Solving Dynamic Non-Causal Algebraic Equation Sets

Andrew Casey

McGill University acasey@cs.mcgill.ca

Abstract. We propose an interpretive system, δModelica, for solving dynamically-specified sets of non-causal equations by a combination of symbolic manipulation and numerical solving. Our system extends the system used by the μModelica compiler (as described in [5]). Equations are simplified and canonicalized, made causal (by matching variables to equations), sorted (with respect to dependencies), and then evaluated. When equations are added or removed, partial results are reused to reduce execution time. Our system also supports multi-threaded computation.

1 Introduction

Scientists are, for the most part, not computer programmers. They write their formulas not as lists of assignments but as sets of equations. Furthermore, once they have modelled a system by a set of equations, they expect entities to be able to enter and leave the system dynamically and to be able to update the set of equations accordingly.

The μModelica compiler (described in [5]) is an efficient system for solving statically-specified sets of non-causal equations. Rather than solving the entire system numerically, it pre-processes the set of equations, simplifying them and converting as many as possible to causal form. It then performs as much symbolic evaluation as possible, passing the remaining equations to a numeric solver (Octave).

Unfortunately, the μModelica compiler has very limited support for modifying the set of equations after compilation. In some situations, it is possible to "enable" and "disable" equations by multiplying both sides by a constant (namely, 1 or 0) but this system is ad hoc and can only handle a pre-determined number of additions.

We propose an extension of the μModelica system for handling dynamically-specified sets of non-causal equations. Our system, δModelica, is an interpreter that maintains a current set of equations and attempts to reuse partial results when equations are added to or removed from the current set. Furthermore, it attempts to solve the set of equations in a multi-threaded way to take advantage of the multi-core computers that are now available to most scientists.

Our system serves primarily as a proof-of-concept implementation and a demonstration tool. It presently handles only algebraic equations (whereas a

practical system would need to handle differential equations or at least have a concept of time). Furthermore, it is written in Java and interfaces naively with Octave. It is unlikely, therefore, to be useful for high-performance scientific computing in its present form.

2 Implementation

Our implementation of δModelica closely follows the design of μModelica, as described by [3], [4], and [5], but with a few notable differences, described below. δModelica was created without reference to the source code of μModelica.

We chose to implement δModelica in Java because we hope that the code will be reusable for another project that we are developing in that language.

2.1 Scanner and Parser

In general, a multi-pass compiler/interpreter can be designed in one of two ways. The object-oriented approach is to define a class for each syntactic entity and then add code for each pass to each class. For example, there might be an *Addition* class with methods for constant folding, pretty printing, etc. With this approach, syntactic entities are cleanly encapsulated, so it is very easy to add a new one. In particular, none of the existing classes need to be modified. However, adding a new pass is more complicated, requiring changes to all existing classes.

The dual approach is to define a class for each pass and have a method (or other branching construct) therein for each syntactic entity. For example, there might be a *ConstantFolding* class with methods for addition entities, subtraction entities, etc. With this approach, it is easy to add a new pass but complicated to add a new entity.

We decided to use the second approach for two reasons. First, we expected our syntax to be fairly stable because the notation for algebraic equations is quite standard. Second, we wanted each phase to have a clear and easily accessible result so that we could reuse as much information as possible. For example, we wanted to be able to store the result of applying constant folding to an expression so that if it reappeared in the modified set of equations, we could avoid re-applying the pass to that expression.

Having decided to forego the object-oriented approach, we decided to use the *SableCC* compiler compiler because of its convenient tree-walker interface. The grammar definitions for the scanner and parser are much as they would have been in any compiler compiler. The major difference is in the generated classes. SableCC provides a *DepthFirstAdapter* class that we used as the base class for all of our simplification passes.

We also appreciated SableCC's built-in concrete- to abstract-syntax conversion. While not all that we would have done by hand, it substantially reduced the work of flattening the concrete syntax tree (CST) and made custom tree node classes unnecessary. That is, the abstract syntax tree (AST) structure produced

by SableCC is so nice that we were able to use it directly for all of our further transformations.

