedades antes mencionadas durante todo el proceso de compilación. Una ecuación *for*, que itera un millón de veces cuenta como una sola durante la compilación del modelo, pero si se expande en alguna de las etapas de compilación, hay que procesar un millón de ecuaciones diferentes provocando un notable decrecimiento en la eficiencia del compilador.

El Ejemplo 2.16 es una línea de transmisión y es un claro ejemplo de las dificultades que puede tener un modelo. Por un lado, el Ejemplo *LC_circuit*,

presentado en la página 17, fue adaptado para poder ser usado como un componente electrónico al igual que un capacitor o un inductor. El modelo *LC_line* representa una línea de trasmisión, que no es más que una conexión en serie de N circuitos *LC_circuit*.

```
1  package Circuits
2    model LC_circuit
3      capacitor cap(v(start = 1));
4      inductor ind(L = 2);
5      Pin p1,p2,p3;
6    equation
7      connect(ind.p,p3);
8      connect(ind.p,cap.p);
9      connect(cap.n,p1);
10     connect(ind.n,p2);
11   end LC_circuit;
12
13   model LC_line
14     constant Integer N = 10;
15     LC_circuit lc[N];
16     ground gr;
17   equation
18     connect(lc[N].p1,lc[N].p2)
19     for i in 1:N - 1 loop
20       connect(lc[i + 1].p3,lc[i].p2);
21     end for;
22     for i in 1:N loop
23       connect(gr.p,lc[i].p1);
24     end for;
25   end LC_line;
26 end Circuits;
```

Ejemplo 2.16: Modelo de una línea de trasmisión.

Dentro del algoritmo de aplanado, el primer objetivo es mantener la vectorización en el modelo, es decir, tratar eficientemente las ecuaciones *for*, y por consiguiente, los definiciones de arreglo, de tal manera que estas perduren en el proceso de aplanado.

El segundo objetivo es reducir las ecuaciones *connect*, estas no pueden ser expresadas directamente como ODEs, sino que tienen que ser reducidas al par de ecuaciones que vimos en la sección 2.1.6. Las ecuaciones *connect* no pueden ser resueltas localmente, se necesita tener presente todas las conexiones para poder reducirlas.

Vistas estas dos problemáticas, aclaramos que hemos dividido en dos etapas el proceso de aplanado:

- Etapa 1: transformación a un modelo monolítico, eliminando clases e instancias en el modelo. Tratado especial de ecuaciones *for*, para evitar su expansión (Capítulo 3).

- Etapa 2: reducción de ecuaciones *connect* mediante la resolución de componentes conexas (Capítulo 4).

2.4. Trabajo Relacionado

Actualmente existen varias herramientas para desarrollo y simulación de modelos descriptos en Modelica. Algunas de estas son privativas y otras son libres.

OpenModelica es una alternativa libre, presenta un entorno de trabajo con la posibilidad de crear los modelos gráficamente y una completísima librería, en la cual la mayoría de los objetos básicos están creados y acompañados de ejemplos. OpenModelica posee desarrollado un algoritmo de aplanado pero no está documentado y no genera código Modelica válido (genera código intermedio que es útil para su propio uso interno).

Además hay que mencionar que este aplanado expande todas las definiciones de arreglos y las ecuaciones *for*. Si probamos de aplanar un modelo, como el que podemos observar en el Ejemplo 2.17, el compilador después de unos cuantos minutos nos devuelve un simpático error: Stack overflow. Sería necesario tener una supercomputadora para poder trabajar con estos modelos. A pesar de esto, el proceso de compilación de modelos de OpenModelica funciona correctamente para modelos que no sean muy grandes.

```
1 class A
2   Real a[10000000];
3 equation
4   for i in 1:10000000
5     a[i] = i * 2;
6   end for;
7 end A;
```

Ejemplo 2.17: Modelo grande en pocas líneas de código.

Además de OpenModelica, existen aplicaciones pagas, como Dymola [6] y MathModelica [10], que pueden funcionar un poco más eficientes pero ninguno implementa el algoritmo de aplanado que planteamos en esta tesina.

Se han realizado trabajos en relación a las otras etapas del proceso de compilación en relación a tratar con modelos grandes. En [1] se intenta obtener las estructuras repetitivas dentro de un modelo ya expandido. Aunque esto es un paso en el sentido correcto, lo que se propone en esta tesina es utilizar una representación vectorial del modelo (sin nunca llegar a expandirlo).

Por otro lado, también existe un aplanador de clases en C++. En [3] desarrollaron este tipo de herramienta, que consigue mejorar la performance de los programas C++. Dado que C++ está orientado a objetos, tiene composición y herencia, el aplanado de código C++ es muy similar al apla-

nado de modelos Modelica. Abstrayéndonos del lenguaje de programación, el concepto básico de aplanado de clases es el mismo para todos.

Capítulo 3

Algoritmo de Aplanado - Primera Etapa

Aquí presentamos la primera etapa del algoritmo de aplanado, consistente en obtener una estructura plana preservando las definiciones vectoriales y las ecuaciones *for*. Los ecuaciones de tipo *Connect* son preservadas ya que serán removidas en la segunda etapa del aplanado.

3.1. Problema a resolver

Los subsiguientes pasos del proceso de compilación de un modelo, para su simulación, necesitan que le brindemos código libre de clases, objetos e instancias y otras funcionalidades del lenguaje. Es decir, debemos reemplazar la orientación a objeto que presenta Modelica y presentar todo el modelo dentro de una única clase. Pero eso sí, no debemos perder información durante esta etapa.

```
1  package Circuits
2    connector Pin
3      Real v;
4      flow Real i;
5    end Pin;
6    model OnePort
7      pin p,n;
8      Real v;
9      Real i;
10   equation
11     v = p.v - n.v;
12     i = p.i;
13     i = -n.i;
14   end OnePort;
15 end Circuits;
```

Ejemplo 3.1: Clase no aplanada.

En esta primer etapa del algoritmo, tenemos que eliminar toda referencia a instancias de objetos excluyendo aquellas que sean de tipo connector. Estas requieren de un tratado especial cuando el modelo está aplanado. Al final del aplanado todas las ecuaciones connect se acumulan en la única clase, y es ahí donde podemos proceder a reemplazarlas por sus ecuaciones de igualdad asociadas (desarrollado en el capítulo 4).

En el Ejemplo 3.1 podemos observar que la clase *OnePort* tiene dos instancias de la clase *Pin*. Después del proceso de aplanado, el modelo resultante se puede apreciar en el Ejemplo 3.2. Notamos que no hay más definiciones de clase (salvo la principal), por cada instancia de la clase *Pin*, se agregaron dos variables que representan a la variable v e i que poseía el conector *Pin*.

```
1  package Circuits
2    model OnePort
3      Real p_v;
4      flow Real p_i;
5      Real n_v;
6      flow Real n_i;
7      Real v;
8      Real i;
9    equation
10     v = p_v - n_v;
11     i = p_i;
12     i = -n_i;
13   end OnePort;
14 end Circuits;
```

Ejemplo 3.2: Clase aplanada.

3.2. Algoritmo

El pseudocódigo del algoritmo de aplanado esta representado en el Algoritmo 3.3. Como podemos observar en la línea 9 el proceso es recursivo,es decir, para aplanar la clase principal primero debemos aplanar las instancias de clase que ésta posea y así sucesivamente, hasta llegar a las clases que no tengan uso de otras clases.

Dada una clase *C*, lo primero que debemos hacer es expandir la clase (Expand(C)). Entendemos con expandir a eliminar las herencias definidas en la misma, agregando declaraciones de variables, ecuaciones y demás que la clase heredada pueda tener a la clase que se quiere expandir. Ver sección 3.2.1.

```
1  Flat(C):
2    Expand(C);
3    foreach v in Variables(C):
4      t = ResolveType(v);
5      if isBasic(t) then
6        ChangeType(v,t);
7      else if isClass(t) then
8        ApplyModification(C,t,Modification(v));
9        Flat(t);
10       RemoveComposition(C,t);
11       if isConnector(t) then
12         ChangeType(v,t);
13       else
14         Remove(t);
15       end if;
16     end if;
17   end foreach;
18   foreach e in Equations(C):
19     ChangeVarName(e);
20   foreach e in Statements(C):
21     ChangeVarName(e);
```

Algoritmo 3.3: Algoritmo de aplanado.

A continuación, por cada variable v definida en C, hacemos:

1. Resolver el tipo final de la variable (ResolveType), determinar el tipo de la variable. Al terminar esta función no solo recibimos el nombre del tipo sino también modificadores del mismo (Typeprefix y dimensiones). Ver sección 3.2.3.

2. Si el tipo encontrado t es un tipo básico(isBasic) simplemente cambiamos la definición de la variable v, agregando los prefijos de tipo, las definiciones de arreglo y cambiamos su tipo al encontrado (ChangeType(v,t)).

3. Si t es una clase que no tiene el prefijo connector, aplicamos las modificaciones de clase (ApplyModification. Sección 3.2.2), aplanamos la clase (Flat), y por último aplicamos RemoveComposition en t hacia C. Esta función levanta en un nivel de jerarquía las variables, ecuaciones y sentencias aplicando un renombramiento de variables según corresponda para eliminar ambigüedades, como en el Ejemplo 3.2. Ver sección 3.2.4.

4. Si t es una clase, y tiene el prefijo *connector*, realizamos los mismos cambios como si fuese una clase común, pero vamos a agregar la variable y la clase conectora en C para mantener referencia del tipo de conector a la hora de reducir las ecuaciones connect.

El tercer y último paso (ChangeVarName) consiste en recorrer cada ecuación y sentencia y cambiar cada acceso a una variable dentro de un objeto, el operador "." ($a.b$), por una única variable representante de esa operación (a_b).

Cuando aplicamos RemoveComposition, a cada variable interna se le agrega un prefijo de la forma "NombreDeInstancia_", así que en este paso el operador "." se reemplaza por un guión bajo para que concuerde con las variables que agregamos anteriormente. Al realizar este cambio de nombre, el objeto a puede presentar índices de arreglos, en cuyo caso se mantienen pero se trasladan al final de la variable.

A continuación vemos una ecuación con algunas alternativas y sus trasformaciones. Las Ecuaciones 3.4 y 3.5 muestran los cambios que se aplican.

```
1  a.b + c.d.e = q[1].r[2].t
```

Ecuación 3.4: Ecuación antes de aplanar.

```
1  a_b + c_d_e = q_r_t[1][2]
```

Ecuación 3.5: Ecuación después de aplicar las transformaciones.

También puede aparecer en este proceso una referencia a una variable constante dentro de una clase que esté fuera de este contexto (Ver ejemplo 3.6). En este caso, se reemplaza la referencia a la variable por el valor de la misma. En el ejemplo 3.6 se observa el caso mencionado. Luego de aplicar el proceso de aplanado, donde se produce el cambio de la variable constante por su valor, el resultado es el modelo 3.7.

Mientras que recorremos todas las ecuaciones, cuando nos encontramos con una ecuación *connect* vamos a revisar las dos variables utilizadas. Aquel conector cuyo flujo sea hacia el exterior va a ser marcado con el operador unario de negación, para dejar referencia a la hora de reducir las conexiones.

```
1  class Root
2    class Valores
3      constant Real pi = 3.14;
4    end Valores;
5    class A
6      Real angulo;
7    equation
8      angulo = A.pi;
9    end A;
10   A objeto;
11 end Root;
```

Ejemplo 3.6: Constante dentro de una ecuación.

```
class Root
  Real objeto_angulo;
equation
  objeto_angulo = 3.14;
end Root;
```

Ejemplo 3.7: Constante reemplazada por su valor.

3.2.1. Herencia

En esta etapa del algoritmo se eliminan todas las referencias a herencia que tenga la clase. En pocas palabras, todas las definiciones son copiadas a la clase hijo, variables, definiciones de tipos, ecuaciones y sentencias.

Hay que tener en cuenta que la clase padre puede tener modificaciones, las cuales deben ser aplicadas con anterioridad. Ver sección 3.2.2 para el tratado de modificaciones.

El algoritmo general está representado en el Algoritmo 3.8 y a continuación ejemplificamos el algoritmo para mostrar cómo lidiamos con algunos detalles.

```
Expand(C):
  foreach e in Extends(C):
    t = ResolveType(e);
    ApplyModification(t,Modification(e));
    ExpandClass(c,t);
  end foreach;

ExpandClass(c,t):
  foreach e in Equations(t):
    add(e,C);
  foreach e in Statements(t):
    add(e,C);
  foreach e in Types(t)
    add(e,C);
  foreach v in Variables(t):
    t = ResolveType(v);
    ChangeType(v,t);
    if (isClass(t)):
      add(t,C);
    end if;
    add(v,C);
  end foreach;
```

Algoritmo 3.8: Función Expand.

Como se puede ver en el Ejemplo 3.9, la clase *Capacitor* hereda de *OnePort*. Para expandir la clase *Capacitor* completamente debemos expandir cada herencia por separado (una clase puede heredar múltiples clases). *Capacitor* solo hereda de una clase *OnePort*, y su definición está explícita, sin

embargo el primer paso es resolver el tipo de la clase que se hereda ya que no es tan simple encontrar la definición de la clase en todos los casos (Ver sección 3.2.3).

Para poder copiar las variables y ecuaciones de la clase *OnePort* a *Capacitor*, antes debemos asegurarnos que ésta esté expandida. Así, aplicamos Expand sobre *OnePort*. Expandida la clase procedemos a aplicar, si se observan, las modificaciones definidas sobre *OnePort*. En este caso no hay. (Ver sección 3.2.2).

```
model Root

  ...

  model Capacitor
    extends OnePort;
    parameter Real C = 1;
  equation
    C * der(v) = i;
  end Capacitor;
end Root;
```

Ejemplo 3.9: Clase con herencias

Con la definición correcta de *OnePort*, cada definición de tipo, sentencia y ecuación se copian de la clase padre a la clase hijo (de *OnePort* a *Capacitor*).

En cambio, para las variables hay un tratado especial. Al copiar directamente una variable ésta puede quedar fuera de contexto, es por eso que debemos buscar la definición explícita del tipo, modificar la variable según el tipo encontrado (cambio de tipo, prefijos y dimensionalidades) y agregarla definitivamente. También se agrega la definición de tipo en caso que fuese una clase. El resultado final de expandir la clase *Capacitor* se observa en el Ejemplo 3.10.

```
model Root
  model Capacitor
    Pin p,n;
    Real v;
    Real i;
    parameter Real C = 1;
  equation
    v = p.v - n.v;
    i = p.i;
    i = -n.i;
    C * der(v) = i;
  end Capacitor;
end Root;
```

Ejemplo 3.10: Clase *Capacitor* sin herencias.

En el Ejemplo 3.11 vemos un modelo abstracto para representar la situación en que el tipo de una variable quede fuera de contexto. Podemos observar que la clase Q hereda de D. Si simplemente agregamos las variables que D posee, nos queda una definición sin referencia. Es el caso de la variable d, cuyo tipo es la clase C, pero en el contexto de la clase Q no existe. Por eso hay que buscar y agregar el tipo de cada variable.

```
1  model Root
2    model B
3      model C
4        Real c;
5      end C;
6      model D
7        C d;
8      end D;
9    end B;
10   model Q
11     extends B.D;
12   end Q;
13   Q a;
14 end Root;
```

Ejemplo 3.11: Variables fuera de contexto.

Hay que mencionar que dentro de C podría haber otros nombres de tipos que queden fuera de este nuevo contexto. Para evitar esto, esta nueva clase que agregamos va a mantener a la clase B como su padre. De esta forma, cuando aplanemos esta nueva clase C, cualquier referencia fuera de ella procederá a buscar dentro del contexto B y no en Q. El Ejemplo 3.12 muestra como queda expandida la clase Q.

```
1 model Root
2   model Q
3     model C
4       Real c;
5     end C;
6     C d;
7   end Q;
8   Q a;
9 end Root;
```

Ejemplo 3.12: Clase Q sin herencias.

3.2.2. Modificaciones

Para resolver las modificaciones es necesario tener referencia de la clase objetivo a la que se le aplican las modificaciones y del contexto donde se desarrolla, o sea; la clase donde se generan las modificaciones. Esto es así porque al redeclarar una variable o un tipo, se puede hacer referencia a

un tipo que existe en el contexto donde ocurre la modificación, pero en el contexto de la clase implicada no.

```
model Root
  connector Pin2
    Real b1,b2,c2;
  end Pin2;
  class B
    Pin2 p;
  end B;
  model A
    connector Pin
      Real b1,b2;
    end Pin;
    B b(redeclare Pin p);
  end A;
  A a;
end Root;
```

Ejemplo 3.13: Modificaciones y su contexto.

Vimos que las modificaciones pueden ser de Clase, de Asignación o de Equidad. Cuando aplicamos una modificación sobre una clase, la única que puede ocurrir es la modificación de clase. Existen varias situaciones donde podemos encontrar una de estas: en las herencias, declaración de variables y en los sinónimos de tipos.

Las modificaciones de Clase no son más que una lista de submodificaciones conocidas como Argumentos. Cada uno de estos argumentos pueden cambiar el estado de una variable, cambiar de tipo a una variable o redefinir un tipo existente. Dependiendo de cada una de estas situaciones procederemos de la siguiente manera:

1. Si el argumento es una modificación de asignación, buscamos en la clase objetivo la referencia a la variable y agregamos o cambiamos el valor por defecto de la variable, es decir, cambiamos la asignación.

2. Si el argumento agrega modificaciones a una variable, buscamos en la clase objetivo la variable y agregamos estas modificaciones.

3. Si el argumento redeclara un tipo tenemos dos opciones:

   a) Si redefine un enumerador lo sustituimos por el viejo.

   b) Si redefine un tipo existente, debemos buscar la definición final del tipo en el contexto. Si es del tipo clase, la agregamos como un nuevo tipo en la clase. Luego agregamos un alias de tipo reemplazando el viejo nombre para redirigirlo a la nueva clase. En este alias agregamos los prefijos de tipos y definiciones de arreglo que tenga definido el nuevo tipo.

4. La redeclaración de variables es muy parecida a la redeclaración de tipos: buscamos el nuevo tipo de la variable, y si es de tipo clase, agregamos la definición de la nueva clase; buscamos la variable a modificar y cambiamos su tipo al nuevo nombre de la clase. También tenemos que agregar los prefijos de tipos y, si existen, los índices de dimensionalidad.

3.2.3. Resolución de tipos

Muchas veces el tipo de una variable o la definición de una clase no está explícita, están ocultas en sinónimos de tipos que agregan diferentes propiedades extras al tipo base y a la vez, pueden no estar definidas dentro del nivel jerárquico donde nos encontramos. En este módulo pretendemos dar un algoritmo que resuelva los nombres de tipos y nos dé como resultado la definición exacta.

Dado un contexto C, si queremos resolver el tipo $A.B.D$ nos enfocamos primero en A. Con la definición exacta de A, como nuevo contexto resolvemos el tipo $B.D$, así sucesivamente hasta llegar al último eslabón de la cadena (en este caso D).

Para expandir un nombre de tipo por separado, supongamos el tipo A, buscamos dentro del contexto C una definición de A. Si no existe procedemos a buscar en el padre de C, si no existe en el padre, nuevamente, subimos otro nivel hacia el abuelo de C y así vamos subiendo por el árbol de clases hasta encontrar una definición de A. Esta definición la nombramos A' y al contexto donde A' se encuentra lo nombramos C'.

Si A' es de tipo clase, ya tenemos el tipo deseado; retornamos A'. En cambio si A' es un sinónimo de tipo, primeramente debemos expandir C' (Sección 3.2.1).

Como sabemos, un sinónimo de tipo se compone por un nombre, prefijos, arreglos y modificaciones. Estos últimos tres hay que reservarlos para luego aplicarlos. Con el nombre de tipo (puede ser una cadena de referencias de tipo $P.Q.R$), se ejecuta recursivamente el algoritmo de resolución de tipo.