We encountered only two minor deficiencies. First, there was no way to store numeric literals as numeric primitives in Java. They are presently stored as strings and converted to/from doubles as needed. This would, of course, be unacceptable in a high-performance scenario. Second, the SableCC AST structure supports lists of subnodes (e.g. a node may have a list of children rather than a predetermined number) but its clone mechanism does not deep clone lists. As a result, we found it necessary to write our own cloning code.

The μModelica scanner and parser are not described in [3], [4], or [5] so we cannot say how they compare.

2.2 Simplification

Initially, an equation is represented as an AST with constants and variables at the leaves and operators at the internal nodes. Clearly, a given expression admits many such representations. This presents two problems. First, the representation maybe be unwieldy – it may contain obviously unsimplified subtrees (e.g. with unfolded constants). Second, it is difficult to determine whether two expressions are equal if their representations are different. Canonicalizing all representations resolves the second problem. Furthermore, if the canonical representation requires simplification, then it resolves the first problem as well.

Unfortunately, expression simplification is a difficult problem ([3]). Since there is no "simplest" form for a given equation, we accept that there is no optimal solution and settle for incremental improvements.

Simplification consists of a number of passes over the AST of each non-causal equation. By subtracting the right-hand expression from the left-hand expression, we reduce each equation to a single expression, which must be equal to zero. It is to this unified expression that the simplifications are applied.

Overview The overall simplification process is as follows. The individual passes are described below.

- OpSimplifier
- BinOpFlattener & ConstantFolder
- Until fixed point:
  - ExponentDistributor
  - BinOpFlattener & ConstantFolder
  - ExponentGatherer
  - BinOpFlattener & ConstantFolder
  - LiteralDistributor
  - BinOpFlattener & ConstantFolder
- CanonicalSorter
- TermGatherer
- BinOpFlattener & ConstantFolder
- CanonicalSorter

OpSimplifier OpSimplifier replaces division by multiplication with a negative exponent, subtraction by addition with a negative constant multiplier, and unary minus by multiplication with a negative constant multiplier. This reduces the number of cases required by the other passes and allows more terms to be gathered into n-ary addition and multiplication expressions (see BinOpFlattener below). Clearly, this pass only needs to be performed once since none of the other passes introduce division, subtraction, or unary minus operators.

BinOpFlattener Though each addition node has a list of subexpressions, in the AST produced by the parser, each addition node has exactly two operands. Since addition is associative and commutative, we would like to collapse these into a small number of n-ary operators so that we have maximum flexibility while simplifying. This also helps us to produce canonical representations because the structure is less complicated.

If BinOpFlattener finds an addition or multiplication node with a child of the same type, it adds the child's children to the parent. By applying this process recursively, it is possible to flatten nested binary operators into a single n-ary operator.

BinOpFlattener also simplifies nested exponentiation operators, though not in the same way. Since exponentiation is neither commutative nor associative, it cannot be expressed as an n-ary operator. However, an expression $(a^b)^c$ can be simplified to a^{bc}.

Virtually all passes re-arrange the AST in such a way that addition or multiplication nodes may become nested under nodes of the same type, so this pass is frequently re-applied.

ConstantFolder ConstantFolder is perhaps the most important simplification pass. It performs several important functions. First, it evaluates any expressions whose operands are all constants, replacing the expressions with constants. This results in substantial simplifications.

Second, ConstantFolder coalesces the constant operands of n-ary operators. Once this pass has been run, one can be sure that each n-ary operator has at most one constant operand. This helps with a number of other passes.

Third, ConstantFolder applies the following simplifications (recommended by [3]):

- $0 + expr \rightarrow expr$
- $0 * expr \rightarrow 0$ [1]
- $1 * expr \rightarrow expr$
- $expr^1 \rightarrow expr$
- $expr^0 \rightarrow 1$
- $1^{expr} \rightarrow 1$

[1] This simplification is disabled by default because it can hide division by zero errors.

Finally, ConstantFolder cleans up any n-ary operators that have been simplified until they have only a single operator. This is particularly important for eliminating unnecessary levels of nesting.

As one might expect, ConstantFolder can gainfully be employed after every pass.

ExponentDistributor ExponentDistributor applies the transformation $(a * b)^{expr} \rightarrow a^{expr} * b^{expr}$. Though not a simplification, this transformation can expose some extra constant folding opportunities and makes like-terms easier to recognize later on.

ExponentGatherer ExponentGatherer applies the transformation $id^{expr1} * id^{expr2} \rightarrow id^{expr1+expr2}$. In addition to being a simplification, this transformation makes like-terms easier to recognize.

ExponentGatherer should be run after ExponentDistributor so that it can handle cases like $id1^{expr1} * (id1 * id2)^{expr2}$.

Note that ExponentGatherer does not simplify the new sum in the exponent. That is left to BinOpFlattener and ConstantFolder.

LiteralDistributor LiteralDistributor distributes constant multipliers across sums. This eliminates a layer of nesting, exposes more constant folding opportunities, and brings the expression closer to canonical form.

CanonicalSorter CanonicalSorter simply orders the subexpressions in an n-ary operator. In particular, they are ordered as follows.

1. Constants, in numerical order.
2. Variables, in alphabetical order.
3. Sums, in dictionary order (i.e. regard them as tuples of expressions and use the dictionary order).
4. Products, in dictionary order.
5. Powers, in dictionary order.

The actual order is less important than the existence of an order.

CanonicalSorter is applied only after the fixed point is reached because it is quite expensive and it does not contribute to any of the aforementioned simplifications. However, it is important for TermGatherer and affected by TermGatherer, so it is performed both before and after that pass.

CanonicalSorter is the only pass that does not need to be followed by BinOpFlattener and ConstantFolder – it does not require either.

TermGatherer Within an addition node, if two terms differ only in their constant multiplier (which will be the first subterm, if it exists), then TermGatherer combines them as like terms. For example, $2x^2$ and $3x^2$ are combined into $5x^2$, but $2x^2$ and wx^2 are not combined.

TermGatherer is essential for reducing the number of occurrences of a variable in an equation so that it can be more easily solved for that variable.

Fixed Point Observe that no one of the transformation passes undoes the work of any other. This means that the expression cannot oscillate between two (or more values). It is also impossible for the expression to grow indefinitely (since only ExponentDistributor and LiteralDistributor increase complexity and they do not create new opportunities for each other). Therefore, the fixed point loop must eventually terminate.

Comparison to μModelica μModelica performs a few transformations that δModelica does not. In particular, it has a general rule that $0^c = 0$ and it includes the inverse transformation of ExponentDistributor. δModelica omits the first because it is not true when $c \leq 0$ and will otherwise be caught by constant folding. It omits the second because it either causes problems with the fixed point loop or is so limited that it provides only an insignificant benefit.

μModelica also performs a lot more canonical sorting. In particular, it includes two sorting steps in its fixed point loop without any clear justification (perhaps it makes testing the applicability of other transformations faster). Otherwise, the pre-processing, fixed point loop, and post-processing are quite similar.

2.3 Causality Assignment

δModelica performs causality assignment in the same way as μModelica([5]). Artificial source and sink nodes are introduced; the source node is connected to all equation nodes; equation nodes are connected to the nodes corresponding to the variables for which they can be solved; and variable nodes are connected to the sink. All edges are assigned unit weight, and Dinic's algorithm for maximum network flow ([2]) is applied. Edges between equations and variables indicate which equations should be solved for which variables.

One possible difference (not addressed in [3], [4], or [5]) between δModelica and μModelica is the determination of which variables are solvable. In δModelica, an equation is considered to be solvable for a variable if that variable appears exactly once in the equation and not in an exponent. This is more restrictive than it needs to be, but spending excessive time solving every possible algebraic equation would be self-defeating since δModelica is an interpreter.