Como resultado obtenemos un tipo t (que puede ser básico o de clase), prefijos de tipo y definiciones de arreglo. Si t es de tipo clase, expandimos t y le aplicamos las modificaciones que se encontraban definidas en A'. Por último, unimos las prefijos de tipos definidos en A' con los que obtuvimos al ejecutar nuevamente el algoritmo. Lo mismo realizamos con las definiciones de arreglos. Como resultado final de este ciclo del algoritmo devolvemos el tipo t (con las modificaciones que le habíamos realizado), los prefijos de tipos y los índices de array, si es que existen.

```
1  ResolveType(contex,type):
2    nameList = split(type,".");
3    foreach name in nameList:
4      typeDef = resolveDependencies(contex,name);
5      if (Type(typeDef) == Class)
6        contex = Type(typeDef);
7    end foreach;
8    return typeDef;
9
10 resolveDependencies(contex,name):
11   (class,type) = findTypeByName(contex,name);
12   Expand(class);
13   return getFinalClass(class,type);
14
15 findTypeByName(contex,name):
16   t = get(contex,name);
17   if (t) return (context,t);
18   else return findTypeByName(Father(contex),name);
19
20 getFinalClass(class,type):
21   if (isTypeDef(type)) then
22     t = ResolveType(class,Definition(Type));
23     if (isClass(t)) then
24       Expand(t);
25       ApplyModification(class,t,Modification(t));
26       AddPrefix(t,Prefix(type))
27       AddArray(t,Array(type))
28     end if;
29     return t;
30   else
31     return type;
```

Ejemplo 3.14: Resolución de tipos.

3.2.4. Remover Composiciones

En esta parte del algoritmo de aplanado ocurre el desglose de una instancia de una clase en varias variables y ecuaciones. De tener una instancia a un objeto, vamos a agregar variables, ecuaciones y sentencias según corresponda en la clase donde está definida. Para esto necesitamos referencia de un contexto C, la definición de la clase a reducir T, el nombre de la instancia (nombre de la variable con la que fue definida) y la información asociada a la definición de la instancia.

En el Ejemplo 3.15 vamos a remover la composición de la variable a en la clase *Root*. *Root* es el contexto, T es la clase A, el nombre de la instancia es a y la información asociada es que tiene definida un índice (de rango 10).

```
model Root
  model A
    Real x;
    Real y;
  equation
    x + 10 = y;
  end A;
  A a[10];
end Root;
```

Ejemplo 3.15: Ejemplo para RemoveComposition.

Dada toda esa información por cada variable v definida en T hacemos:

1. Le agregamos un prefijo a la variable para quitar ambigüedades. Este prefijo es el nombre de variable de la instancia más un guión. En el ejemplo mencionado tendríamos *a_x* y *a_y*.

2. Si la instancia a la clase está vectorizada, la variable v pasará a tener la misma dimensión que la instancia, o en el caso de que ya haya sido un arreglo se le agragará una dimensión más.

3. Si la variable v presenta una modificación de equidad, hay que verificar su expresión. Se aplican las mismas modificaciones que le vamos hacer a las ecuaciones, por eso dejamos la explicación para después. Si la variable va a ser vectorizada, debemos envolver la expresión dentro de la función *fill*, para que rellene todos los elementos del arreglo con ese valor (La función *fill(e,N)* es una función ad-hoc de Modelica que devuelve un array de N posiciones rellenado de la expresión e [11]).

4. Como último paso agregamos la variable v en el contexto C.

Por su parte, las ecuaciones y sentencias tampoco pueden ser agregadas sin modificaciones. De la misma forma que cambiamos los nombres de las variables y/o le agregamos dimensionalidad, debemos hacerlo en las ecuaciones y sentencias. Cada variable que está usada en las ecuaciones o sentencias y que fue renombrada en el punto anterior, también tiene que ser renombrada de la misma forma en la ecuación o sentencia.

Igualmente, si la instancia estaba vectorizada, cada una de estas variables deben estar vectorizadas, por lo que le vamos a agregar un índice de acceso de la forma i (el nombre debe cambiar para evitar ambigüedades con otra variable).

Por último este paquete de ecuaciones tiene que ser agregado en el contexto C. Si estábamos en el caso de una instancia vectorizada, creamos una ecuación *for* que contenga a todas las ecuaciones que están definidas. El rango de iteración de la ecuación *for* corresponde con la dimensionalidad de la instancia. Si la instancia no está vectorizada, se agregan las ecuaciones directamente en la clase C.

```
model Root
  Real a_x[10];
  Real a_y[10];
equation
  for i in 1:10 do
    a_x[i] + 10 = a_y[i];
  end loop;
end Root;
```

Ejemplo 3.16: Composición reducida.

3.3. Tratamiento de ecuaciones *for* y vectorización del modelo

Para mantener la vectorización del modelo se fueron aplicando diferentes técnicas a lo largo de la primera etapa de aplanado; así quedo implícito cómo resolver este problema. En esta sección vamos a agrupar y mencionar puntualmente qué partes del aplanado ayudan al fin principal de este trabajo.

Además de tratar las ecuaciones *for* también debemos mantener vectorizadas las definiciones de variables para que el modelo no pierda sentido. La primera clave la encontramos a la hora de reducir las composiciones de clases, es decir, cuando reducimos instancias de clases a múltiples definiciones de variables. En esta etapa podemos observar que si una instancia estaba vectorizada, todas las variables que están definidas en ellas van a ser agregadas vectorizadas de la misma forma que la instancia estaba. Además, si una variable interna ya estaba vectorizada, esta propiedad se mantendrá.

En el Ejemplo 2.16, el modelo *LC_line* tiene una instancia vectorizada de la clase *LC_circuit*. Al reducir esta composición agregamos instancias vectorizadas de la variable *p1_v*, *p1_i*, *p2_v*, *p2_i* entre otras más (además realizamos los renombramientos de variables). En el Ejemplo 3.17 observamos un fragmento de las definiciones de variables resultante del aplanado.

```
package Circuits
  model LC_line
    constant Integer N = 10;
    ground gr;
    Real lc_p1_v[N];
    Real lc_p1_i[N];
    Real lc_p2_v[N];
    Real lc_p2_i[N];
    ...
  end LC_line;
end Circuits;
```

Ejemplo 3.17: Fragmento de definiciones de variables expandidas.

Por otra parte, al reducir estas instancias vectorizadas las ecuaciones que ésta posea también tienen que ser tratadas. Para comenzar, vimos con anterioridad que todas estas ecuaciones son agregadas dentro de una sola ecuación *for* que itera entre 1 y el tamaño de la dimensionalidad definido. Luego, por cada ecuación se analiza en busca de referencias a variables y siempre y cuando éstas sean variables locales, se le agrega el prefijo y el índice de vectorización. Así tenemos, sin expandir, N conjuntos de ecuaciones que representan las que estaban dentro de las instancias. En el Ejemplo 3.18 observamos como quedan las ecuaciones del modelo *LC_circuit* dentro del modelo *LC_line* al ser aplanado.

```
1  package Circuits
2    model LC_line
3      ...
4    equation
5      for i in 1:N loop
6        connect(lc_ind_p[i],lc_p3[i]);
7        connect(lc_ind_p[i],lc_cap_p[i]);
8        connect(lc_cap_n[i],lc_p1i[]);
9        connect(lc_ind_n[i],lc_p2[i]);
10     end for;
11
12   end LC_line;
13 end Circuits;
```

Ejemplo 3.18: Uso de ecuaciones *for* al aplanar.

Por último tenemos que analizar qué sucede con las ecuaciones que ya existían en el modelo original. En el Ejemplo 2.16 la clase *LC_line* tiene dos ecuaciones *for*. A estas ecuaciones, simplemente, hay que hacerles un renombramiento de variables para que coincidan con el nombre de las variables que actualmente posee la clase. El Ejemplo 3.19 muestra como quedan las ecuaciones mencionadas.

```
1  package Circuits
2    model LC_line
3      ...
4    equation
5
6      for i in 1:N - 1 loop
7        connect(lc_p3[i + 1],ind_p2[i]);
8      end for;
9      for i in 1:N loop
10       connect(gr_p[i],lc_p1[i]);
11     end for;
12   end LC_line;
13 end Circuits;
```

Ejemplo 3.19: Uso de ecuaciones *for* al aplanar.

3.4. Complejidad

En esta sección no pretendemos hacer ningún análisis formal sobre la complejidad computacional, sino que vamos a dar una explicación informal de la complejidad de la etapa de aplanado y a exponer las ventajas que presenta frente a otros algoritmos. Para que quede mejor fundado, en el capítulo 6 haremos una comparación, con varios ejemplos, entre la implementación de este algoritmo con el algoritmo de aplanado de la herramienta OpenModelica y vamos a observar las diferencias que existen entre cada uno.

Esta etapa del aplanado se repite por cada declaración de variable de tipo clase que haya en el modelo. Si consideramos que la copia de clases y variables se hace en tiempo constante, podemos deducir que el algoritmo se encuentra en orden lineal con respecto a la cantidad de declaraciones que haya.

Cabe destacar que el costo computacional no aumenta si aumentamos algún parámetro del modelo, ya que consideramos un arreglo o una ecuación *for* como un único elemento, es decir, mantenemos el parámetro. Por ello si consideramos modelos de tamaño variable definidos por un parámetro, la complejidad es constante en función de este tamaño.

3.5. Ejemplos

A continuación veremos un ejemplo práctico. El Modelo 3.20, tomado de [18], representa un edificio de N habitaciones cada una de ellas con su propio aire acondicionado. La idea del modelo es analizar el consumo de energía eléctrica que demandaría el edificio.

El modelo se constituye de una clase que representa una habitación (*Room*) y otra que representa un aire acondicionado (AC). Luego, una tercera clase (*CoolRoom*) compuesta de una habitación y un aire acondicionado, conecta tanto la corriente eléctrica como la entrada y salida de aire de estos dos elementos. Por último, la clase raíz define N de estas habitaciones y lleva control de la energía consumida.

En la línea 60 y 61 del Modelo 3.20 observamos la definición del parámetro N y como se definen las habitaciones en función de éste. Solamente con cambiar el valor de N, aumentamos o disminuimos el tamaño del modelo. La Figura 3.1 muestra la representación gráfica de esta situación.

```
1 model Airconds
2   connector ThermalConnector
3     Real temperature;
4   end ThermalConnector;
5
6   connector PowerConnector
7     Real power;
```

```
8   end PowerConnector;
9
10  model AC
11    parameter Real Pot(fixed = false);
12    discrete Real TRef(start = 20);
13    ThermalConnector th_in;
14    PowerConnector pow_out;
15    discrete Real on;
16  initial algorithm
17    Pot := 13 + rand(0.2);
18  equation
19    pow_out.power = on * Pot;
20  algorithm
21    when th_in.temperature - TRef + on - 0.5 > 0 then
22      on := 1;
23    elsewhen th_in.temperature - TRef + on - 0.5 < 0 then
24      on := 0;
25    end when;
26    when time > 1000 then
27      TRef := 20.5;
28    end when;
29    when time > 2000 then
30      TRef := 20;
31    end when;
32  end AC;
33  model Room
34    parameter Real Cap(fixed = false);
35    parameter Real Res(fixed = false);
36    parameter Real THA = 32;
37    ThermalConnector th_out;
38    PowerConnector ac_pow;
39    Real temp(start = 20);
40    discrete Real noise;
41  initial algorithm
42    Cap := 550 + rand(100);
43    Res := 1.8 + rand(0.4);
44  equation
45    th_out.temperature = temp;
46    der(temp) * Cap = THA / Res - ac_pow.power - temp / Res +
          noise / Res;
47  algorithm
48    when sample(0, 1) then
49      noise := rand(2) - 1;
50    end when;
51  end Room;
52  model CoolRoom
53    Room room;
54    AC ac;
55  equation
56    connect(ac.th_in, room.th_out);
57    connect(ac.pow_out, room.ac_pow);
58  end CoolRoom;
59
60  constant Integer N = 10;
```

```
61   CoolRoom rooms[N];
62   Real Power;
63 equation
64   Power = sum(rooms.ac.pow_out.power);
65 end Airconds;
```

Modelo 3.20: Modelo de habitaciones con sistema de aire acondicionado.

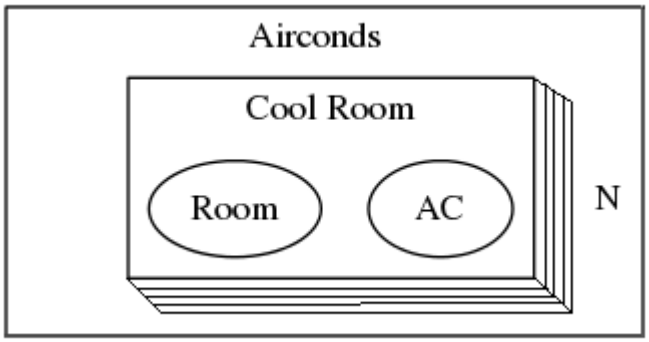

Figura 3.1: Representación gráfica del modelo 3.20.

El Modelo A.1 es la versión aplanada del modelo original. Observamos como se mantuvo el parámetro N (definido en la línea 2) y las variables, ecuaciones y sentencias quedaron escritas en función de N.

Concluimos entonces, que ni el costo computacional ni el tamaño del modelo aplanado dependen de la cantidad de instancias que se establecieron de *CoolRoom* en el modelo original.

Vemos también que el modelo sólo presenta variables de tipos básicos (sin clases ni conectores) ni posee herencia. Se puede ver que por cómo fue construida la versión aplanada (una sucesión de cambios semánticamente equivalentes) esta última representa el mismo modelo que el original.

Capítulo 4

Algoritmo de Aplanado - Segunda Etapa

En este capítulo describimos la segunda etapa del algoritmo de aplanado, consistente en reemplazar las expresiones *connect* por las ecuaciones correspondientes. Proponemos un algoritmo para detectar componentes conexas en un grafo bipartito para realizar el objetivo de esta etapa. Este grafo mantiene, mediante etiquetas en sus nodos y aristas, la información referida a las ecuaciones *connect*.

4.1. Problema a resolver

Después de aplicar la primer etapa, nos queda una única clase con todas las definiciones de variables y ecuaciones. En principio podríamos dar a esta clase como aplanada, pero para que sea compatible con las siguientes etapas de compilación, debemos reducir las ecuaciones *connect*. Para reducir estas ecuaciones, debemos determinar conjuntos de elementos que están conectados.

Lo complejo del algoritmo aparece cuando estos elmentos están vectorizados y la conexión entre dos elementos puede darse en un subconjunto del dominio total de alguno de los elementos. Para resolver esto, podríamos expandir las ecuaciones vectorizadas (ecuaciones *for*) y aplicar una búsqueda en profundidad para encontrar todas las componentes conexas [13]. Sin embargo, carece de sentido aplicar esta técnica, ya que buscamos mantener las ecuaciones *for* en el modelo. En consecuencia, agregamos etiquetas especiales al grafo para representar la vectorización del modelo y así reducir a una arista lo que podrían haber sido N en el grafo. A este grafo lo denominamos vectorizado.

A continuación, explicamos las tres etapas para resolver las ecuaciones *connect*.

4.2. Algoritmo

El algoritmo de resolución de conexiones se devide en tres etapas:

1. Generación de un grafo bipartito a partir de los connects.
2. Determinación de componente conexas del grafo generado.
3. Generación de ecuaciones a partir de las soluciones del punto anterior.

4.2.1. Grafo Bipartito

Para la resolución del problema vamos a crear un grafo bipartito no dirigido a partir de las ecuaciones *connect*. Cada variable que aparezca dentro de un *connect* va a ser un nodo y también por cada ecuación *connect* agregamos otro.

Por otro lado, va a existir una arista entre dos nodos, un nodo *variable* y un nodo *Connect*, si sólo si, existe una ecuación Connect con esa variable. De esta forma, las dos variables dentro de un Connect quedan comunicadas a través de un nodo *Connect*.

Tanto las aristas como los nodos van a estar etiquetados con datos particulares al contexto. Cada nodo *connect* lleva referencia si estaba dentro de una ecuación *for* y, en caso afirmativo, almacena el rango de la variable iteradora de la ecuación *for*. En las aristas guardamos el índice de acceso de la variable relacionada con el connect.

Una conexión involucra a dos variables. Dentro del grafo, queda representada por tres nodos, dos nodos variables v y v' y un nodo *Connect* c, y dos aristas, r y r', que unen estos tres nodos. Según este escenario definimos a:

- v' como el nodo complementario de v pasando por la arista r.
- r' como la arista complementaria de r.

```
1  model Root
2    Real a[10],d[10],c[10],d[10];
3  equation
4    for i in 1:10 do
5      connect(a[i],b[i])
6    end loop;
7    for i in 1:5 do
8      connect(c[i],d[i])
9    end loop;
10 end Root;
```

Ejemplo 4.1: Modelo con ecuaciones Connect.

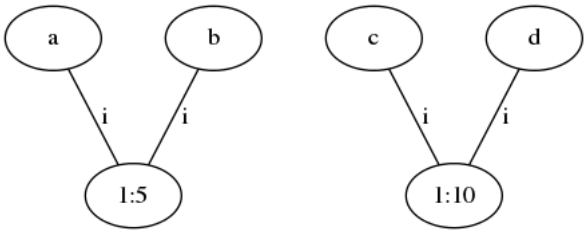

Figura 4.1: Ejemplo de Grafo Bipartito.

Para armar el grafo presentado, recorremos cada ecuación de la clase y cuando encontramos una ecuación *connect*:

1. Agregamos un nodo por cada variable. Considerar las variables marcadas con el signo menos como deferente a aquella que no lo presente.

2. Agregamos al nodo que representa a la ecuación *connect*.

3. Si la ecuación estaba dentro de un *for*, agregamos el rango de iteración como metadato del nodo *connect*.

4. Agregamos dos aristas, entre cada variable y el nodo que representa al *connect*.

5. Por cada arista, si la variable asociada tiene un índice de acceso, agregamos esa referencia a la arista.

6. Normalizamos la variable iteradora, es decir, llevamos a todas al mismo nombre de variable.

7. Eliminamos la ecuación *connect*, y si la ecuación *for* queda vacía, también.

La Figura 4.1 es el grafo asociado al modelo presentado en el Ejemplo 4.1.

4.2.2. Determinación de conexiones

En esta sección vamos a desarrollar el algoritmo para de determinar conjuntos de variables conectadas entre sí, de forma que se mantenga su vectorización y así evitar expandir las iteraciones de ecuaciones.

Empecemos analizando un caso acotado e iremos agregando mayor complejidad.

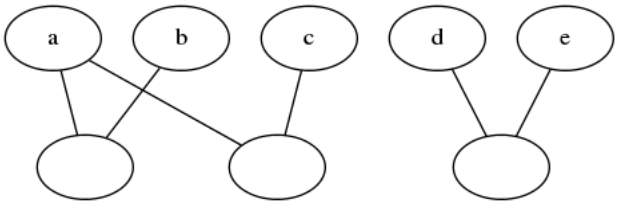

Figura 4.2: Grafo asociado al Modelo 4.2.

```
1  model Root
2    connect Pin
3      Real a;
4      flow Real v;
5    end Pin;
6    Pin a,b,c,d,e;
7  equation
8    connect(a,b);
9    connect(a,c);
10   connect(d,e);
11 end Root;
```

Ejemplo 4.2: Modelo con Connect Simple.

El Ejemplo 4.2 no presenta variables iteradas, por lo tanto es fácil determinar la conexiones de las variables. Viendo el grafo de conexiones en la Figura 4.2 y aplicando una búsqueda en profundidad (DFS) para encontrar las componentes conexas, obtenemos como resultado dos componentes: $\langle a, b, c\rangle$ y $\langle d, e\rangle$.

En casos simples, en los cuales el modelo no presenta dimensionalidad, basta con aplicar un algoritmo de resolución de componentes conexas. Ahora bien, qué pasa si agregamos vectorización a las aristas en el ejemplo anterior, es decir, partimos del Modelo 4.3.

```
1  model Root
2    connect Pin
3      Real a;
4      flow Real v;
5    end Pin;
6    Pin a[10],b[10],c[10];
7  equation
8    for i in 1:10 loop
9      connect(a[i],b[i]);
10     connect(a[i],c[i]);
11   end loop;
12 end test;
```

Ejemplo 4.3: Modelo con *connect* dentro de un *for*.

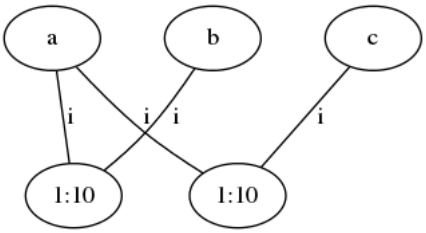

Figura 4.3: Grafo asociado al Modelo 4.3.

Si vemos el grafo de la Figura 4.3, notamos que es parecido al grafo de la Figura 4.2, con la diferencia que en los nodos *connect* tenemos los rangos de vectorización. Esto indica que existen, en este ejemplo, 10 caminos posibles entre a y b, y entre a y c. Analizando esto, a la hora de aplicar DFS para buscar las componentes conexas del grafo, no basta con que exista una arista entre dos nodos para visitarlo. Resulta fundamental tener en cuenta si existe intersección entre los rangos de visita.

En el grafo de la Figura 4.3, si arrancamos visitando el nodo a, vemos que existen dos aristas de salida con el mismo rango hacia los nodos b y c. Luego, desde los nodos b y c vemos que no existen otros caminos posibles por recorrer. De esta forma, a queda conectado con b y c con rango 1:10. La componente conexa sería:

1. $\langle a, b, c \rangle$ entre 1:10

Ahora vamos a analizar un ejemplo más complejo que posee dos ecuaciones *for* con diferentes rangos. El modelo se puede ver en el Ejemplo 4.4 y su grafo asociado en la Figura 4.4.

```
1  model Root
2    connect Pin
3      Real a;
4      flow Real v;
5    end Pin;
6    Pin a[10],b[10],c[10],d[10];
7  equation
8    for i in 1:5 loop
9      connect(a[i],b[i]);
10   end loop;
11   for i in 1:10 loop
12     connect(b[i],c[i]);
13     connect(b[i],d[i]);
14   end loop;
15 end Root;
```

Ejemplo 4.4: Modelo con ecuaciones *connect* con varias variables y rango distintos.

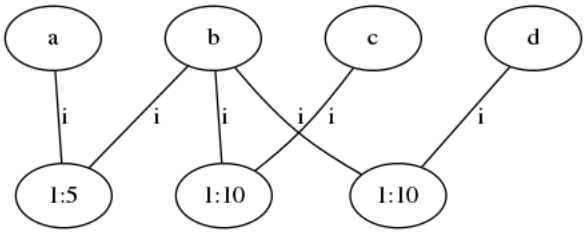

Figura 4.4: Grafo asociado al Modelo 4.4.

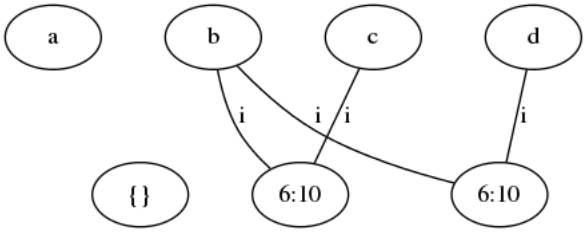

Figura 4.5: Grafo asociado al Modelo 4.4 después de visitar al nodo b desde a.

Arrancamos la búsqueda en profundidad por el nodo a (el resultado debe ser el mismo si arrancamos desde cualquier nodo). Observamos que existe un único camino hacia el nodo b con el intervalo 1:5, entonces visitamos el nodo b en dicho intervalo. Esto quiere decir que estamos visitando al nodo b simultáneamente por las aristas (índices): 1,2,3,4,5.

Ahora el objetivo del algoritmo es buscar qué caminos existen que comuniquen b con otros nodos en el intervalo 1:5. En este ejemplo, tanto c como d tienen una porción de sus aristas posible. Entonces visitamos c y d por las aristas 1:5. Por cada visita a un nodo, podamos la porción del grafo que utilizamos. En el ejemplo, hubiesemos podado las aristas ab,bc y bd en el intervalo 1:5. La Figura 4.5 representa el grafo después de ser podado.

Cuando visitamos un nodo resolvemos un subconjunto del problema mayor. Al retornar de la visita de c y d, tenemos dos soluciones a subproblemas. Debemos combinar estas soluciones para conformar la soluciones del problema que envuelve a b. En este caso, la solución de visitar b en el intervalo 1:5 es: $\langle b, c, d\rangle$ entre 1:5.

Si analizamos el grafo resultante en la Figura 4.5, veremos que quedaron aristas sin podar. Desde el nodo b podemos visitar de vuelta los nodos c y d con el rango 6:10 y así obtenemos otra componente conexa que conforma otra solución del problema total. En definitiva, obtenemos dos componentes conexas:

1. $\langle a, b, c, d\rangle$ entre 1:5
2. $\langle b, c, d\rangle$ entre 6:10

Ahora analicemos un modelo un poco más complejo.

En el modelo en el Ejemplo 4.5, los intervalos en la variable b son disjuntos. En la Figura 4.6 vemos el grafo asociado a este ejemplo. Con este ejemplo vamos a observar que una visita a un nodo puede implicar más de un camino solución. Si arrancamos desde el nodo a, visitamos a b por el intervalo 1:5; luego, desde el nodo b no es tan simple elegir los nodos para visitar, porque la instersección de todos los intervalos de salida es disjunta. Tenemos que ir chequeando subconjuntos de aristas que hagan que la intersección de sus intervalos con el intervalo de visita al nodo no sea disjunta. Cabe destacar que tenemos que comenzar desde el subconjunto más grande e ir bajando en cantidad de elementos.

```
1  model Root
2    connect Pin
3      Real a;
4      flow Real v;
5    end Pin;
6    Pin a[5],b[5],c[5],d[5];
7  equation
8    for i in 1:5 loop
9      connect(a[i],b[i]);
10   end loop;
11   for i in 1:3 loop
12     connect(b[i],c[i]);
13   end loop;
14   for i in 4:5 loop
15     connect(b[i],d[i]);
16   end loop;
17 end test;
```

Ejemplo 4.5: Modelo con diferentes rangos de Conexión.

En el ejemplo que estamos analizando, el subconjunto más grande ya lo analizamos. En el paso siguiente tenemos dos subconjuntos de un elemento que cumplen con la condición. Cuál elijamos primero no va a alterar la solución.

Así, elegimos la arista que va al nodo c. La intersección entre 1:3 y 1:5 da 1:3, por lo que visitamos el nodo c en el intervalo 1:3 y obtenemos una sola solución a la cual le agregamos a b. El intervalo solución es aquel que obtuvimos cuando buscamos si la instersección de los intervalos del subconjunto no era nula.

Luego de realizar el mismo procedimiento con d, las soluciones de visitar el nodo b son:

1. $\langle b, c \rangle$ entre 1:3
2. $\langle b, d \rangle$ entre 4:5

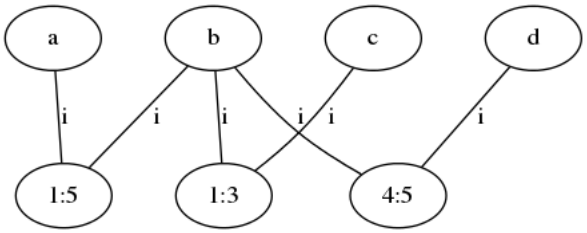

Figura 4.6: Grafo asociado al Modelo 4.5

Cuando retornamos de la visita a b con estas dos soluciones, nos queda agregar al nodo a en las dos soluciones.

En resumen, el algoritmo se basa en una búsqueda en profundida donde la visita a un nodo no está determinada sólo por la conexión de una arista sino que depende también del rango de la misma. Se pueden visitar simultáneamente varios nodos o varios por separados; hasta un mismo nodo puede ser visitado varias veces en la misma iteración. Cada vez que visitamos a un nodo podamos la parte del grafo que utilizamos.

Corrimientos de intervalo

Hasta ahora analizamos muchos casos en los cuales los índices de las variables eran simples. Pero qué ocurriría si en una de las variables de las ecuaciones *connect*, el índice de la misma tiene alguna modificación. Para entender esto, observemos el Ejemplo 4.6.

```
1 for i in 1:5 loop
2   connect(a[i],b[i]);
3   connect(b[i + 3],c[i]);
4 end for;
```

Ejemplo 4.6: Modelo con Corrimiento de Intervalo.

Apreciamos que en la segunda ecuación tenemos un corrimiento de intervalo en la variable b, es decir, el índice de la variable es de la forma $i + k$. Esto tenemos que tenerlo en cuenta a la hora de calcular intervalos.

Por ejemplo, si arrancamos visitando el nodo b y buscamos el intervalo de intersección entre sus dos caminos, tenemos que entre b y a el intervalo que afecta b es 1:5, y entre b y c el intervalo es 1 + 3: 5 + 3, es decir, 4:8. Entonces al realizar la intersección obtenemos el intervalo 4:5, es decir, el nodo b queda enlazado al mismo tiempo con a y c en los índices 4 y 5. Queda claro que aunque el rango de la ecuación *for* es 1:5, el intervalo de salida de la variable b en esa ecuación *connect* es otro. Este cambio en el índice de las variables lo conocemos como corrimiento de intervalo. Terminando de analizar el ejemplo, nos quedan dos caminos más, donde ambos nodos están

mutuamente excluidos. Uno es a con b entre 1:3, y el otro es b con c entre 6:8.

Estos corrimientos también afectan en otras partes del algoritmo. En el ejemplo anterior, si arrancamos visitando nodo c, el intervalo de intersección con b es 1:5; pero al visitar el nodo b hay que tener en cuenta el corrimiento que posee en el índice, por lo que el intervalo se transforma a 4:8 (1 + 3:5 + 3). Desde el nodo b con el intervalo 4:8, buscamos la intersección con a, la cual resulta ser 4:5. Notamos la simulitud en las soluciones aunque cambiamos el nodo de inicio.

Por último, debe tenerse en cuenta que al regresar de visitar el nodo b la subsolución está en función del corrimiento *+ 3*; motivo por el cual tenemos que aplicar un corrimiento inverso para reescribir las soluciones en función de c. Esto se trata de restarle 3 al intervalo y agregarle *+ 3* a los índices de las variables.

La solución para el nodo c quedaría:

1. $\langle c, b+3, a+3 \rangle$ entre 1:2
2. $\langle c, b+3 \rangle$ entre 3:5

En principio, esta es la variante más básica que podemos encontrar sobre un índice. Los casos de estudio de esta tesina llegan hasta ese punto. Cuando la función que modifica un índice se vuelve más compleja se producen conexiones muy alternadas que en muchos casos no queda otra opción que expandir la ecuación. El estudio de técnicas para evitar esta expansión cuando nos encontramos con índices más complejos quedó fuera del marco de la tesina. Se propone como trabajo a futuro.

Para facilitar la definición del algoritmo principal y teniendo en cuenta los corrimientos de intervalos, definimos a continuación un conjuntos de funciones para facilitar el manejo de intervalos:

1. **proyIntervalo**: proyecta un intervalo según el índice de acceso a la variable.
$$proyIntervalo([q:r], i+k) = [q+k, r+k] \tag{4.1}$$
2. **proyIntervaloInv**: proyecta de forma inversa un intervalo según el índice de acceso a la variable.
$$proyIntervaloInv([q:r], i+k) = [q-k, r-k] \tag{4.2}$$
3. **getIntervalo**: devuelve el intervalo de una arista teniendo en cuenta el índice asociado a la arista y el intervalo del nodo conector.
4. **proySolución**: proyecta una solución según el corrimiento de una arista dada. Es decir, suma en el intervalo y resta en los índices. Si hacemos cuentas no cambiamos las soluciones, es otra forma de representarlos.

5. **proySoluciónInv**: proyecta una solución de forma inversa según el corrimiento una arista dada. Realiza las cuentas al revés.

Para interpretar estas funciones, es preciso analizar un ejemplo complejo.

El modelo en el Ejemplo 4.7 posee una ecuación *connect* con corrimientos en sus dos variables. Analizamos varias iteraciones del algoritmo para interpretar dónde se usa cada función:

```
1  model Root
2    connect Pin
3      Real a;
4      flow Real v;
5    end Pin;
6    Pin a[10],b[10];
7  equation
8    for i in 1:5 loop
9      connect(a[i + 3],b[i + 5]);
10   end for;
11   for i in 6:10 loop
12     connect(b[i],c[i]);
13   end for;
14 end Root;
```

Ejemplo 4.7: Modelo de muestra.

1. Visitamos el nodo *a*.
2. Encontramos una arista con intervalo 4:8 (getIntervalo).
3. Visitamos el nodo *b* con intervalo 1:5. Resultado de aplicar la función proyIntervaloInv(4:8 , i + 3), ya que el nodo tenía corrimiento.
4. Proyectamos 1:5 según el índice i + 5 (proyIntervalo([1:5],i + 5) = [6:10]).
5. Buscamos intersección de intervalos con aristas salientes de *b*. Encontramos a *c*.
6. Al visitar el nodo *c* obtenemos como solución a: *c* entre 6:10.
7. No quedan más nodos por recorrer en *b*. Devolvemos la solución $< b, c >$ entre 6:10. Pero como visitamos el nodo por una arista con índice *i + 5*, debemos reescribir la solución en función de esta arista: $< b + 5, c + 5 >$ entre 1:5 (proySoluciónInv).
8. Cuando retornamos del nodo *b*, las soluciones deben ser reescritas en función del índice *i + 3*, por si existe colisión entre otras soluciones de otros nodos (no es el caso). Al aplicar proySolución obtenemos: $\langle b + 5 - 3, c + 5 - 3 \rangle$ entre 4:8.

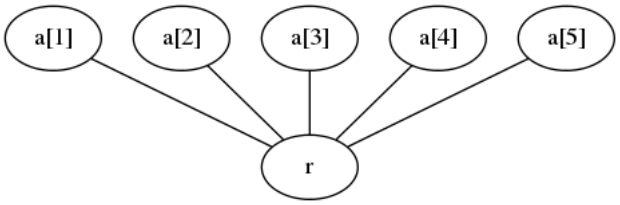

Figura 4.7: Grafo del Modelo 4.8 expandido

9. Por último agregamos a a al conjunto solución y aplicamos proySoluciónInv, como sucedió con el nodo b. La solución final es: $\langle a+3, b+5, c+5\rangle$ entre [1:5].

Conexiones *N* a uno

Un hecho a tener en cuenta, es que en una ecuación *connect* puede suceder que n elementos de una variables vectorizadas estén conectadas con un único elemento. Analizando el Ejemplo 4.8, vemos que los cinco índices de a están conectados con r, que a su vez y debido a esta conexión, cada índice de a está conectado con todos los demás elementos de a. En el grafo de la Figura 4.7 se puede ver cómo quedan las conexiones.

```
1 for i in 1:5 loop
2   connect(a[i],r);
3 end for;
```

Ejemplo 4.8: Multiples conexiones a uno.

Este tipo de ecuación hace que no se pueda mantener la ecuación *for* para reducir la conexión, ya que existe una relación todos con todos. Durante el algoritmo hay que detectar estos sucesos y llevar referencia de este tipo de solución, ya que a la hora de generar las ecuaciones en el caso de las variables de flujo en particular no se puede agregar una ecuación *for*. Se debe agregar una sola ecuación donde se suma cada componente.

Combinación de soluciones

Es importarte aclarar qué ocurre cuando tenemos un intervalo de visita I y visitamos varios nodos. Cada visita a un nodo nos va a devolver un conjunto de subsoluciones, ahora debemos combinar éstas para generar soluciones al problema mayor.

Sabemos que las subsoluciones de un mismo nodo son disjuntas (son diferentes componentes conexas), entonces por cada subsolución debemos intersectarla con las subsoluciones de los otros nodos. Con intersectarlas nos referimos a comprobar si la intersección de los intervalos no es nula.

En conclusión, hay que buscar todas las combinaciones de posibles subsoluciones y quedarnos con aquellas que el intervalo no sea vacío. Hay que aclarar que si una de estas soluciones es del tipo N a uno, no podemos calcular la intersección pero vamos a actuar como si fuese no vacía.

```
1  for i in 1:10 loop
2    connect(a[i],b[i]);
3    connect(a[i],c[i]);
4  end for;
5  for i in 1:5 loop
6    connect(b[i],d[i]);
7    connect(c[i],p[i]);
8  end for;
9  for i in 6:10 loop
10   connect(b[i],e[i]);
11   connect(c[i],q[i]);
12 end for;
```

Ejemplo 4.9: Modelo con varias soluciones.

En el Ejemplo 4.9, si arrancamos por el nodo a tenemos que visitar los nodos b y c en el intervalo 1:10. Como resultado vamos a obtener los siguientes dos conjuntos de subsoluciones:

1. Nodo b:
   a) $\langle d \rangle$ entre 1:5.
   b) $\langle e \rangle$ entre 6:10.
2. Nodo c:
   a) $\langle p \rangle$ entre 1:5.
   b) $\langle q \rangle$ entre 6:10.

Para determinar las soluciones finales, tenemos que combinar estas subsoluciones. Las combinaciones posibles son 4:

1. d entre 1:5 con p entre 1:5.
2. d entre 1:5 con q entre 6:10. Intersección vacía.
3. e entre 6:10 con p entre 1:5. Intersección vacía.
4. d entre 6:10 con q entre 6:10.

Quedándonos con las soluciones cuyos intervalos no son vacíos y agregando el nodo a, las soluciones resultan ser:

1. $\langle a, d, p \rangle$ entre 1:5.
2. $\langle a, q, e \rangle$ entre 6:10.

Algoritmo General

Dado un nodo v, una arista e adyacente a v y un intervalo P, la función VisitarNodo se define así:

1. Si e está definido (es decir, venimos desde una arista) podamos el grafo según el intervalo de visita.
2. Determinamos el conjunto U más grande de aristas cuya intersección de intervalos no sea vacía e incluimos a P en la intersección. El intervalo obtenido lo denominamos I.
3. Si U es vacío saltamos al paso 8.
4. Por cada arista r de U:
   a) Buscamos el vértice complementario de v pasando por r, llamado v', y la arista r' complementaria a r.
   b) Calculamos la proyección inversa de I. Obtenemos I'.
   c) Visitamos el nodo v' desde la arista r' con el intervalo I'.
   d) Proyectamos las soluciones de la visita a v' según la arista r'.
5. Combinamos las soluciones de los diferentes nodos.
6. Para cada solución resultante agregamos el nodo v.
7. Restamos I a P ($P = P \cap I$).
8. Volvemos al paso 2.
9. Si P es distinto de vacío, agregamos al nodo v con el intervalo P como una solución simple.
10. Proyectamos de forma inversa cada solución según la arista e.

Este algoritmo debe ser aplicado sobre cada nodo del tipo *variable* del grafo para asegurarnos de abarcar todas las componentes conexas. Cada visita a un nodo nos va a devolver un conjunto de soluciones; la unión de todas estas soluciones es la respuesta final del algoritmo. Debe tenerse en cuenta que la visita a un nodo puede devolver una solución vacía, ya que ese nodo pudo haber sido visitado desde algún otro nodo y así provocar la podada del grafo.

En el paso 3 chequeamos si U es vacío. Esto nos indica que ya no existe una arista disponible para movernos sobre ella y visitar otro nodo. Al ocurrir esto termina la visita al nodo y devolvemos las soluciones que encontramos.

4.2.3. Generación de ecuaciones

Determinado el conjunto de conexiones, el siguiente paso es mecánico. Debemos agregar ecuaciones como las que vimos en la sección 2.1.6.

Un grupo de variables conectadas entre sí deben ser todas del mismo tipo, es decir, tienen las mismas variables de flujo y de potencial. Por eso, por cada variable que tenga el conector debemos agregar una o varias ecuaciones de la siguiente forma:

1. Suma de los flujos igual a cero: dada una variable de flujo, la suma de cada instancia de esta variable en el conjunto solución de conectores debe ser igual a cero.

2. Variables de potencial iguales: toda variable de potencial debe ser igual a sus pares de otros conectores. Agregamos varias ecuaciones de equidad según la cantidad de conectores que tengamos en la solución.

```
1 connector Pin
2   flow Real i,v;
3   Real p,r;
4 end Pin;
```

Ejemplo 4.10: Clase Conector.

También hay que mencionar que si estamos en presencia de una solución vectorizada, cada una de las ecuaciones que mencionamos anteriormente es agregada dentro de una ecuación *for* para mantener su iteración. Así mantenemos pequeña la descripción del modelo, evitando expandir cada una de las conexiones que presenta el modelo.