2.4 Equation Ordering

The equation ordering used by δModelica is more complicated than that described in [5] in two important ways. First, it explicitly addresses the dependencies of equations that are not solved for any variable (as a result of causality

assignment or because of the limited algebraic manipulation). Second, it produces a partial order (i.e. a directed acyclic graph (DAG)) rather than a total order (i.e. as a flat list).

The first step in ordering the equations is to construct a dependency graph. An equation solved for a variable depends on all other variables in the equation. An unsolved equation depends on all variables therein. If an equation depends on a variable for which there is no causal equation, then a dummy node is introduced until the sorting is completed. Dummy nodes are considered to depend upon all non-dummy nodes. This ensures that an equation depending on an unknown variable will be in a strongly-connected component (SCC) with all other equations that depend on that same variable.

Once the graph is built, a DAG of SCCs is constructed using the standard algorithm ([1]). SCCs may either be trivial (consisting of a single solved equation) or multiple (consisting of one or more unsolved equations). Dummy nodes are not included in this DAG. Though the SCCs form a DAG, they are returned as a topologically sorted list for convenience during solving.

2.5 Evaluation

Each SCC returned by the equation orderer is actually a *Runnable* object. Trivial SCCs compute their results by substituting in the values provided by their dependencies and multiple SCCs spawn Octave processes to solve their systems. Spawning a new Octave process, piping in the appropriate text and parsing the result introduces a fair bit of overhead, but numerical solving is already much slower than algebraic solving, so the difference is less important that it would otherwise be.

δModelica provides three different threading policies for processing the DAG of SCCs.

Singlethreaded The equation solver returns the SCCs in topological order, so they can simply be solved sequentially. This is how μModelica works.

Maximally Multithreaded Since each SCC is actually a *Runnable* object, each one can be given its own thread. Each SCC will wait until it is notified of the completion of all dependencies and then evaluate its own equation(s). This is very simple and provides maximum concurrency, but it has a lot of overhead because a lot of threads will be created needlessly. For example, it may be that the DAG is actually a list, with each SCC depending on a unique predecessor. In this case, no thread can work until its predecessor terminates so there is no concurrency, despite the large number of threads created.

In a practical system, this approach would be unworkable.

Capped Multithreading A more sophisticated approach is to maintain a pool of a fixed number of threads (probably corresponding to the number of cores available on the machine) and match up SCCs and threads as needed.

The algorithm works basically as follows. A priority queue is maintained with SCCs with satisfied dependencies ordered topologically. An SCC from the head of the queue is matched with a thread from the thread pool. When the thread finishes computing the result of the SCC, it requests a child of the SCC. If one of the children has all dependencies satisfied, then the thread starts processing it immediately. Otherwise, the thread is returned to the thread pool. This ensures that threads will not be started unnecessarily for linear segments in the DAG.

Processing children immediately, possibly out of topological order, can never produce a deadlock because children are only processed if they will not have to block. That is, a thread will never take an SCC out of topological order and then block. Since threads can only block on SCCs acquired in topological order, deadlock is impossible.

2.6 Incremental Processing

If care is taken to track additions and deletions of equations, then it is sometimes possible to update previous solutions rather than beginning from scratch.

Addition Only If new non-causal equations are added to the system but none are deleted, then an attempt is made to solve the new equations separately and integrate the two solutions. Even if this is not possible, many of the intermediate data structures can be updated rather than rebuilt. In particular, the list of equations, the lists of solvable variables, and the variable frequency counts can be computed separately for the new equations and appended to the existing values.

If the existing system seems to be inconsistent (as indicated by a fatal error flag), then the list of equations, the lists of solvable equations, and the variable frequency counts are updated and everything else is recomputed (i.e. causality assignment, equations ordering, and evaluation happen again).

If the new equations do not share any variables in common with the existing system, then they can be solved as a separate system and the results can be appended.