Por último, hay que tener en cuenta que todos aquellos conectores que están libres (sin conectar), sus variables de flujos deben ser igual a cero. Teniendo el modelo casi aplanado y observando el grafo desarrollado para resolver las conexiones, podemos identificar aquellos conectores que no hayan sido utilizados. Por un lado, si está declarado como variable pero no existe como nodo en el grafo, por otro lado puede existir en el grafo pero no haber estado conectado en su totalidad. Si llevamos referencia de las índices que conectamos sobre esa variable podemos determinar si quedó una parte libre.

Por cada una de estas variables que encontramos, debemos agregar una ecuación por cada variable de flujo que contenga donde éstas sean igual a cero. Si la variable está vectorizada la envolvemos en una ecuación *for*.

Suponiendo que todas las variables son del tipo Pin del Ejemplo 4.10 y la solución:

- $\langle c, b, a + 3 \rangle$ entre 1:10

La descomposición en ecuaciones quedaría como en el Ejemplo 4.11.

```
for i in 1:10 loop
  c_p[i] = b_p[i];
  b_p[i] = a_p[i + 3];

  c_r[i] = b_r[i];
  b_r[i] = a_r[i + 3];

  c_i[i] + b_i[i] + a_i[i + 3] = 0;
  c_v[i] + b_v[i] + a_v[i + 3] = 0;
end for;
```

Ejemplo 4.11: Ecuaciones generadas por los conectores.

Observamos que la variable p (variable de potencial) de cada instancia quedaron igualadas entre sí. Lo mismo ocurrió con la variable r. Por su parte, las variables de flujos se agruparon en una sumatoria igualada a cero.

En caso de que la solución sea del tipo N a uno, las ecuaciones de flujo son las que cambian. Como todos los elementos quedan unidos entre sí no podemos mantener la ecuación *for*, en cambio, debemos agregar una única ecuación que represente el flujo de la variable del conector. Para aquellas variables que están vectorizadas puede usarse una función que posee Modelica, llamada *sum*, para sumar los elementos de un vector. En el Ejemplo 4.12 podemos ver una conexión N a uno y la ecuación de flujo que generaría.

```
for i in 1:N loop
  connect(a[i],r);
end for;

// Ecuación generada
sum(a_i[1:N]) + r_i = 0;
```

Ejemplo 4.12: Ecuaciones N a uno.

4.3. Eliminación de variables

Como última etapa, después de reducir los conexiones, eliminamos aquellas variables que son instancias de clases de tipo conectores. Estas variables las habíamos dejado para tener referencia de qué tipo de conector eran y poder realizar la descomposición en diferentes ecuaciones. Una vez realizado esto, ya no son de más utilidad.

4.4. Complejidad

En el peor de los casos, cuando tenemos que buscar los conjuntos de intersección de los intervalos (paso 2 del algoritmo), debemos probar con todas las combinaciones de aristas posibles, generando esto una complejidad

del orden de $O(2^n)$. Pero en la práctica, vamos a aplanar modelos reales donde la cantidad de conexiones que puede tener un nodo no son más de tres o cuatro.

Por otro lado, como el algoritmo mantiene la vectorización del modelo, mantenemos constante el tiempo computacional aunque aumentemos los parámetros. Esta misma propiedad logramos en la primer etapa del aplanado.

4.5. Ejemplos

El Modelo 4.13 representa una línea de transmisión LC (circuito resonante compuesto por una bobina y un capacitor) utilizando el Método de Líneas [7]. Éste puede ser representado por N secciones de circuitos LC. Al principio de la línea se encuentra el suministro de energía *PulseVoltage*. Luego de realizar la primer etapa del aplanado, nos encontramos con el modelo A.2.

```
1  model trline
2    Modelica.Electrical.Analog.Sources.PulseVoltage pulsevoltage1;
3    Modelica.Electrical.Analog.Basic.Resistor resistor1;
4    Modelica.Electrical.Analog.Basic.Ground ground1;
5    constant Integer N = 10;
6    line_segment[N] line_segment1;
7  equation
8    connect(pulsevoltage1.n, resistor1.n);
9    for i in 1:N - 1 loop
10     connect(line_segment1[i].pin1, line_segment1[i + 1].pin0);
11   end for;
12   connect(line_segment1[1].pin0, pulsevoltage1.p);
13   connect(line_segment1[N].pin1, resistor1.p);
14 end trline;
15
16
17 model line_segment
18   Modelica.Electrical.Analog.Basic.Ground ground1;
19   Modelica.Electrical.Analog.Basic.Resistor resistor1;
20   Modelica.Electrical.Analog.Basic.Inductor inductor1;
21   Modelica.Electrical.Analog.Basic.Capacitor capacitor1;
22   Modelica.Electrical.Analog.Interfaces.Pin pin1;
23   Modelica.Electrical.Analog.Interfaces.Pin pin0;
24 equation
25   connect(pin0, resistor1.p);
26   connect(pin1, capacitor1.p);
27   connect(capacitor1.n, ground1.p);
28   connect(inductor1.n, capacitor1.p);
29   connect(resistor1.n, inductor1.p);
30 end line_segment;
```

Modelo 4.13: Conexión de N circuitos iguales.

La ecuaciones que nos interesan en esta sección se presentan en las líneas 115-120 y 92-97. La Figura 4.14 muestra las ecuaciones *connect* separadas del modelo aplanado. Podemos observar que a grandes rasgos existe una gran similitud entre las ecuaciones del modelo original y el resultado después de aplanar.

```
1  connect(pulsevoltage1_n , resistor1_n);
2  for i in 1:N-1 loop
3    connect(line_segment1_pin1[i] , line_segment1_pin0[i+1]);
4  end for;
5  connect(line_segment1_pin0[1] , pulsevoltage1_p);
6  connect(line_segment1_pin1[N] , resistor1_p);
7
8  for _Index_0 in 1:N loop
9    connect((-line_segment1_pin0[_Index_0]) ,
         line_segment1_resistor1_p[_Index_0]);
10   connect((-line_segment1_pin1[_Index_0]) ,
         line_segment1_capacitor1_p[_Index_0]);
11   connect(line_segment1_capacitor1_n[_Index_0] ,
         line_segment1_ground1_p[_Index_0]);
12   connect(line_segment1_inductor1_n[_Index_0] ,
         line_segment1_capacitor1_p[_Index_0]);
13   connect(line_segment1_resistor1_n[_Index_0] ,
         line_segment1_inductor1_p[_Index_0]);
14 end for;
```

Ejemplo 4.14: Ecuaciones *connect* en el modelo aplanado.

A través de estas ecuaciones y luego de generar el grafo bipartito, a simple vista notamos 6 componentes conexas, las cuales pueden observarse en las Figuras 4.8, 4.9, 4.10, 4.11, 4.12 y 4.13. Podemos observar que la variable *line_segment1_pin1* está repetida, aparece con y sin signo negativo. Esto es debido a la orientación del flujo.

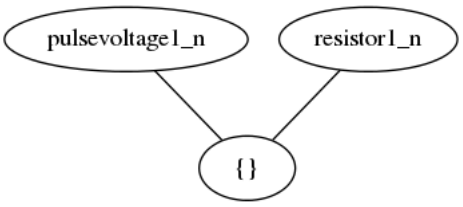

Figura 4.8: Componentes conexas del modelo *trline*.

Analizando más detenidamente estas componentes observamos, por ejemplo, que la Figura 4.9 presenta realmente 3 componentes conexas. Al aplicar el algoritmo para resolver las conexiones sobre este subconjunto del problema obtenemos tres soluciones:

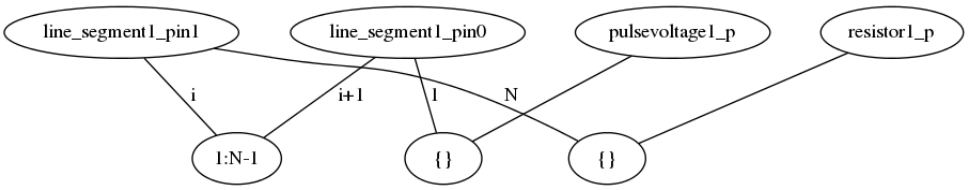

Figura 4.9: Componentes conexas del modelo *trline*.

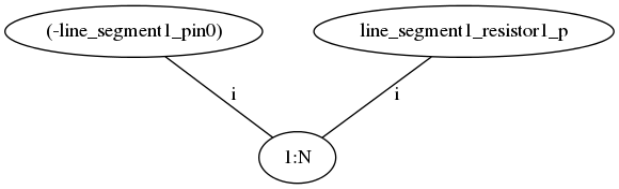

Figura 4.10: Componentes conexas del modelo *trline*.

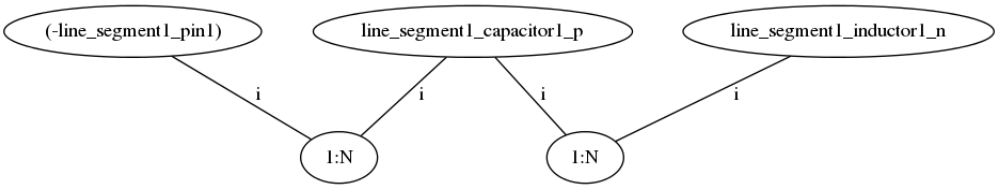

Figura 4.11: Componentes conexas del modelo *trline*.

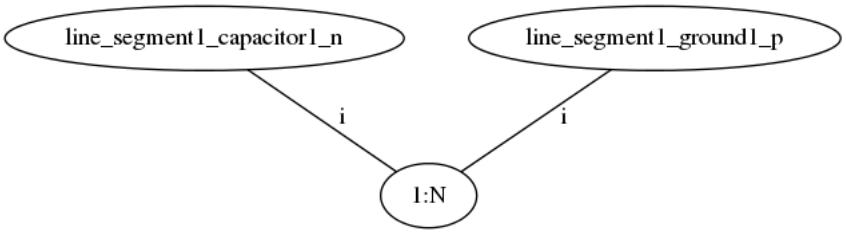

Figura 4.12: Componentes conexas del modelo *trline*.

1. $\langle line\_segment1\_pin0, line\_segment1\_pin1\rangle$ entre 1:9
2. $\langle resistor1\_p, line\_segment1\_pin1[10]\rangle$
3. $\langle pulsevoltage1\_p, line\_segment1\_pin0[1]\rangle$

En el Ejemplo4.15 podemos ver las ecuaciones generadas a partir de la primera solución de la componente que estábamos analizando. Estos conectores poseían una variable de potencial y otra de flujo, de ahí que tenemos una ecuación de cada tipo. Las variables de potencial igualadas (línea 2) y la suma de las variables de flujo igual a cero (línea 5).

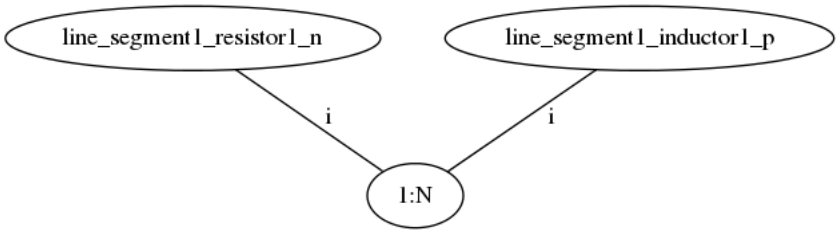

Figura 4.13: Componentes conexas del modelo *trline*.

```
1 for i in 1:9 loop
2   line_segment1_pin0_v[i+1] = line_segment1_pin1_v[i];
3 end for;
4 for i in 1:9 loop
5   line_segment1_pin0_i[i+1]+line_segment1_pin1_i[i] = 0;
6 end for;
```

Ejemplo 4.15: Ecuaciones obtenidas de la primer componente.

Por último, el modelo en el Anexo A.3 es el resultado de aplicar el proceso de aplanado y resolución de conexiones al Modelo 4.13.

Capítulo 5

Implementación del Aplanado

El algoritmo de aplanado fue implementado íntregramente en C++, el cual Pertenece al proyecto ModelicaCC [1], que contiene diversas herramientas para compilar los modelos y simularlos.

ModelicaCC arovecha la librería Boost[2] [14], que incluye la implementación eficiente de estructura de datos (listas, grafos, etc) y aporta una herramienta óptima para crear parsers; la cual fue utilizada para leer e interpretar los modelos.

Dentro de la implementación existe un objeto (clase de C++) que representa a las clases de Modelica. Ésta mantiene todo el contenido definido en ella:

1. Tabla de variables donde se almacena toda la información relevante de las variables (tipos, prefijos, índices, etc).
2. Tabla de tipo donde están definidas las clases y alias de tipo.
3. Una lista de ecuaciones.
4. Una lista de sentencias.

El modelo completo está representado con la estructura de datos árbol. Cada "clase" queda enlazada con la clase en la que está definida (su padre) y para abajo queda enlazada con las clases (sus hijos) que estén definidas dentro de ella. Existe una clase que es la raíz del árbol, es decir, no posee padre y existen varias clases que son las hojas del árbol (no poseen hijos). Dentro de cada nodo del árbol se definen los tipos, variables, ecuaciones, etc.

Los principales módulos implementados para el aplanado son:

[1] http://sourceforge.net/projects/modelicacc/
[2] http://www.boost.org/

1. **flatter.cpp**: implementa el algoritmo principal de aplanado (primera etapa).

2. **remove_composition.cpp**: contiene las funciones para sustituir una definición de variable de tipo clase por el conjunto de variables y ecuaciones que ésta contenía.

3. **classfinder.cpp**: se encarga de expandir las herencias de clases así como también de resolver nombres de tipos. Incluso aplica modificaciones.

4. **connectos.cpp**: contiene todos los algoritmos para crear el grafo bipartito y resolver los conjuntos conexos.

Además de estos módulos, dentro del proyecto ModelicaCC están implementadas las estructuras de datos utilizadas como tabla de tipos y variables, los tipos propiamente dichos, estructura de variables, el parser y AST del lenguaje, etc. Estos módulos no fueron desarrollados en el marco de la tesina, sino que son de uso general dentro del grupo de investigación.

También se hace uso del patrón del diseño Visitor [12], para desarrollar funciones que apliquen cambios en la estructura de las ecuaciones y sentencias. Por ejemplo, a la hora de agregarle el prefijo a las variables en las ecuaciones cuando removemos las composiciones.

Capítulo 6

Ejemplos y Comparaciones

En este capítulo analizaremos varios ejemplos y compararemos el algoritmo de aplanado desarrollado en este trabajo con la utilidad que pasea OpenModelica para realizar los aplanados.

Arrancamos analizando un modelo extraído de la propia librería de OpenModelica que no está vectorizado, comparamos el tiempo que demora aplanar con ambas herramientas y el espacio (en bytes) que ocupa el modelo resultante. Luego analizamos dos ejemplos que están vectorizados, realizando las mismas comparaciones que con el primer modelo pero variando los parámetros que determinan el tamaño del modelo.

6.1. Circuito de Chua

El primer ejemplo que analizamos es un modelo simple (sin vectorizar), conocido como el circuito de Chua [15]. Podemos observar que el Modelo 6.1 está compuesto por un par de componentes electrónicos que están conectados entre sí.

```
1  model ChuaCircuit
2    import Modelica.Electrical.Analog.Basic;
3    import Modelica.Electrical.Analog.Examples.Utilities;
4    Basic.Inductor L(L = 18, i(start = 0, fixed = true))
5    Basic.Resistor Ro(R = 0.0125)
6    Basic.Conductor G(G = 0.5649999999999999)
7    Basic.Capacitor C1(C = 10, v(start = 4, fixed = true))
8    Basic.Capacitor C2(C = 100, v(start = 0, fixed = true))
9    Utilities.NonlinearResistor Nr(Ga(min = -1) = -0.757576, Gb(
         min = -1) = -0.409091, Ve = 1)
10   Basic.Ground Gnd
11 equation
12   connect(L.n, Ro.p)
13   connect(C2.p, G.p)
14   connect(L.p, G.p)
15   connect(G.n, Nr.p)
```

```
16   connect(C1.p, G.n)
17   connect(Ro.n, Gnd.p)
18   connect(C2.n, Gnd.p)
19   connect(Gnd.p, C1.n)
20   connect(Gnd.p, Nr.n)
21 end ChuaCircuit
```

Modelo 6.1: Circuito de Chua.

Al aplicar ambas herramientas sobre el modelo obtuvimos que OpenModelica tardó 1 segundo y el modelo final ocupa 5800 bytes . Por su parte, ModelicaCC tardó 0.1 segundo y el modelo resultante ocupa 2500 bytes. Ambos modelos aplanados pueden versen en las Figuras A.4 y A.5. No existe una diferencia notoria en tiempo y espacio para este ejemplo.

6.2. Aires Acondicionados

El primer de los modelos vectorizados que vamos a analizar, es el modelo *Airconds* expuesto en la sección 3.5.

La diferencia de tamaño entre los Modelos A.7 y A.6 es notorio a simple vista. Sin perder semántica, nuestra herramienta mantuvo las definiciones de arreglos e introdujo ecuaciones y sentencias *for* para no repetir código. La Tabla 6.1 muestra los resultados de ir aumentando el valor de N. Analizamos el tiempo de compilación y el tamaño en bytes que ocupa el modelo final.

N	OpenModelica		ModelicaCC	
	Tiempo(seg)	Tamaño(bytes)	Tiempo(seg)	Tamaño(bytes)
10	0.160	15.814	0.012	2081
100	1.204	160.039	0.012	2084
500	19.440	818.439	0.012	2084
1000	85.912	1.641.484	0.012	2087
2000	606.480	3.331.484	0.012	2087
3000	2077.872	5.021.484	0.012	2087
4000	4597.380	6.711.484	0.012	2087
5000	10140	8.401.484	0.012	2087
10000	Error	Error	0.012	2090

Cuadro 6.1: Tiempos de Aplanado para distintos N para el Modelo 3.20

Aunque el tamaño final no representa un problema técnico, debemos recordar que el resultado del aplanado será luego alimentado a etapas subsiguientes de la compilación. Si el modelo resultante del aplanado es grande, luego será imposible de procesar.

Según los datos obtenidos, queda claro como nuestra herramienta permanece constante al tamaño del modelo, y ni el tiempo ni el espacio computacional se desbordan. Además de las estadísticas, se realizó la simulación

en OpenModelica del código obtenido con nuestra herramienta. El resultado obtenido es el mismo que simular el modelo original.

6.3. Línea de transmisión

El Modelo 6.2, que es una línea de transmisión diseñada con la herramienta OpenModelica, vamos a tomarla como segundo ejemplo vectorizado para realizar estadísticas.

Los modelos de los Anexos A.8 y A.9 muestran el resultado de aplanar el modelo, con el valor de N igual 3, con ambos algoritmos. También, incluimos la Tabla 6.2 con estadística sobre el tiempo de compilación y tamaño del modelo con diferentes valores de N. Como sucedió en el ejemplo anterior las diferencias en tiempo y espacio son notorias entre los dos algoritmos.

N	OpenModelica		ModelicaCC	
	Tiempo(seg)	Tamaño(bytes)	Tiempo(seg)	Tamaño(bytes)
10	3.792	33.212	0.048	3.708
100	5.632	302.374	0.052	3.723
500	19.440	818.439	0.044	3.725
1000	51.628	3.048.164	0.052	3.738
2000	179.232	6.160.336	0.052	3.740
3000	393.452	9.272.336	0.044	3.740
4000	704.552	12.384.336	0.052	3.740
5000	1107.732	15.496.336	0.052	3.740
10000	Error	Error	0.058	3.753

Otro nombre 6.2: Tiempos de Aplanado para distinto N para el Ejemplo 6.2

```
1  model lcline
2    model lcsection
3      Modelica.Electrical.Analog.Interfaces.Pin pin2;
4      Modelica.Electrical.Analog.Basic.Inductor inductor1;
5      Modelica.Electrical.Analog.Interfaces.Pin pin;
6      Modelica.Electrical.Analog.Interfaces.Pin pin1;
7      Modelica.Electrical.Analog.Basic.Capacitor capacitor1;
8    equation
9      connect(capacitor1.p, pin1);
10     connect(pin, inductor1.p);
11     connect(inductor1.n, capacitor1.p);
12     connect(capacitor1.n, pin2);
13   end lcsection;
14
15   Modelica.Electrical.Analog.Basic.Resistor resistor1;
16   Modelica.Electrical.Analog.Basic.Ground ground1;
```

```
17   constant Integer N = 10;
18   lcsection[N] lc;
19   Modelica.Electrical.Analog.Sources.ConstantVoltage
         constantvoltage1;
20 equation
21   connect(resistor1.n, ground1.p);
22   connect(constantvoltage1.n, ground1.p);
23   connect(constantvoltage1.p, lc[1].pin);
24   connect(resistor1.p, lc[N].pin1);
25   connect(ground1.p, lc[N].pin2);
26   for i in 1:N - 1 loop
27     connect(ground1.p, lc[i].pin2);
28     connect(lc[i].pin1, lc[i + 1].pin);
29   end for;
30 end lcline;
```

Modelo 6.2: Línea de transmisión.

Capítulo 7

Conclusiones y Trabajo a Futuro

El aplanado de modelos Modelica es un paso fundamental para simularlo. Existen diversas herramientas que implementan algoritmos de aplanadas, pero todas tienen problemas al tratar con modelos grandes. Nuestro objetivo fue obtener un algoritmo eficiente de aplanado de modelos Modelica que preserve ciertas propiedades y que pueda aplicarse a modelos grandes.

En esta tesina:

- Desarrollamos un algoritmo de aplanado que preserva la vectorización del modelo. Esto nos permite tratar con modelos grandes y generar un resultado vectorizado (por ende con una descripción breve) que podrá ser tratado por las etapas sucesivas de compilación. (Capítulo 3).

- Dentro de este algorimo, desarrollamos un métodos para encontrar las componente conexas dentro de un grafo bipartito vectorizado, que luego aplicamos para calcular las conexiones del modelo sin necesidad de expandir el grafo (Capítulo 4).

- Implementamos ambos algoritmos en C++ dentro de la herramienta ModelicaCC (Capítulo 5).

- Realizamos pruebas tanto de ejemplos simples como de ejemplos vectoriales, concluyendo que las transformaciones aplicadas por la herramienta desarrollada llegaban al resultado correcto.

- Comparamos nuestra implementación del algoritmo de aplanado con la de la herramienta OpenModelica para distintos tamaños de modelos vectorizados. Tomando como medida el costo computacional y espacial nuestra herramienta tiene una complejidad constante contra una supra-lineal de OpenModelica (Capítulo 6).

Además del desarrollo, queda un documento claro y detallado sobre el algoritmo de aplanado. Actualmente, el estándar Modelica no define ninguno y las herramientas, tanto libres como pagas, no presentaron formalmente su implementación.

Los resultados de este trabajo fueron enviados como artículo completo a la conferencia Modelica 2015 [8], el cual fue aceptado.

Sobre el futuro próximo, quedan varias ideas por investigar y aplicar:

- El algorimo de resolución de ecuaciones *connect* no acepta anidaciones de dos o más ecuaciones *for*. Para lidiar con esto, una opción es, previamente a aplicar la resolución de conexiones, reescribir las ecuaciones a un sistema de una única ecuación *for* (si es posible). Otra alternativa, más costosa, es expandir el algoritmo presentado en esta tesina para trabajar con varias dimensiones. En esencia, la idea sería la misma.

- Actualmente, utilizamos el entorno gráfico de OpenModelica para generar modelos. Ocurre que al grabar el modelo, los componentes utilizados de la librería *Modelica* no están presentes, solo están referenciados. La librería *Modelica* se carga dinámicamente al aplanar el modelo. OpenModelica presenta una opción que exporta el modelo total, la cúal utilizamos. Sugerimos como trabajo futuro, una herramienta capaz de cargar dinámicamente los componentes necesarios de la librería *Modelica* y así independizarnos de OpenModelica.

- Aplicar un caché de modelos aplanados con el objetivo de reducir a uno la cantidad de veces que aplanamos una misma clase.

- Estudiar la posibilidad de paralelizar el aplanado de clases. Si podemos determinar que dos clases son independientes entre sí, estas dos clases podrían ser aplanadas de forma paralela sin interferir en el resultado final.

- Al resolver las conexiones, si no tenemos en cuenta la vectorización del grafo, en éste quedan determinadas varias componentes conexas. Éstas son independientes entre sí, por lo tanto, se podría paralelizar el algoritmo de resolución de componentes conexas para grafos vectorizados.

Bibliografía

[1] Matthias Arzt, Volker Waurich, and Jörg Wensch. Towards utilizing repeating structures for constant time compilation of large modelica models. In *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, EOOLT '14, pages 35–38, New York, NY, USA, 2014. ACM.

[2] Modelica Association. Modelica - a unified object-oriented language for systems modeling - language specification version 3.3. Technical report, 2012.

[3] Umesh Bellur, Al Villarica, Kevin Shank, Imram Bashir, and Doug Lea. Flattening C++ Classes. New York CASE Center, Syracuse NY 13244, 1992.

[4] Federico Bergero, Xenofon Floros, Joaquín Fernández, Ernesto Kofman, and François E. Cellier. Simulating Modelica models with a Stand–Alone Quantized State Systems Solver. In *9th International Modelica Conference*, pages 237–246, Munich, Germany, 2012.

[5] Federico Bergero, Ernesto Kofman, and François E. Cellier. A Novel Parallelization Technique for DEVS Simulation of Continuous and Hybrid Systems. *Simulation: Transactions of the Society for Modeling and Simulation International*, 89(6):663–683, 2013.

[6] Dag Brück, Hilding Elmqvist, Sven Erik Mattsson, and Hans Olsson. Dymola for multi-engineering modeling and simulation. In *Proceedings of Modelica 2002*, 2002.

[7] François E. Cellier and Ernesto Kofman. Continuous system simulation. *Springer*, 2006.

[8] F.Bergero, M.Botta, E. Campostrini, and E. Kofman. Efficient compilation of large scale modelica models. In *11th International Modelica Conference*. Aceptado.

[9] Joaquín Fernández and Ernesto Kofman. Implementación autónoma de métodos de integración numeŕica qss. Master's thesis, Universidad Nacional de Rosario, 2012.

[10] Peter Fritzson. MathModelica-An Object-Oriented Mathematical Modeling and Simulation Environment. *Mathematica Journal*, 10(1):187, 2006.

[11] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach.* Willey, 2 edition, 2014.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Patrones de diseño.* Addison-Wesley, 2003.

[13] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[14] Björn Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost.* 1 edition, 2005.

[15] Kennedy. Three steps to chaos - part i: Evolution. *IEEE Transactions on Circuits And System*, 40(10):640–656, 1993.

[16] E. Kofman. Quantization–Based Simulation of Differential Algebraic Equation Systems. *Simulation: Transactions of the Society for Modeling and Simulation International*, 79(7):363–376, 2003.

[17] Kalil T. Swain Oldham. *The doctrine of description: Gustav Kirchhoff, classical physics, and the "purpose of all scienceïn 19th-century Germany.* PhD thesis, University of California,Berkeley, 2008.

[18] Cristian Perfumo, Ernesto Kofman, Julio Braslavsky, and John K. Ward. Load management: Model-based control of aggregate power for populations of thermostatically controlled loads. *Energy Conversion and Management*, pages 55:36–48, 2012.

Apéndice

Apéndice A

Modelos aplanados

A.1. Modelo *Airconds* primera etapa de aplanado

```
model Airconds
  constant Integer N=10;
  parameter Real rooms_room_Cap[N](fixed= false );
  parameter Real rooms_room_Res[N](fixed= false );
  parameter Real rooms_room_THA[N]= fill(32,N);
  Real rooms_room_th_out_temperature[N];
  ThermalConnector rooms_room_th_out[N];
  Real rooms_room_ac_pow_power[N];
  PowerConnector rooms_room_ac_pow[N];
  Real rooms_room_temp[N](each start=20);
  discrete Real rooms_room_noise[N];
  parameter Real rooms_ac_Pot[N](fixed= false );
  discrete Real rooms_ac_TRef[N](each start=20);
  Real rooms_ac_th_in_temperature[N];
  ThermalConnector rooms_ac_th_in[N];
  Real rooms_ac_pow_out_power[N];
  PowerConnector rooms_ac_pow_out[N];
  discrete Real rooms_ac_on[N];
  Real Power;
initial algorithm
    for _Index_0 in 1:N loop
      rooms_room_Cap[_Index_0]:=550+rand(100);
      rooms_room_Res[_Index_0]:=1.8+rand(0.4);
      rooms_ac_Pot[_Index_0]:=13+rand(0.2);
    end for;
equation
  Power = sum(rooms_ac_pow_out_power);
  for _Index_0 in 1:N loop
        connect(rooms_ac_th_in[_Index_0] , rooms_room_th_out[
            _Index_0]);
        connect(rooms_ac_pow_out[_Index_0] , rooms_room_ac_pow[
            _Index_0]);
```

```
31      rooms_room_th_out_temperature[_Index_0] = rooms_room_temp[
            _Index_0];
32      der(rooms_room_temp[_Index_0])*rooms_room_Cap[_Index_0] =
            rooms_room_THA[_Index_0]/rooms_room_Res[_Index_0]-
            rooms_room_ac_pow_power[_Index_0]-rooms_room_temp[
            _Index_0]/rooms_room_Res[_Index_0]+rooms_room_noise[
            _Index_0]/rooms_room_Res[_Index_0];
33      rooms_ac_pow_out_power[_Index_0] = rooms_ac_on[_Index_0]*
            rooms_ac_Pot[_Index_0];
34    end for;
35  algorithm
36      for _Index_0 in 1:N loop
37        when sample(0,1) then
38          rooms_room_noise[_Index_0]:=rand(2)-1;
39        end when;
40        when rooms_ac_th_in_temperature[_Index_0]-rooms_ac_TRef[
              _Index_0]+rooms_ac_on[_Index_0]-0.5>0 then
41          rooms_ac_on[_Index_0]:=1;
42        elsewhen rooms_ac_th_in_temperature[_Index_0]-
              rooms_ac_TRef[_Index_0]+rooms_ac_on[_Index_0]-0.5<0
              then
43          rooms_ac_on[_Index_0]:=0;
44        end when;
45        when time>1000 then
46          rooms_ac_TRef[_Index_0]:=20.5;
47        end when;
48        when time>2000 then
49          rooms_ac_TRef[_Index_0]:=20;
50        end when;
51      end for;
52  end Airconds;
```

Modelo A.1: Aplanado del modelo 3.20

A.2. Modelo *trline* después de la primer etapa de aplanado

```
1  model trline
2    parameter Real pulsevoltage1_V(start=1);
3    parameter Real pulsevoltage1_width(min=Modelica.Constants.
         small,max=100,start=50);
4    parameter Real pulsevoltage1_period(min=Modelica.Constants.
         small,start=1);
5    parameter Real pulsevoltage1_offset=0;
6    parameter Real pulsevoltage1_startTime=0;
7    parameter Real pulsevoltage1_signalSource_amplitude=
         pulsevoltage1_V;
8    parameter Real pulsevoltage1_signalSource_width(min=Modelica.
         Constants.small,max=100)=pulsevoltage1_signalSource_width;
```

```
9   parameter Real pulsevoltage1_signalSource_period(min=Modelica.
        Constants.small,start=1)=pulsevoltage1_signalSource_period;
10  parameter Integer pulsevoltage1_signalSource_nperiod=(-1);
11  parameter Real pulsevoltage1_signalSource_offset=0;
12  parameter Real pulsevoltage1_signalSource_startTime=0;
13  Real pulsevoltage1_signalSource_T_width=
        pulsevoltage1_signalSource_period*
        pulsevoltage1_signalSource_width/100;
14  Real pulsevoltage1_signalSource_T_start;
15  Integer pulsevoltage1_signalSource_count;
16  output Real pulsevoltage1_signalSource_y;
17  Real pulsevoltage1_v;
18  Real pulsevoltage1_i;
19  Real pulsevoltage1_p_v;
20  Real pulsevoltage1_p_i;
21  PositivePin pulsevoltage1_p;
22  Real pulsevoltage1_n_v;
23  Real pulsevoltage1_n_i;
24  NegativePin pulsevoltage1_n;
25  parameter Real resistor1_R(start=1);
26  parameter Real resistor1_T_ref=300.15;
27  parameter Real resistor1_alpha=0;
28  Real resistor1_R_actual;
29  Real resistor1_v;
30  Real resistor1_i;
31  Real resistor1_p_v;
32  Real resistor1_p_i;
33  PositivePin resistor1_p;
34  Real resistor1_n_v;
35  Real resistor1_n_i;
36  NegativePin resistor1_n;
37  parameter Boolean resistor1_useHeatPort= false ;
38  parameter Real resistor1_T=resistor1_T_ref;
39  Real resistor1_LossPower;
40  Real resistor1_T_heatPort;
41  Real ground1_p_v;
42  Real ground1_p_i;
43  Interfaces_Pin ground1_p;
44  constant Integer N=10;
45  Real line_segment1_ground1_p_v[N];
46  Real line_segment1_ground1_p_i[N];
47  Interfaces_Pin line_segment1_ground1_p[N];
48  parameter Real line_segment1_resistor1_R[N](each start=1);
49  parameter Real line_segment1_resistor1_T_ref[N]=fill(300.15,N)
        ;
50  parameter Real line_segment1_resistor1_alpha[N]=fill(0,N);
51  Real line_segment1_resistor1_R_actual[N];
52  Real line_segment1_resistor1_v[N];
53  Real line_segment1_resistor1_i[N];
54  Real line_segment1_resistor1_p_v[N];
55  Real line_segment1_resistor1_p_i[N];
56  PositivePin line_segment1_resistor1_p[N];
57  Real line_segment1_resistor1_n_v[N];
58  Real line_segment1_resistor1_n_i[N];
```

```
59    NegativePin line_segment1_resistor1_n[N];
60    parameter Boolean line_segment1_resistor1_useHeatPort[N]=fill(
          false ,N);
61    parameter Real line_segment1_resistor1_T[N]=fill(
          line_segment1_resistor1_T_ref ,N);
62    Real line_segment1_resistor1_LossPower[N];
63    Real line_segment1_resistor1_T_heatPort[N];
64    parameter Real line_segment1_inductor1_L[N](each start=1);
65    Real line_segment1_inductor1_v[N];
66    Real line_segment1_inductor1_i[N](each start=0);
67    Real line_segment1_inductor1_p_v[N];
68    Real line_segment1_inductor1_p_i[N];
69    PositivePin line_segment1_inductor1_p[N];
70    Real line_segment1_inductor1_n_v[N];
71    Real line_segment1_inductor1_n_i[N];
72    NegativePin line_segment1_inductor1_n[N];
73    parameter Real line_segment1_capacitor1_C[N](each start=1);
74    Real line_segment1_capacitor1_v[N](each start=0);
75    Real line_segment1_capacitor1_i[N];
76    Real line_segment1_capacitor1_p_v[N];
77    Real line_segment1_capacitor1_p_i[N];
78    PositivePin line_segment1_capacitor1_p[N];
79    Real line_segment1_capacitor1_n_v[N];
80    Real line_segment1_capacitor1_n_i[N];
81    NegativePin line_segment1_capacitor1_n[N];
82    Real line_segment1_pin1_v[N];
83    Real line_segment1_pin1_i[N];
84    Modelica_Electrical_Analog_Interfaces_Pin line_segment1_pin1[N
          ];
85    Real line_segment1_pin0_v[N];
86    Real line_segment1_pin0_i[N];
87    Modelica_Electrical_Analog_Interfaces_Pin line_segment1_pin0[N
          ];
88  initial algorithm
89      pulsevoltage1_signalSource_count:=integer((time-
            pulsevoltage1_signalSource_startTime)/
            pulsevoltage1_signalSource_period);
90      pulsevoltage1_signalSource_T_start:=
            pulsevoltage1_signalSource_startTime+
            pulsevoltage1_signalSource_count*
            pulsevoltage1_signalSource_period;
91  equation
92      connect(pulsevoltage1_n , resistor1_n);
93    for i in 1:N-1 loop
94          connect(line_segment1_pin1[i] , line_segment1_pin0[i+1])
                ;
95    end for;
96      connect(line_segment1_pin0[1] , pulsevoltage1_p);
97      connect(line_segment1_pin1[N] , resistor1_p);
98    pulsevoltage1_v = pulsevoltage1_signalSource_y;
99    pulsevoltage1_v = pulsevoltage1_p_v-pulsevoltage1_n_v;
100   0 = pulsevoltage1_p_i+pulsevoltage1_n_i;
101   pulsevoltage1_i = pulsevoltage1_p_i;
```

```
102   when integer((time-pulsevoltage1_signalSource_startTime)/
          pulsevoltage1_signalSource_period)>pre(
          pulsevoltage1_signalSource_count) then
103     pulsevoltage1_signalSource_count = pre(
            pulsevoltage1_signalSource_count)+1;
104     pulsevoltage1_signalSource_T_start = time;
105   end when;
106   pulsevoltage1_signalSource_y =
          pulsevoltage1_signalSource_offset+(if time<
          pulsevoltage1_signalSource_startTime or
          pulsevoltage1_signalSource_nperiod==0 or
          pulsevoltage1_signalSource_nperiod>0 and
          pulsevoltage1_signalSource_count>=
          pulsevoltage1_signalSource_nperiod then 0 else if time<
          pulsevoltage1_signalSource_T_start+
          pulsevoltage1_signalSource_T_width then
          pulsevoltage1_signalSource_amplitude else 0);
107   assert(1+resistor1_alpha*(resistor1_T_heatPort-resistor1_T_ref
          )>=1e-15,"Temperature outside scope of model!");
108   resistor1_R_actual = resistor1_R*(1+resistor1_alpha*(
          resistor1_T_heatPort-resistor1_T_ref));
109   resistor1_v = resistor1_R_actual*resistor1_i;
110   resistor1_LossPower = resistor1_v*resistor1_i;
111   resistor1_v = resistor1_p_v-resistor1_n_v;
112   0 = resistor1_p_i+resistor1_n_i;
113   resistor1_i = resistor1_p_i;
114   ground1_p_v = 0;
115   for _Index_0 in 1:N loop
116         connect((-line_segment1_pin0[_Index_0]) ,
                line_segment1_resistor1_p[_Index_0]);
117         connect((-line_segment1_pin1[_Index_0]) ,
                line_segment1_capacitor1_p[_Index_0]);
118         connect(line_segment1_capacitor1_n[_Index_0] ,
                line_segment1_ground1_p[_Index_0]);
119         connect(line_segment1_inductor1_n[_Index_0] ,
                line_segment1_capacitor1_p[_Index_0]);
120         connect(line_segment1_resistor1_n[_Index_0] ,
                line_segment1_inductor1_p[_Index_0]);
121
122     line_segment1_ground1_p_v[_Index_0] = 0;
123     assert(1+line_segment1_resistor1_alpha[_Index_0]*(
            line_segment1_resistor1_T_heatPort[_Index_0]-
            line_segment1_resistor1_T_ref[_Index_0])>=1e-15,"
            Temperature outside scope of model!");
124     line_segment1_resistor1_R_actual[_Index_0] =
            line_segment1_resistor1_R[_Index_0]*(1+
            line_segment1_resistor1_alpha[_Index_0]*(
            line_segment1_resistor1_T_heatPort[_Index_0]-
            line_segment1_resistor1_T_ref[_Index_0]));
125     line_segment1_resistor1_v[_Index_0] =
            line_segment1_resistor1_R_actual[_Index_0]*
            line_segment1_resistor1_i[_Index_0];
126     line_segment1_resistor1_LossPower[_Index_0] =
            line_segment1_resistor1_v[_Index_0]*
```

```
        line_segment1_resistor1_i[_Index_0];
127     line_segment1_resistor1_v[_Index_0] =
        line_segment1_resistor1_p_v[_Index_0]-
        line_segment1_resistor1_n_v[_Index_0];
128     0 = line_segment1_resistor1_p_i[_Index_0]+
        line_segment1_resistor1_n_i[_Index_0];
129     line_segment1_resistor1_i[_Index_0] =
        line_segment1_resistor1_p_i[_Index_0];
130     if not line_segment1_resistor1_useHeatPort[_Index_0] then
131       line_segment1_resistor1_T_heatPort[_Index_0] =
            line_segment1_resistor1_T[_Index_0];
132     end if;
133     line_segment1_inductor1_L[_Index_0]*der(
        line_segment1_inductor1_i[_Index_0]) =
        line_segment1_inductor1_v[_Index_0];
134     line_segment1_inductor1_v[_Index_0] =
        line_segment1_inductor1_p_v[_Index_0]-
        line_segment1_inductor1_n_v[_Index_0];
135     0 = line_segment1_inductor1_p_i[_Index_0]+
        line_segment1_inductor1_n_i[_Index_0];
136     line_segment1_inductor1_i[_Index_0] =
        line_segment1_inductor1_p_i[_Index_0];
137     line_segment1_capacitor1_i[_Index_0] =
        line_segment1_capacitor1_C[_Index_0]*der(
        line_segment1_capacitor1_v[_Index_0]);
138     line_segment1_capacitor1_v[_Index_0] =
        line_segment1_capacitor1_p_v[_Index_0]-
        line_segment1_capacitor1_n_v[_Index_0];
139     0 = line_segment1_capacitor1_p_i[_Index_0]+
        line_segment1_capacitor1_n_i[_Index_0];
140     line_segment1_capacitor1_i[_Index_0] =
        line_segment1_capacitor1_p_i[_Index_0];
141   end for;
142   resistor1_T_heatPort = resistor1_T;
143 end trline;
```