Otherwise, some judgment is required. If either the existing system or the new system dwarfs the other (as determined by user-specified configuration variables), then it is probably worthwhile to solve the larger one, substitute the results into the smaller one, solve the smaller one, and then append the results. This approach is somewhat risky because if the smaller system cannot be solved after substituting the results from the larger system, then the combined system must be solved from scratch. On the other hand, if the sizes are comparable, then the results are likely to be incompatible, so we simply solve the entire system from scratch (modulo reuse of existing data structures).

Deletion Only If some non-causal equations are deleted from the system but none are added, then an attempt is made to clean up the existing solution, without recomputing.

If the existing solution contains a fatal error or if a large proportion of the original system is deleted (as determined by user-specified configuration variables), then the system is simply solved from scratch because digging through the intermediate data structures is likely to be more expensive than just solving a smaller system. Some intermediate data structures are updated rather than rebuilt: the list of equations, the lists of solvable variables, and the variable frequency counts. It would probably be preferable to simply discard these data structures and start fresh, but this would interfere with the Mixed case described below.

If the deleted equations are not depended on by any non-deleted equations then they are simply removed from the system. This is determined by examining the DAG of SCCs described above. If all SCCs reachable from the SCC containing a deleted equation contain only deleted equations, then the SCC can be deleted.

Otherwise, if some non-deleted equations depend on deleted equations, then the system is solved from scratch (modulo the updated data structures mentioned above).

Mixed If equations are both added and removed (net change – the same equation cannot be both added and removed), then the additions and deletions are processed separately.

The deletions are processed first because deletion becomes more expensive for as the size of the base system grows whereas addition does not. That is, deletion is more expensive after than before addition whereas addition is unaffected by deletion.

There is, however, a crucial special case. If an equation defining a single variable (e.g. $x = 1$) is replaced by another equation defining the same variable (e.g. $x = 2$) then the single equation is changed in the dependency graph and the results are propagated through. This saves a significant amount of time when fiddling with values of initial values/conditions.

The potential downside to this approach is that processing the addition might require an almost complete rebuild, nullifying any work done by the deletion. Unfortunately, it is quite expensive to determine whether or not this rebuild will be necessary ahead of time, so no attempt is made to do so.

The main reason for using this simplistic approach is predictability. Since modifying an equation is conceptually the same as deleting the old one and then adding the new one (rather than the reverse), this is what the mixed case does.

3 Tests

We created a suite of automated functional tests for the scanner, the parser, the simplification passes, causality assignment, equation ordering, and system evaluation. Since the classes are nicely divided along these lines, the correspondence between the test classes and the actual classes is very clear.

4 Interfaces

Since δModelica is intended to serve as a backend for a variety of domain-specific scientific applications, it was explicitly designed to have a clean programmatic interface. For demonstration purposes, it also has a sample (graphical) user interface.

4.1 Programmatic Interface

δModelica provides three default solvers: numeric, naive, and incremental. The numeric solver simply passes the entire system to Octave. The naive solver uses the mixed symbolic/numeric approach described above as well as capped multithreading but solves from scratch every time. The incremental solvers uses capped multithreading and attempts to update existing data structures rather than solving from scratch.

The DAG of SCCs is the most fragile data structure and is therefore the most opaque. However, all of the simplification passes, variable finding, and causality assignment are exposed as part of the programmatic interface. This enables very flexible use of the library. For example, the incremental addition routine does not depend on any non-public functionality. (Unfortunately, the deletion routine modifies the equation ordering and is therefore dependent on library internals.)

4.2 User Interface

The sample graphical user interface (GUI) for δModelica is very simple. On the left hand side is a list of non-causal equations. Each has a tooltip showing either its simplified form or an appropriate error message (from the scanner or parser). When the user presses the "Solve" button, the incremental solver is called and the results are written to the list on the right hand side. The left hand side may be loaded from or stored to file and the right hand side may be exported. Both file formats are presently plain-text.