Modelo A.2: Modelo 4.13 aplanado

A.3. Modelo *trline* completamente aplanado

```
1 model trline
2   parameter Real pulsevoltage1_V(start=1);
3   parameter Real pulsevoltage1_width(min=Modelica.Constants.
        small,max=100,start=50);
4   parameter Real pulsevoltage1_period(min=Modelica.Constants.
        small,start=1);
5   parameter Real pulsevoltage1_offset=0;
6   parameter Real pulsevoltage1_startTime=0;
7   parameter Real pulsevoltage1_signalSource_amplitude=
        pulsevoltage1_V;
```

```
8   parameter Real pulsevoltage1_signalSource_width(min=Modelica.
        Constants.small,max=100)=pulsevoltage1_signalSource_width;
9   parameter Real pulsevoltage1_signalSource_period(min=Modelica.
        Constants.small,start=1)=pulsevoltage1_signalSource_period;
10  parameter Integer pulsevoltage1_signalSource_nperiod=(-1);
11  parameter Real pulsevoltage1_signalSource_offset=0;
12  parameter Real pulsevoltage1_signalSource_startTime=0;
13  Real pulsevoltage1_signalSource_T_width=
        pulsevoltage1_signalSource_period*
        pulsevoltage1_signalSource_width/100;
14  Real pulsevoltage1_signalSource_T_start;
15  Integer pulsevoltage1_signalSource_count;
16  output Real pulsevoltage1_signalSource_y;
17  Real pulsevoltage1_v;
18  Real pulsevoltage1_i;
19  Real pulsevoltage1_p_v;
20  Real pulsevoltage1_p_i;
21  Real pulsevoltage1_n_v;
22  Real pulsevoltage1_n_i;
23  parameter Real resistor1_R(start=1);
24  parameter Real resistor1_T_ref=300.15;
25  parameter Real resistor1_alpha=0;
26  Real resistor1_R_actual;
27  Real resistor1_v;
28  Real resistor1_i;
29  Real resistor1_p_v;
30  Real resistor1_p_i;
31  Real resistor1_n_v;
32  Real resistor1_n_i;
33  parameter Boolean resistor1_useHeatPort= false ;
34  parameter Real resistor1_T=resistor1_T_ref;
35  Real resistor1_LossPower;
36  Real resistor1_T_heatPort;
37  Real ground1_p_v;
38  Real ground1_p_i;
39  constant Integer N=10;
40  Real line_segment1_ground1_p_v[N];
41  Real line_segment1_ground1_p_i[N];
42  parameter Real line_segment1_resistor1_R[N](each start=1);
43  parameter Real line_segment1_resistor1_T_ref[N]=fill(300.15,N)
        ;
44  parameter Real line_segment1_resistor1_alpha[N]=fill(0,N);
45  Real line_segment1_resistor1_R_actual[N];
46  Real line_segment1_resistor1_v[N];
47  Real line_segment1_resistor1_i[N];
48  Real line_segment1_resistor1_p_v[N];
49  Real line_segment1_resistor1_p_i[N];
50  Real line_segment1_resistor1_n_v[N];
51  Real line_segment1_resistor1_n_i[N];
52  parameter Boolean line_segment1_resistor1_useHeatPort[N]=fill(
        false ,N);
53  parameter Real line_segment1_resistor1_T[N]=fill(
        line_segment1_resistor1_T_ref,N);
54  Real line_segment1_resistor1_LossPower[N];
```

```
55   Real line_segment1_resistor1_T_heatPort[N];
56   parameter Real line_segment1_inductor1_L[N](each start=1);
57   Real line_segment1_inductor1_v[N];
58   Real line_segment1_inductor1_i[N](each start=0);
59   Real line_segment1_inductor1_p_v[N];
60   Real line_segment1_inductor1_p_i[N];
61   Real line_segment1_inductor1_n_v[N];
62   Real line_segment1_inductor1_n_i[N];
63   parameter Real line_segment1_capacitor1_C[N](each start=1);
64   Real line_segment1_capacitor1_v[N](each start=0);
65   Real line_segment1_capacitor1_i[N];
66   Real line_segment1_capacitor1_p_v[N];
67   Real line_segment1_capacitor1_p_i[N];
68   Real line_segment1_capacitor1_n_v[N];
69   Real line_segment1_capacitor1_n_i[N];
70   Real line_segment1_pin1_v[N];
71   Real line_segment1_pin1_i[N];
72   Real line_segment1_pin0_v[N];
73   Real line_segment1_pin0_i[N];
74 initial algorithm
75     pulsevoltage1_signalSource_count:=integer((time-
           pulsevoltage1_signalSource_startTime)/
           pulsevoltage1_signalSource_period);
76     pulsevoltage1_signalSource_T_start:=
           pulsevoltage1_signalSource_startTime+
           pulsevoltage1_signalSource_count*
           pulsevoltage1_signalSource_period;
77 equation
78   pulsevoltage1_v = pulsevoltage1_signalSource_y;
79   pulsevoltage1_v = pulsevoltage1_p_v-pulsevoltage1_n_v;
80   0 = pulsevoltage1_p_i+pulsevoltage1_n_i;
81   pulsevoltage1_i = pulsevoltage1_p_i;
82   when integer((time-pulsevoltage1_signalSource_startTime)/
         pulsevoltage1_signalSource_period)>pre(
         pulsevoltage1_signalSource_count) then
83     pulsevoltage1_signalSource_count = pre(
           pulsevoltage1_signalSource_count)+1;
84     pulsevoltage1_signalSource_T_start = time;
85   end when;
86   pulsevoltage1_signalSource_y =
         pulsevoltage1_signalSource_offset+(if time<
         pulsevoltage1_signalSource_startTime or
         pulsevoltage1_signalSource_nperiod==0 or
         pulsevoltage1_signalSource_nperiod>0 and
         pulsevoltage1_signalSource_count>=
         pulsevoltage1_signalSource_nperiod then 0 else if time<
         pulsevoltage1_signalSource_T_start+
         pulsevoltage1_signalSource_T_width then
         pulsevoltage1_signalSource_amplitude else 0);
87   assert(1+resistor1_alpha*(resistor1_T_heatPort-resistor1_T_ref
         )>=1e-15,"Temperature outside scope of model!");
88   resistor1_R_actual = resistor1_R*(1+resistor1_alpha*(
         resistor1_T_heatPort-resistor1_T_ref));
89   resistor1_v = resistor1_R_actual*resistor1_i;
```

```
90  resistor1_LossPower = resistor1_v*resistor1_i;
91  resistor1_v = resistor1_p_v-resistor1_n_v;
92  0 = resistor1_p_i+resistor1_n_i;
93  resistor1_i = resistor1_p_i;
94  ground1_p_v = 0;
95  for _Index_0 in 1:N loop
96    line_segment1_ground1_p_v[_Index_0] = 0;
97    assert(1+line_segment1_resistor1_alpha[_Index_0]*(
          line_segment1_resistor1_T_heatPort[_Index_0]-
          line_segment1_resistor1_T_ref[_Index_0])>=1e-15,"
          Temperature outside scope of model!");
98    line_segment1_resistor1_R_actual[_Index_0] =
          line_segment1_resistor1_R[_Index_0]*(1+
          line_segment1_resistor1_alpha[_Index_0]*(
          line_segment1_resistor1_T_heatPort[_Index_0]-
          line_segment1_resistor1_T_ref[_Index_0]));
99    line_segment1_resistor1_v[_Index_0] =
          line_segment1_resistor1_R_actual[_Index_0]*
          line_segment1_resistor1_i[_Index_0];
100   line_segment1_resistor1_LossPower[_Index_0] =
          line_segment1_resistor1_v[_Index_0]*
          line_segment1_resistor1_i[_Index_0];
101   line_segment1_resistor1_v[_Index_0] =
          line_segment1_resistor1_p_v[_Index_0]-
          line_segment1_resistor1_n_v[_Index_0];
102   0 = line_segment1_resistor1_p_i[_Index_0]+
          line_segment1_resistor1_n_i[_Index_0];
103   line_segment1_resistor1_i[_Index_0] =
          line_segment1_resistor1_p_i[_Index_0];
104   if not line_segment1_resistor1_useHeatPort[_Index_0] then
105     line_segment1_resistor1_T_heatPort[_Index_0] =
            line_segment1_resistor1_T[_Index_0];
106   end if;
107   line_segment1_inductor1_L[_Index_0]*der(
          line_segment1_inductor1_i[_Index_0]) =
          line_segment1_inductor1_v[_Index_0];
108   line_segment1_inductor1_v[_Index_0] =
          line_segment1_inductor1_p_v[_Index_0]-
          line_segment1_inductor1_n_v[_Index_0];
109   0 = line_segment1_inductor1_p_i[_Index_0]+
          line_segment1_inductor1_n_i[_Index_0];
110   line_segment1_inductor1_i[_Index_0] =
          line_segment1_inductor1_p_i[_Index_0];
111   line_segment1_capacitor1_i[_Index_0] =
          line_segment1_capacitor1_C[_Index_0]*der(
          line_segment1_capacitor1_v[_Index_0]);
112   line_segment1_capacitor1_v[_Index_0] =
          line_segment1_capacitor1_p_v[_Index_0]-
          line_segment1_capacitor1_n_v[_Index_0];
113   0 = line_segment1_capacitor1_p_i[_Index_0]+
          line_segment1_capacitor1_n_i[_Index_0];
114   line_segment1_capacitor1_i[_Index_0] =
          line_segment1_capacitor1_p_i[_Index_0];
115 end for;
```

```
116   resistor1_T_heatPort = resistor1_T;
117   resistor1_n_v = pulsevoltage1_n_v;
118   resistor1_n_i+pulsevoltage1_n_i = 0;
119   for i in 1:9 loop
120     line_segment1_pin0_v[i+1] = line_segment1_pin1_v[i];
121   end for;
122   for i in 1:9 loop
123     line_segment1_pin0_i[i+1]+line_segment1_pin1_i[i] = 0;
124   end for;
125   resistor1_p_v = line_segment1_pin1_v[10];
126   resistor1_p_i+line_segment1_pin1_i[10] = 0;
127   pulsevoltage1_p_v = line_segment1_pin0_v[1];
128   pulsevoltage1_p_i+line_segment1_pin0_i[1] = 0;
129   for i in 1:10 loop
130     line_segment1_resistor1_p_v[i] = line_segment1_pin0_v[i];
131   end for;
132   for i in 1:10 loop
133     line_segment1_resistor1_p_i[i]+(-line_segment1_pin0_i[i]) =
          0;
134   end for;
135   for i in 1:10 loop
136     line_segment1_inductor1_n_v[i] =
          line_segment1_capacitor1_p_v[i];
137     line_segment1_inductor1_n_v[i] = line_segment1_pin1_v[i];
138   end for;
139   for i in 1:10 loop
140     line_segment1_inductor1_n_i[i]+line_segment1_capacitor1_p_i[
          i]+(-line_segment1_pin1_i[i]) = 0;
141   end for;
142   for i in 1:10 loop
143     line_segment1_ground1_p_v[i] = line_segment1_capacitor1_n_v[
          i];
144   end for;
145   for i in 1:10 loop
146     line_segment1_ground1_p_i[i]+line_segment1_capacitor1_n_i[i]
          = 0;
147   end for;
148   for i in 1:10 loop
149     line_segment1_inductor1_p_v[i] = line_segment1_resistor1_n_v
          [i];
150   end for;
151   for i in 1:10 loop
152     line_segment1_inductor1_p_i[i]+line_segment1_resistor1_n_i[i
          ] = 0;
153   end for;
154   ground1_p_i = 0;
155 end trline;
```

Modelo A.3: Modelo 4.13 totalmente aplanado

A.4. Modelo del *circuito Chua* aplanado por OpenModelica

```
class ChuaCircuit
  Real L.v(quantity = "ElectricPotential", unit = "V");
  Real L.i(quantity = "ElectricCurrent", unit = "A", start =
      0.0, fixed = true);
  Real L.p.v(quantity = "ElectricPotential", unit = "V");
  Real L.p.i(quantity = "ElectricCurrent", unit = "A");
  Real L.n.v(quantity = "ElectricPotential", unit = "V");
  Real L.n.i(quantity = "ElectricCurrent", unit = "A");
  parameter Real L.L(quantity = "Inductance", unit = "H", start
      = 1.0) = 18.0;
  Real Ro.v(quantity = "ElectricPotential", unit = "V");
  Real Ro.i(quantity = "ElectricCurrent", unit = "A");
  Real Ro.p.v(quantity = "ElectricPotential", unit = "V");
  Real Ro.p.i(quantity = "ElectricCurrent", unit = "A");
  Real Ro.n.v(quantity = "ElectricPotential", unit = "V");
  Real Ro.n.i(quantity = "ElectricCurrent", unit = "A");
  parameter Boolean Ro.useHeatPort = false;
  Real Ro.LossPower(quantity = "Power", unit = "W");
  Real Ro.T_heatPort(quantity = "ThermodynamicTemperature", unit
       = "K", displayUnit = "degC", min = 0.0, start = 288.15,
      nominal = 300.0);
  parameter Real Ro.R(quantity = "Resistance", unit = "Ohm",
      start = 1.0) = 0.0125;
  parameter Real Ro.T_ref(quantity = "ThermodynamicTemperature",
       unit = "K", displayUnit = "degC", min = 0.0, start =
      288.15, nominal = 300.0) = 300.15 ;
  parameter Real Ro.alpha(quantity = "
      LinearTemperatureCoefficient", unit = "1/K") = 0.0;
  Real Ro.R_actual(quantity = "Resistance", unit = "Ohm");
  parameter Real Ro.T(quantity = "ThermodynamicTemperature",
      unit = "K", displayUnit = "degC", min = 0.0, start =
      288.15, nominal = 300.0) = Ro.T_ref;
  Real G.v(quantity = "ElectricPotential", unit = "V");
  Real G.i(quantity = "ElectricCurrent", unit = "A");
  Real G.p.v(quantity = "ElectricPotential", unit = "V");
  Real G.p.i(quantity = "ElectricCurrent", unit = "A");
  Real G.n.v(quantity = "ElectricPotential", unit = "V");
  Real G.n.i(quantity = "ElectricCurrent", unit = "A");
  parameter Boolean G.useHeatPort = false;
  Real G.LossPower(quantity = "Power", unit = "W");
  Real G.T_heatPort(quantity = "ThermodynamicTemperature", unit
      = "K", displayUnit = "degC", min = 0.0, start = 288.15,
      nominal = 300.0);
  parameter Real G.G(quantity = "Conductance", unit = "S", start
       = 1.0) = 0.5649999999999999 ;
  parameter Real G.T_ref(quantity = "ThermodynamicTemperature",
      unit = "K", displayUnit = "degC", min = 0.0, start =
      288.15, nominal = 300.0) = 300.15;
```

```
34   parameter Real G.alpha(quantity = "
         LinearTemperatureCoefficient", unit = "1/K") = 0.0;
35   Real G.G_actual(quantity = "Conductance", unit = "S");
36   parameter Real G.T(quantity = "ThermodynamicTemperature", unit
         = "K", displayUnit = "degC", min = 0.0, start = 288.15,
         nominal = 300.0) = G.T_ref;
37   Real C1.v(quantity = "ElectricPotential", unit = "V", start =
         4.0, fixed = true);
38   Real C1.i(quantity = "ElectricCurrent", unit = "A");
39   Real C1.p.v(quantity = "ElectricPotential", unit = "V");
40   Real C1.p.i(quantity = "ElectricCurrent", unit = "A");
41   Real C1.n.v(quantity = "ElectricPotential", unit = "V");
42   Real C1.n.i(quantity = "ElectricCurrent", unit = "A");
43   parameter Real C1.C(quantity = "Capacitance", unit = "F", min
         = 0.0, start = 1.0) = 10.0;
44   Real C2.v(quantity = "ElectricPotential", unit = "V", start =
         0.0, fixed = true);
45   Real C2.i(quantity = "ElectricCurrent", unit = "A");
46   Real C2.p.v(quantity = "ElectricPotential", unit = "V");
47   Real C2.p.i(quantity = "ElectricCurrent", unit = "A");
48   Real C2.n.v(quantity = "ElectricPotential", unit = "V");
49   Real C2.n.i(quantity = "ElectricCurrent", unit = "A") ;
50   parameter Real C2.C(quantity = "Capacitance", unit = "F", min
         = 0.0, start = 1.0) = 100.0;
51   Real Nr.v(quantity = "ElectricPotential", unit = "V");
52   Real Nr.i(quantity = "ElectricCurrent", unit = "A") ;
53   Real Nr.p.v(quantity = "ElectricPotential", unit = "V");
54   Real Nr.p.i(quantity = "ElectricCurrent", unit = "A");
55   Real Nr.n.v(quantity = "ElectricPotential", unit = "V");
56   Real Nr.n.i(quantity = "ElectricCurrent", unit = "A");
57   parameter Real Nr.Ga(quantity = "Conductance", unit = "S", min
         = -1.0) = -0.757576 ;
58   parameter Real Nr.Gb(quantity = "Conductance", unit = "S", min
         = -1.0) = -0.409091 ;
59   parameter Real Nr.Ve(quantity = "ElectricPotential", unit = "V
         ") = 1.0 ;
60   Real Gnd.p.v(quantity = "ElectricPotential", unit = "V");
61   Real Gnd.p.i(quantity = "ElectricCurrent", unit = "A") ;
62 equation
63   L.L * der(L.i) = L.v;
64   L.v = L.p.v - L.n.v;
65   0.0 = L.p.i + L.n.i;
66   L.i = L.p.i;
67   assert(1.0 + Ro.alpha * (Ro.T_heatPort - Ro.T_ref) >= 1e-15, "
         Temperature outside scope of model!");
68   Ro.R_actual = Ro.R * (1.0 + Ro.alpha * (Ro.T_heatPort - Ro.
         T_ref));
69   Ro.v = Ro.R_actual * Ro.i;
70   Ro.LossPower = Ro.v * Ro.i;
71   Ro.v = Ro.p.v - Ro.n.v;
72   0.0 = Ro.p.i + Ro.n.i;
73   Ro.i = Ro.p.i;
74   Ro.T_heatPort = Ro.T;
```

```
75    assert(1.0 + G.alpha * (G.T_heatPort - G.T_ref) >= 1e-15, "
          Temperature outside scope of model!");
76    G.G_actual = G.G / (1.0 + G.alpha * (G.T_heatPort - G.T_ref));
77    G.i = G.G_actual * G.v;
78    G.LossPower = G.v * G.i;
79    G.v = G.p.v - G.n.v;
80    0.0 = G.p.i + G.n.i;
81    G.i = G.p.i;
82    G.T_heatPort = G.T;
83    C1.i = C1.C * der(C1.v);
84    C1.v = C1.p.v - C1.n.v;
85    0.0 = C1.p.i + C1.n.i;
86    C1.i = C1.p.i;
87    C2.i = C2.C * der(C2.v);
88    C2.v = C2.p.v - C2.n.v;
89    0.0 = C2.p.i + C2.n.i;
90    C2.i = C2.p.i;
91    Nr.i = if Nr.v < (-Nr.Ve) then Nr.Gb * (Nr.v + Nr.Ve) - Nr.Ga
          * Nr.Ve else if Nr.v > Nr.Ve then Nr.Gb * (Nr.v - Nr.Ve) +
          Nr.Ga * Nr.Ve else Nr.Ga * Nr.v;
92    Nr.v = Nr.p.v - Nr.n.v;
93    0.0 = Nr.p.i + Nr.n.i;
94    Nr.i = Nr.p.i;
95    Gnd.p.v = 0.0;
96    L.p.i + C2.p.i + G.p.i = 0.0;
97    L.n.i + Ro.p.i = 0.0;
98    Ro.n.i + Gnd.p.i + Nr.n.i + C2.n.i + C1.n.i = 0.0;
99    Nr.p.i + G.n.i + C1.p.i = 0.0;
100   L.n.v = Ro.p.v;
101   C2.p.v = G.p.v;
102   C2.p.v = L.p.v;
103   C1.p.v = G.n.v;
104   C1.p.v = Nr.p.v;
105   C1.n.v = C2.n.v;
106   C1.n.v = Gnd.p.v;
107   C1.n.v = Nr.n.v;
108   C1.n.v = Ro.n.v;
109 end ChuaCircuit;
```

Modelo A.4: Modelo del *circuito Chua* aplanado por OpenModelica

A.5. Modelo del *cicuito Chua* aplanado por ModelicaCC

```
1 model ChuaCircuit
2   parameter Real L_L(start=1)=18;
3   Real L_v;
4   Real L_i(start=0,fixed= true );
5   Real L_p_v;
6   Real L_p_i;
```

```
7   Real L_n_v;
8   Real L_n_i;
9   parameter Real Ro_R(start=1)=0.0125;
10  parameter Real Ro_T_ref=300.15;
11  parameter Real Ro_alpha=0;
12  Real Ro_R_actual;
13  Real Ro_v;
14  Real Ro_i;
15  Real Ro_p_v;
16  Real Ro_p_i;
17  Real Ro_n_v;
18  Real Ro_n_i;
19  parameter Boolean Ro_useHeatPort= false ;
20  parameter Real Ro_T=Ro_T_ref;
21  Real Ro_LossPower;
22  Real Ro_T_heatPort;
23  parameter Real G_G(start=1)=0.565;
24  parameter Real G_T_ref=300.15;
25  parameter Real G_alpha=0;
26  Real G_G_actual;
27  Real G_v;
28  Real G_i;
29  Real G_p_v;
30  Real G_p_i;
31  Real G_n_v;
32  Real G_n_i;
33  parameter Boolean G_useHeatPort= false ;
34  parameter Real G_T=G_T_ref;
35  Real G_LossPower;
36  Real G_T_heatPort;
37  parameter Real C1_C(start=1)=10;
38  Real C1_v(start=4,fixed= true );
39  Real C1_i;
40  Real C1_p_v;
41  Real C1_p_i;
42  Real C1_n_v;
43  Real C1_n_i;
44  parameter Real C2_C(start=1)=100;
45  Real C2_v(start=0,fixed= true );
46  Real C2_i;
47  Real C2_p_v;
48  Real C2_p_i;
49  Real C2_n_v;
50  Real C2_n_i;
51  parameter Real Nr_Ga(min=(-1))=(-0.757576);
52  parameter Real Nr_Gb(min=(-1))=(-0.409091);
53  parameter Real Nr_Ve=1;
54  Real Nr_v;
55  Real Nr_i;
56  Real Nr_p_v;
57  Real Nr_p_i;
58  Real Nr_n_v;
59  Real Nr_n_i;
60  Real Gnd_p_v;
```

```
61    Real Gnd_p_i;
62  equation
63    L_L*der(L_i) = L_v;
64    L_v = L_p_v-L_n_v;
65    0 = L_p_i+L_n_i;
66    L_i = L_p_i;
67    assert(1+Ro_alpha*(Ro_T_heatPort-Ro_T_ref)>=1e-15,"Temperature
          outside scope of model!");
68    Ro_R_actual = Ro_R*(1+Ro_alpha*(Ro_T_heatPort-Ro_T_ref));
69    Ro_v = Ro_R_actual*Ro_i;
70    Ro_LossPower = Ro_v*Ro_i;
71    Ro_v = Ro_p_v-Ro_n_v;
72    0 = Ro_p_i+Ro_n_i;
73    Ro_i = Ro_p_i;
74    if  not Ro_useHeatPort then
75      Ro_T_heatPort = Ro_T;
76    end if;
77    assert(1+G_alpha*(G_T_heatPort-G_T_ref)>=1e-15,"Temperature
          outside scope of model!");
78    G_G_actual = G_G/(1+G_alpha*(G_T_heatPort-G_T_ref));
79    G_i = G_G_actual*G_v;
80    G_LossPower = G_v*G_i;
81    G_v = G_p_v-G_n_v;
82    0 = G_p_i+G_n_i;
83    G_i = G_p_i;
84    if  not G_useHeatPort then
85      G_T_heatPort = G_T;
86    end if;
87    C1_i = C1_C*der(C1_v);
88    C1_v = C1_p_v-C1_n_v;
89    0 = C1_p_i+C1_n_i;
90    C1_i = C1_p_i;
91    C2_i = C2_C*der(C2_v);
92    C2_v = C2_p_v-C2_n_v;
93    0 = C2_p_i+C2_n_i;
94    C2_i = C2_p_i;
95    Nr_i = if Nr_v<((-Nr_Ve)) then Nr_Gb*(Nr_v+Nr_Ve)-Nr_Ga*Nr_Ve
          else if Nr_v>Nr_Ve then Nr_Gb*(Nr_v-Nr_Ve)+Nr_Ga*Nr_Ve else
           Nr_Ga*Nr_v;
96    Nr_v = Nr_p_v-Nr_n_v;
97    0 = Nr_p_i+Nr_n_i;
98    Nr_i = Nr_p_i;
99    Gnd_p_v = 0;
100   Ro_p_v = L_n_v;
101   Ro_p_i+L_n_i = 0;
102   L_p_v = G_p_v;
103   L_p_v = C2_p_v;
104   L_p_i+G_p_i+C2_p_i = 0;
105   Nr_p_v = C1_p_v;
106   Nr_p_v = G_n_v;
107   Nr_p_i+C1_p_i+G_n_i = 0;
108   Nr_n_v = C1_n_v;
109   Nr_n_v = C2_n_v;
110   Nr_n_v = Gnd_p_v;
```

```
111   Nr_n_v = Ro_n_v;
112   Nr_n_i+C1_n_i+C2_n_i+Gnd_p_i+Ro_n_i = 0;
113 end ChuaCircuit;
```

Modelo A.5: Modelo del *circuito Chua* aplanado por ModelicaCC

A.6. Modelo *Airconds* aplanado con OpenModelica para N=3

```
1  class Airconds
2    constant Integer N = 3;
3    parameter Real rooms[1].room.Cap(fixed = false);
4    parameter Real rooms[1].room.Res(fixed = false);
5    parameter Real rooms[1].room.THA = 32.0;
6    Real rooms[1].room.th_out.temperature;
7    Real rooms[1].room.ac_pow.power;
8    Real rooms[1].room.temp(start = 20.0);
9    discrete Real rooms[1].room.noise;
10   parameter Real rooms[1].ac.Pot(fixed = false);
11   discrete Real rooms[1].ac.TRef(start = 20.0);
12   Real rooms[1].ac.th_in.temperature;
13   Real rooms[1].ac.pow_out.power;
14   discrete Real rooms[1].ac.on;
15   parameter Real rooms[2].room.Cap(fixed = false);
16   parameter Real rooms[2].room.Res(fixed = false);
17   parameter Real rooms[2].room.THA = 32.0;
18   Real rooms[2].room.th_out.temperature;
19   Real rooms[2].room.ac_pow.power;
20   Real rooms[2].room.temp(start = 20.0);
21   discrete Real rooms[2].room.noise;
22   parameter Real rooms[2].ac.Pot(fixed = false);
23   discrete Real rooms[2].ac.TRef(start = 20.0);
24   Real rooms[2].ac.th_in.temperature;
25   Real rooms[2].ac.pow_out.power;
26   discrete Real rooms[2].ac.on;
27   parameter Real rooms[3].room.Cap(fixed = false);
28   parameter Real rooms[3].room.Res(fixed = false);
29   parameter Real rooms[3].room.THA = 32.0;
30   Real rooms[3].room.th_out.temperature;
31   Real rooms[3].room.ac_pow.power;
32   Real rooms[3].room.temp(start = 20.0);
33   discrete Real rooms[3].room.noise;
34   parameter Real rooms[3].ac.Pot(fixed = false);
35   discrete Real rooms[3].ac.TRef(start = 20.0);
36   Real rooms[3].ac.th_in.temperature;
37   Real rooms[3].ac.pow_out.power;
38   discrete Real rooms[3].ac.on;
39   Real Power;
40 initial algorithm
41   rooms[1].room.Cap := 600.0;
```

```
42    rooms[1].room.Res := 2.0;
43  initial algorithm
44    rooms[1].ac.Pot := 13.1;
45  initial algorithm
46    rooms[2].room.Cap := 600.0;
47    rooms[2].room.Res := 2.0;
48  initial algorithm
49    rooms[2].ac.Pot := 13.1;
50  initial algorithm
51    rooms[3].room.Cap := 600.0;
52    rooms[3].room.Res := 2.0;
53  initial algorithm
54    rooms[3].ac.Pot := 13.1;
55  equation
56    rooms[1].room.th_out.temperature = rooms[1].room.temp;
57    der(rooms[1].room.temp) * rooms[1].room.Cap = rooms[1].room.
          THA / rooms[1].room.Res + (rooms[1].room.noise - rooms[1].
          room.temp) / rooms[1].room.Res - rooms[1].room.ac_pow.power
          ;
58    rooms[1].ac.pow_out.power = rooms[1].ac.on * rooms[1].ac.Pot;
59    rooms[2].room.th_out.temperature = rooms[2].room.temp;
60    der(rooms[2].room.temp) * rooms[2].room.Cap = rooms[2].room.
          THA / rooms[2].room.Res + (rooms[2].room.noise - rooms[2].
          room.temp) / rooms[2].room.Res - rooms[2].room.ac_pow.power
          ;
61    rooms[2].ac.pow_out.power = rooms[2].ac.on * rooms[2].ac.Pot;
62    rooms[3].room.th_out.temperature = rooms[3].room.temp;
63    der(rooms[3].room.temp) * rooms[3].room.Cap = rooms[3].room.
          THA / rooms[3].room.Res + (rooms[3].room.noise - rooms[3].
          room.temp) / rooms[3].room.Res - rooms[3].room.ac_pow.power
          ;
64    rooms[3].ac.pow_out.power = rooms[3].ac.on * rooms[3].ac.Pot;
65    Power = rooms[1].ac.pow_out.power + rooms[2].ac.pow_out.power
          + rooms[3].ac.pow_out.power;
66    rooms[3].ac.th_in.temperature = rooms[3].room.th_out.
          temperature;
67    rooms[3].ac.pow_out.power = rooms[3].room.ac_pow.power;
68    rooms[2].ac.th_in.temperature = rooms[2].room.th_out.
          temperature;
69    rooms[2].ac.pow_out.power = rooms[2].room.ac_pow.power;
70    rooms[1].ac.th_in.temperature = rooms[1].room.th_out.
          temperature;
71    rooms[1].ac.pow_out.power = rooms[1].room.ac_pow.power;
72  algorithm
73    when sample(0.0, 1.0) then
74      rooms[1].room.noise := 0.0;
75    end when;
76  algorithm
77    when rooms[1].ac.th_in.temperature + rooms[1].ac.on + -0.5 -
          rooms[1].ac.TRef > 0.0 then
78      rooms[1].ac.on := 1.0;
79    elsewhen rooms[1].ac.th_in.temperature + rooms[1].ac.on + -0.5
           - rooms[1].ac.TRef < 0.0 then
80      rooms[1].ac.on := 0.0;
```

```
81    end when;
82    when time > 1000.0 then
83      rooms[1].ac.TRef := 20.5;
84    end when;
85    when time > 2000.0 then
86      rooms[1].ac.TRef := 20.0;
87    end when;
88  algorithm
89    when sample(0.0, 1.0) then
90      rooms[2].room.noise := 0.0;
91    end when;
92  algorithm
93    when rooms[2].ac.th_in.temperature + rooms[2].ac.on + -0.5 -
          rooms[2].ac.TRef > 0.0 then
94      rooms[2].ac.on := 1.0;
95    elsewhen rooms[2].ac.th_in.temperature + rooms[2].ac.on + -0.5
          - rooms[2].ac.TRef < 0.0 then
96      rooms[2].ac.on := 0.0;
97    end when;
98    when time > 1000.0 then
99      rooms[2].ac.TRef := 20.5;
100   end when;
101   when time > 2000.0 then
102     rooms[2].ac.TRef := 20.0;
103   end when;
104 algorithm
105   when sample(0.0, 1.0) then
106     rooms[3].room.noise := 0.0;
107   end when;
108 algorithm
109   when rooms[3].ac.th_in.temperature + rooms[3].ac.on + -0.5 -
          rooms[3].ac.TRef > 0.0 then
110     rooms[3].ac.on := 1.0;
111   elsewhen rooms[3].ac.th_in.temperature + rooms[3].ac.on + -0.5
          - rooms[3].ac.TRef < 0.0 then
112     rooms[3].ac.on := 0.0;
113   end when;
114   when time > 1000.0 then
115     rooms[3].ac.TRef := 20.5;
116   end when;
117   when time > 2000.0 then
118     rooms[3].ac.TRef := 20.0;
119   end when;
120 end Airconds;
```

Modelo A.6: Aplanado del modelo *Airconds* con OpenModelica Para N=3

A.7. Modelo *Airconds* aplanado con ModelicaCC para N=3

```
model Airconds
  constant Integer N=3;
  parameter Real rooms_room_Cap[N](fixed= false );
  parameter Real rooms_room_Res[N](fixed= false );
  parameter Real rooms_room_THA[N]= fill(32,N);
  Real rooms_room_th_out_temperature[N];
  Real rooms_room_ac_pow_power[N];
  Real rooms_room_temp[N](each start=20);
  discrete Real rooms_room_noise[N];
  parameter Real rooms_ac_Pot[N](fixed= false );
  discrete Real rooms_ac_TRef[N](each start=20);
  Real rooms_ac_th_in_temperature[N];
  Real rooms_ac_pow_out_power[N];
  discrete Real rooms_ac_on[N];
  Real Power;
initial algorithm
    for _Index_0 in 1:N loop
      rooms_room_Cap[_Index_0]:=550+rand(100);
      rooms_room_Res[_Index_0]:=1.8+rand(0.4);
      rooms_ac_Pot[_Index_0]:=13+rand(0.2);
    end for;
equation
  Power = sum(rooms_ac_pow_out_power);
  for _Index_0 in 1:N loop
    room_th_out_temperature[_Index_0] = rooms_room_temp[_Index_0
        ];
    der(rooms_room_temp[_Index_0])*rooms_room_Cap[_Index_0] =
        rooms_room_THA[_Index_0]/rooms_room_Res[_Index_0]-
        room_ac_pow_power[_Index_0]-rooms_room_temp[_Index_0]/
        rooms_room_Res[_Index_0]+rooms_room_noise[_Index_0]/
        rooms_room_Res[_Index_0];
    ac_pow_out_power[_Index_0] = rooms_ac_on[_Index_0]*
        rooms_ac_Pot[_Index_0];
  end for;
  for i in 1:3 loop
    rooms_room_th_out_temperature[i] =
        rooms_ac_th_in_temperature[i];
  end for;
  for i in 1:3 loop
    rooms_room_ac_pow_power[i] = rooms_ac_pow_out_power[i];
  end for;
algorithm
    for _Index_0 in 1:N loop
      when sample(0,1) then
        rooms_room_noise[_Index_0]:=rand(2)-1;
      end when;
      when ac_th_in_temperature[_Index_0]-rooms_ac_TRef[_Index_0
          ]+rooms_ac_on[_Index_0]-0.5>0 then
        rooms_ac_on[_Index_0]:=1;
      elsewhen ac_th_in_temperature[_Index_0]-rooms_ac_TRef[
          _Index_0]+rooms_ac_on[_Index_0]-0.5<0 then
        rooms_ac_on[_Index_0]:=0;
      end when;
      when time[_Index_0]>1000 then
```

```
46          rooms_ac_TRef[_Index_0]:=20.5;
47        end when;
48        when time[_Index_0]>2000 then
49          rooms_ac_TRef[_Index_0]:=20;
50        end when;
51      end for;
52  end Airconds;
```

Modelo A.7: Aplanado del modelo *Airconds* con ModelicaCC para N=3

A.8. Modelo *lcline* aplanado con OpenModelica para N=3

```
1  class lcline
2    Real resistor1.v(quantity = "ElectricPotential", unit = "V") "
       Voltage drop between the two pins (= p.v - n.v)";
3    Real resistor1.i(quantity = "ElectricCurrent", unit = "A") "
       Current flowing from pin p to pin n";
4    Real resistor1.p.v(quantity = "ElectricPotential", unit = "V")
        "Potential at the pin";
5    Real resistor1.p.i(quantity = "ElectricCurrent", unit = "A") "
       Current flowing into the pin";
6    Real resistor1.n.v(quantity = "ElectricPotential", unit = "V")
        "Potential at the pin";
7    Real resistor1.n.i(quantity = "ElectricCurrent", unit = "A") "
       Current flowing into the pin";
8    parameter Boolean resistor1.useHeatPort = false "=true, if
       HeatPort is enabled";
9    Real resistor1.LossPower(quantity = "Power", unit = "W") "Loss
        power leaving component via HeatPort";
10   Real resistor1.T_heatPort(quantity = "ThermodynamicTemperature
       ", unit = "K", displayUnit = "degC", min = 0.0, start =
       288.15, nominal = 300.0) "Temperature of HeatPort";
11   parameter Real resistor1.R(quantity = "Resistance", unit = "
       Ohm", start = 1.0) "Resistance at temperature T_ref";
12   parameter Real resistor1.T_ref(quantity = "
       ThermodynamicTemperature", unit = "K", displayUnit = "degC
       ", min = 0.0, start = 288.15, nominal = 300.0) = 300.15 "
       Reference temperature";
13   parameter Real resistor1.alpha(quantity = "
       LinearTemperatureCoefficient", unit = "1/K") = 0.0 "
       Temperature coefficient of resistance (R_actual = R*(1 +
       alpha*(T_heatPort - T_ref))";
14   Real resistor1.R_actual(quantity = "Resistance", unit = "Ohm")
        "Actual resistance = R*(1 + alpha*(T_heatPort - T_ref))";
15   parameter Real resistor1.T(quantity = "
       ThermodynamicTemperature", unit = "K", displayUnit = "degC
       ", min = 0.0, start = 288.15, nominal = 300.0) = resistor1.
       T_ref "Fixed device temperature if useHeatPort = false";
```

```
16  Real ground1.p.v(quantity = "ElectricPotential", unit = "V") "
        Potential at the pin";
17  Real ground1.p.i(quantity = "ElectricCurrent", unit = "A") "
        Current flowing into the pin";
18  constant Integer N = 3;
19  Real lc[1].pin2.v(quantity = "ElectricPotential", unit = "V")
        "Potential at the pin";
20  Real lc[1].pin2.i(quantity = "ElectricCurrent", unit = "A") "
        Current flowing into the pin";
21  Real lc[1].inductor1.v(quantity = "ElectricPotential", unit =
        "V") "Voltage drop between the two pins (= p.v - n.v)";
22  Real lc[1].inductor1.i(quantity = "ElectricCurrent", unit = "A
        ", start = 0.0) "Current flowing from pin p to pin n";
23  Real lc[1].inductor1.p.v(quantity = "ElectricPotential", unit
        = "V") "Potential at the pin";
24  Real lc[1].inductor1.p.i(quantity = "ElectricCurrent", unit =
        "A") "Current flowing into the pin";
25  Real lc[1].inductor1.n.v(quantity = "ElectricPotential", unit
        = "V") "Potential at the pin";
26  Real lc[1].inductor1.n.i(quantity = "ElectricCurrent", unit =
        "A") "Current flowing into the pin";
27  parameter Real lc[1].inductor1.L(quantity = "Inductance", unit
         = "H", start = 1.0) "Inductance";
28  Real lc[1].pin.v(quantity = "ElectricPotential", unit = "V") "
        Potential at the pin";
29  Real lc[1].pin.i(quantity = "ElectricCurrent", unit = "A") "
        Current flowing into the pin";
30  Real lc[1].pin1.v(quantity = "ElectricPotential", unit = "V")
        "Potential at the pin";
31  Real lc[1].pin1.i(quantity = "ElectricCurrent", unit = "A") "
        Current flowing into the pin";
32  Real lc[1].capacitor1.v(quantity = "ElectricPotential", unit =
         "V", start = 0.0) "Voltage drop between the two pins (= p.
        v - n.v)";
33  Real lc[1].capacitor1.i(quantity = "ElectricCurrent", unit = "
        A") "Current flowing from pin p to pin n";
34  Real lc[1].capacitor1.p.v(quantity = "ElectricPotential", unit
         = "V") "Potential at the pin";
35  Real lc[1].capacitor1.p.i(quantity = "ElectricCurrent", unit =
         "A") "Current flowing into the pin";
36  Real lc[1].capacitor1.n.v(quantity = "ElectricPotential", unit
         = "V") "Potential at the pin";
37  Real lc[1].capacitor1.n.i(quantity = "ElectricCurrent", unit =
         "A") "Current flowing into the pin";
38  parameter Real lc[1].capacitor1.C(quantity = "Capacitance",
        unit = "F", min = 0.0, start = 1.0) "Capacitance";
39  Real lc[2].pin2.v(quantity = "ElectricPotential", unit = "V")
        "Potential at the pin";
40  Real lc[2].pin2.i(quantity = "ElectricCurrent", unit = "A") "
        Current flowing into the pin";
41  Real lc[2].inductor1.v(quantity = "ElectricPotential", unit =
        "V") "Voltage drop between the two pins (= p.v - n.v)";
42  Real lc[2].inductor1.i(quantity = "ElectricCurrent", unit = "A
        ", start = 0.0) "Current flowing from pin p to pin n";
```

```
43  Real lc[2].inductor1.p.v(quantity = "ElectricPotential", unit
        = "V") "Potential at the pin";
44  Real lc[2].inductor1.p.i(quantity = "ElectricCurrent", unit =
        "A") "Current flowing into the pin";
45  Real lc[2].inductor1.n.v(quantity = "ElectricPotential", unit
        = "V") "Potential at the pin";
46  Real lc[2].inductor1.n.i(quantity = "ElectricCurrent", unit =
        "A") "Current flowing into the pin";
47  parameter Real lc[2].inductor1.L(quantity = "Inductance", unit
         = "H", start = 1.0) "Inductance";
48  Real lc[2].pin.v(quantity = "ElectricPotential", unit = "V") "
        Potential at the pin";
49  Real lc[2].pin.i(quantity = "ElectricCurrent", unit = "A") "
        Current flowing into the pin";
50  Real lc[2].pin1.v(quantity = "ElectricPotential", unit = "V")
        "Potential at the pin";
51  Real lc[2].pin1.i(quantity = "ElectricCurrent", unit = "A") "
        Current flowing into the pin";
52  Real lc[2].capacitor1.v(quantity = "ElectricPotential", unit =
         "V", start = 0.0) "Voltage drop between the two pins (= p.
        v - n.v)";
53  Real lc[2].capacitor1.i(quantity = "ElectricCurrent", unit = "
        A") "Current flowing from pin p to pin n";
54  Real lc[2].capacitor1.p.v(quantity = "ElectricPotential", unit
         = "V") "Potential at the pin";
55  Real lc[2].capacitor1.p.i(quantity = "ElectricCurrent", unit =
         "A") "Current flowing into the pin";
56  Real lc[2].capacitor1.n.v(quantity = "ElectricPotential", unit
         = "V") "Potential at the pin";
57  Real lc[2].capacitor1.n.i(quantity = "ElectricCurrent", unit =
         "A") "Current flowing into the pin";
58  parameter Real lc[2].capacitor1.C(quantity = "Capacitance",
        unit = "F", min = 0.0, start = 1.0) "Capacitance";
59  Real lc[3].pin2.v(quantity = "ElectricPotential", unit = "V")
        "Potential at the pin";
60  Real lc[3].pin2.i(quantity = "ElectricCurrent", unit = "A") "
        Current flowing into the pin";
61  Real lc[3].inductor1.v(quantity = "ElectricPotential", unit =
        "V") "Voltage drop between the two pins (= p.v - n.v)";
62  Real lc[3].inductor1.i(quantity = "ElectricCurrent", unit = "A
        ", start = 0.0) "Current flowing from pin p to pin n";
63  Real lc[3].inductor1.p.v(quantity = "ElectricPotential", unit
        = "V") "Potential at the pin";
64  Real lc[3].inductor1.p.i(quantity = "ElectricCurrent", unit =
        "A") "Current flowing into the pin";
65  Real lc[3].inductor1.n.v(quantity = "ElectricPotential", unit
        = "V") "Potential at the pin";
66  Real lc[3].inductor1.n.i(quantity = "ElectricCurrent", unit =
        "A") "Current flowing into the pin";
67  parameter Real lc[3].inductor1.L(quantity = "Inductance", unit
         = "H", start = 1.0) "Inductance";
68  Real lc[3].pin.v(quantity = "ElectricPotential", unit = "V") "
        Potential at the pin";
```

```
69   Real lc[3].pin.i(quantity = "ElectricCurrent", unit = "A") "
       Current flowing into the pin";
70   Real lc[3].pin1.v(quantity = "ElectricPotential", unit = "V")
       "Potential at the pin";
71   Real lc[3].pin1.i(quantity = "ElectricCurrent", unit = "A") "
       Current flowing into the pin";
72   Real lc[3].capacitor1.v(quantity = "ElectricPotential", unit =
       "V", start = 0.0) "Voltage drop between the two pins (= p.
       v - n.v)";
73   Real lc[3].capacitor1.i(quantity = "ElectricCurrent", unit = "
       A") "Current flowing from pin p to pin n";
74   Real lc[3].capacitor1.p.v(quantity = "ElectricPotential", unit
       = "V") "Potential at the pin";
75   Real lc[3].capacitor1.p.i(quantity = "ElectricCurrent", unit =
       "A") "Current flowing into the pin";
76   Real lc[3].capacitor1.n.v(quantity = "ElectricPotential", unit
       = "V") "Potential at the pin";
77   Real lc[3].capacitor1.n.i(quantity = "ElectricCurrent", unit =
       "A") "Current flowing into the pin";
78   parameter Real lc[3].capacitor1.C(quantity = "Capacitance",
       unit = "F", min = 0.0, start = 1.0) "Capacitance";
79   Real constantvoltage1.v(quantity = "ElectricPotential", unit =
       "V") "Voltage drop between the two pins (= p.v - n.v)";
80   Real constantvoltage1.i(quantity = "ElectricCurrent", unit = "
       A") "Current flowing from pin p to pin n";
81   Real constantvoltage1.p.v(quantity = "ElectricPotential", unit
       = "V") "Potential at the pin";
82   Real constantvoltage1.p.i(quantity = "ElectricCurrent", unit =
       "A") "Current flowing into the pin";
83   Real constantvoltage1.n.v(quantity = "ElectricPotential", unit
       = "V") "Potential at the pin";
84   Real constantvoltage1.n.i(quantity = "ElectricCurrent", unit =
       "A") "Current flowing into the pin";
85   parameter Real constantvoltage1.V(quantity = "
       ElectricPotential", unit = "V", start = 1.0) "Value of
       constant voltage";
86 equation
87   assert(1.0 + resistor1.alpha * (resistor1.T_heatPort -
       resistor1.T_ref) >= 1e-15, "Temperature outside scope of
       model!");
88   resistor1.R_actual = resistor1.R * (1.0 + resistor1.alpha * (
       resistor1.T_heatPort - resistor1.T_ref));
89   resistor1.v = resistor1.R_actual * resistor1.i;
90   resistor1.LossPower = resistor1.v * resistor1.i;
91   resistor1.v = resistor1.p.v - resistor1.n.v;
92   0.0 = resistor1.p.i + resistor1.n.i;
93   resistor1.i = resistor1.p.i;
94   resistor1.T_heatPort = resistor1.T;
95   ground1.p.v = 0.0;
96   constantvoltage1.v = constantvoltage1.V;
97   constantvoltage1.v = constantvoltage1.p.v - constantvoltage1.n
       .v;
98   0.0 = constantvoltage1.p.i + constantvoltage1.n.i;
99   constantvoltage1.i = constantvoltage1.p.i;
```

```
100   lc[1].inductor1.L * der(lc[1].inductor1.i) = lc[1].inductor1.v
          ;
101   lc[1].inductor1.v = lc[1].inductor1.p.v - lc[1].inductor1.n.v;
102   0.0 = lc[1].inductor1.p.i + lc[1].inductor1.n.i;
103   lc[1].inductor1.i = lc[1].inductor1.p.i;
104   lc[1].capacitor1.i = lc[1].capacitor1.C * der(lc[1].capacitor1
          .v);
105   lc[1].capacitor1.v = lc[1].capacitor1.p.v - lc[1].capacitor1.n
          .v;
106   0.0 = lc[1].capacitor1.p.i + lc[1].capacitor1.n.i;
107   lc[1].capacitor1.i = lc[1].capacitor1.p.i;
108   lc[2].inductor1.L * der(lc[2].inductor1.i) = lc[2].inductor1.v
          ;
109   lc[2].inductor1.v = lc[2].inductor1.p.v - lc[2].inductor1.n.v;
110   0.0 = lc[2].inductor1.p.i + lc[2].inductor1.n.i;
111   lc[2].inductor1.i = lc[2].inductor1.p.i;
112   lc[2].capacitor1.i = lc[2].capacitor1.C * der(lc[2].capacitor1
          .v);
113   lc[2].capacitor1.v = lc[2].capacitor1.p.v - lc[2].capacitor1.n
          .v;
114   0.0 = lc[2].capacitor1.p.i + lc[2].capacitor1.n.i;
115   lc[2].capacitor1.i = lc[2].capacitor1.p.i;
116   lc[3].inductor1.L * der(lc[3].inductor1.i) = lc[3].inductor1.v
          ;
117   lc[3].inductor1.v = lc[3].inductor1.p.v - lc[3].inductor1.n.v;
118   0.0 = lc[3].inductor1.p.i + lc[3].inductor1.n.i;
119   lc[3].inductor1.i = lc[3].inductor1.p.i;
120   lc[3].capacitor1.i = lc[3].capacitor1.C * der(lc[3].capacitor1
          .v);
121   lc[3].capacitor1.v = lc[3].capacitor1.p.v - lc[3].capacitor1.n
          .v;
122   0.0 = lc[3].capacitor1.p.i + lc[3].capacitor1.n.i;
123   lc[3].capacitor1.i = lc[3].capacitor1.p.i;
124   resistor1.p.i + lc[3].pin1.i = 0.0;
125   resistor1.n.i + lc[2].pin2.i + lc[3].pin2.i + lc[1].pin2.i +
          constantvoltage1.n.i + ground1.p.i = 0.0;
126   lc[1].pin.i + constantvoltage1.p.i = 0.0;
127   (-lc[3].pin.i) + lc[3].inductor1.p.i = 0.0;
128   (-lc[3].pin1.i) + lc[3].capacitor1.p.i + lc[3].inductor1.n.i =
          0.0;
129   lc[2].pin1.i + lc[3].pin.i = 0.0;
130   (-lc[3].pin2.i) + lc[3].capacitor1.n.i = 0.0;
131   lc[3].capacitor1.p.v = lc[3].inductor1.n.v;
132   lc[3].capacitor1.p.v = lc[3].pin1.v;
133   lc[3].inductor1.p.v = lc[3].pin.v;
134   lc[3].capacitor1.n.v = lc[3].pin2.v;
135   lc[2].inductor1.p.i + (-lc[2].pin.i) = 0.0;
136   lc[2].inductor1.n.i + lc[2].capacitor1.p.i + (-lc[2].pin1.i) =
          0.0;
137   lc[2].pin.i + lc[1].pin1.i = 0.0;
138   (-lc[2].pin2.i) + lc[2].capacitor1.n.i = 0.0;
139   lc[2].capacitor1.p.v = lc[2].inductor1.n.v;
140   lc[2].capacitor1.p.v = lc[2].pin1.v;
141   lc[2].inductor1.p.v = lc[2].pin.v;
```

```
142   lc[2].capacitor1.n.v = lc[2].pin2.v;
143   lc[1].inductor1.p.i + (-lc[1].pin.i) = 0.0;
144   lc[1].inductor1.n.i + lc[1].capacitor1.p.i + (-lc[1].pin1.i) =
          0.0;
145   (-lc[1].pin2.i) + lc[1].capacitor1.n.i = 0.0;
146   lc[1].capacitor1.p.v = lc[1].inductor1.n.v;
147   lc[1].capacitor1.p.v = lc[1].pin1.v;
148   lc[1].inductor1.p.v = lc[1].pin.v;
149   lc[1].capacitor1.n.v = lc[1].pin2.v;
150   constantvoltage1.n.v = ground1.p.v;
151   constantvoltage1.n.v = lc[1].pin2.v;
152   constantvoltage1.n.v = lc[2].pin2.v;
153   constantvoltage1.n.v = lc[3].pin2.v;
154   constantvoltage1.n.v = resistor1.n.v;
155   constantvoltage1.p.v = lc[1].pin.v;
156   lc[3].pin1.v = resistor1.p.v;
157   lc[1].pin1.v = lc[2].pin.v;
158   lc[2].pin1.v = lc[3].pin.v;
159 end lcline;
```

Modelo A.8: Modelo 4.13 aplando con OpenModelica para N=3

A.9. Modelo *lcline* aplanado con ModelicaCC para N=3

```
1  model lcline
2    parameter Real resistor1_R(start=1);
3    parameter Real resistor1_T_ref=300.15;
4    parameter Real resistor1_alpha=0;
5    Real resistor1_R_actual;
6    Real resistor1_v;
7    Real resistor1_i;
8    Real resistor1_p_v;
9    Real resistor1_p_i;
10   Real resistor1_n_v;
11   Real resistor1_n_i;
12   parameter Boolean resistor1_useHeatPort= false ;
13   parameter Real resistor1_T=resistor1_T_ref;
14   Real resistor1_LossPower;
15   Real resistor1_T_heatPort;
16   Real ground1_p_v;
17   Real ground1_p_i;
18   constant Integer N=3;
19   Real lc_pin2_v[N];
20   Real lc_pin2_i[N];
21   parameter Real lc_inductor1_L[N](each start=1);
22   Real lc_inductor1_v[N];
23   Real lc_inductor1_i[N](each start=0);
24   Real lc_inductor1_p_v[N];
25   Real lc_inductor1_p_i[N];
```

```
26   Real lc_inductor1_n_v[N];
27   Real lc_inductor1_n_i[N];
28   Real lc_pin_v[N];
29   Real lc_pin_i[N];
30   Real lc_pin1_v[N];
31   Real lc_pin1_i[N];
32   parameter Real lc_capacitor1_C[N](each start=1);
33   Real lc_capacitor1_v[N](each start=0);
34   Real lc_capacitor1_i[N];
35   Real lc_capacitor1_p_v[N];
36   Real lc_capacitor1_p_i[N];
37   Real lc_capacitor1_n_v[N];
38   Real lc_capacitor1_n_i[N];
39   parameter Real constantvoltage1_V(start=1);
40   Real constantvoltage1_v;
41   Real constantvoltage1_i;
42   Real constantvoltage1_p_v;
43   Real constantvoltage1_p_i;
44   Real constantvoltage1_n_v;
45   Real constantvoltage1_n_i;
46 equation
47   assert(1+resistor1_alpha*(resistor1_T_heatPort-resistor1_T_ref
         )>=1e-15,"Temperature outside scope of model!");
48   resistor1_R_actual = resistor1_R*(1+resistor1_alpha*(
         resistor1_T_heatPort-resistor1_T_ref));
49   resistor1_v = resistor1_R_actual*resistor1_i;
50   resistor1_LossPower = resistor1_v*resistor1_i;
51   resistor1_v = resistor1_p_v-resistor1_n_v;
52   0 = resistor1_p_i+resistor1_n_i;
53   resistor1_i = resistor1_p_i;
54   ground1_p_v = 0;
55   for _Index_0 in 1:N loop
56     lc_inductor1_L[_Index_0]*der(lc_inductor1_i[_Index_0]) =
           lc_inductor1_v[_Index_0];
57     lc_inductor1_v[_Index_0] = lc_inductor1_p_v[_Index_0]-
           lc_inductor1_n_v[_Index_0];
58     0 = lc_inductor1_p_i[_Index_0]+lc_inductor1_n_i[_Index_0];
59     lc_inductor1_i[_Index_0] = lc_inductor1_p_i[_Index_0];
60     lc_capacitor1_i[_Index_0] = lc_capacitor1_C[_Index_0]*der(
           lc_capacitor1_v[_Index_0]);
61     lc_capacitor1_v[_Index_0] = lc_capacitor1_p_v[_Index_0]-
           lc_capacitor1_n_v[_Index_0];
62     0 = lc_capacitor1_p_i[_Index_0]+lc_capacitor1_n_i[_Index_0];
63     lc_capacitor1_i[_Index_0] = lc_capacitor1_p_i[_Index_0];
64   end for;
65   constantvoltage1_v = constantvoltage1_V;
66   constantvoltage1_v = constantvoltage1_p_v-constantvoltage1_n_v
         ;
67   0 = constantvoltage1_p_i+constantvoltage1_n_i;
68   constantvoltage1_i = constantvoltage1_p_i;
69   resistor1_T_heatPort = resistor1_T;
70   for i in 1:2 loop
71     constantvoltage1_n_v = lc_pin2_v[i];
72   end for;
```

```
73    constantvoltage1_n_v = lc_pin2_v[3];
74    constantvoltage1_n_v = ground1_p_v;
75    constantvoltage1_n_v = resistor1_n_v;
76    constantvoltage1_n_i+sum(lc_pin2_i[1:2])+lc_pin2_i[3]+
          ground1_p_i+resistor1_n_i = 0;
77    lc_pin_v[1] = constantvoltage1_p_v;
78    lc_pin_i[1]+constantvoltage1_p_i = 0;
79    for i in 2:3 loop
80      lc_pin1_v[i-1] = lc_pin_v[i];
81    end for;
82    for i in 2:3 loop
83      lc_pin1_i[i-1]+lc_pin_i[i] = 0;
84    end for;
85    lc_pin1_v[3] = resistor1_p_v;
86    lc_pin1_i[3]+resistor1_p_i = 0;
87    for i in 1:3 loop
88      lc_pin1_v[i] = lc_inductor1_n_v[i];
89      lc_pin1_v[i] = lc_capacitor1_p_v[i];
90    end for;
91    for i in 1:3 loop
92      (-lc_pin1_i[i])+lc_inductor1_n_i[i]+lc_capacitor1_p_i[i] =
            0;
93    end for;
94    for i in 1:3 loop
95      lc_inductor1_p_v[i] = lc_pin_v[i];
96    end for;
97    for i in 1:3 loop
98      lc_inductor1_p_i[i]+(-lc_pin_i[i]) = 0;
99    end for;
100   for i in 1:3 loop
101     lc_pin2_v[i] = lc_capacitor1_n_v[i];
102   end for;
103   for i in 1:3 loop
104     (-lc_pin2_i[i])+lc_capacitor1_n_i[i] = 0;
105   end for;
106 end lcline;
```

Modelo A.9: Modelo 4.13 aplando con ModelicaCC para N=3