5 Correctness

Any solution returned by δModelica is guaranteed to be a valid solution to the given set of equations. That is, all equations will be satisfied if the solution values are substituted into them.

Many systems of algebraic equations have more than one solution. For example, the intersection of two spheres can be a circle so a system of two algebraic equations can have infinitely many solutions. Unfortunately, δModelica will never return more than one solution. To some extent this limitation is inherited from numerical solvers. A numerical solver cannot necessarily determine how many solutions a set of equations has and, even if it can, cannot necessarily find all of them in a reasonable period of time. Since any given solution may depend on a numerical solver, δModelica does not try to find all solutions.

It is quite possible that δModelica will not find a solution, even if one exists. This can occur if an earlier problem has multiple solutions, only one of which satisfies a later problem. For example, if $x^2 = 1$ and $y = (-x)^{\frac{1}{2}}$, then δModelica will fail to determine that $x = -1$ and $y = 1$ because it always guesses the positive root.

6 Performance

Without a reasonable number of realistic inputs (i.e. sets of non-causal equations), we cannot provide conclusive performance numbers. Instead, we provide these numbers as a sanity check – in at least one non-trivial example δModelica results in a speed-up.

All tests were performed on a 2.33 GHz Intel Core 2 Duo MacBook Pro with 2 GB of RAM running Mac OS X 10.4.10 , J2SE 1.5.0_07, and Octave 3.0.0 Testing. All figures give the average over the last 10 of 11 runs (the first being dropped because of classloading overhead).

6.1 Naive vs Numeric

As an input program, we used a discretization of the following system.

$$
\begin{aligned}
x_0 &= 1 \\
y_0 &= 1 \\
t_0 &= 0 \\
\frac{\partial x}{\partial t} &= x - ty \\
\frac{\partial y}{\partial t} &= y - tx
\end{aligned}
$$

The discretization takes 10 steps with $\Delta t = 0.1$. The Naive mechanism is significantly faster. Furthermore, since the Numeric time increases substantially with the number of steps, we see that the difference is not entirely due to overhead.

Method	# Steps	Time (ms)
Numeric	10	1253
Numeric	20	3389
Naive	10	29
Naive	20	49

6.2 Threads

To compare the different threading approaches, we created an input system that should maximally benefit from extra threads, especially ones that take account of dependency graph structure. The system consists of 1000 equations each of the forms $x_i = x_{i-1} + 1$ and $y_i = y_{i-1} + 1$. The Naive method was used for all tests.

Method	Time (ms)
Single-Threaded	2274
Maximally Multi-Threaded	2664
Capped Multi-Threaded (2)	2187

We see that even in this ideal case, there is only a small benefit from using capped multi-threading. However, as expected, maximal multi-threading introduces an unreasonable amount of overhead.

6.3 Naive vs Incremental

To illustrate the potential benefit of using incremental solving, we compared it to the naive mechanism on 500 steps of the differential system described above and then added a single step.

Method	500 Steps (ms)	501 Steps (ms)
Naive	4223	4296
Incremental	4463	2

We see that, while Naive takes roughly the same amount of time for both problems, Incremental takes virtually no time to add a step. This is because the original system has a good solution and dwarfs the addition so it is solved by substituting in known values.

The Incremental approach takes slightly longer on the original problem because it builds up slightly more intermediate data in order to be able to handle updates.

Clearly an improvement of this magnitude is not to be expected in all cases.

7 Conclusions

Non-causal equations are useful for scientists.

A mixed symbolic/numeric approach is much more efficient at solving sets of non-causal equations than a purely numeric approach.

Taking advantage of existing information can make updating a set of non-causal equations much more efficient.

Adding clever multi-threading to system evaluation results in a (small) performance improvement.

8 Future Work

A number of improvements suggest themselves for future versions of δModelica.

First, δModelica should be extended to handle differential equations. This should be a straightforward extension, using the same approach as in μModelica.

Second, the fixed point loop could be improved. In particular, the BinOpFlattener & ConstantFolder step is only useful if the preceding pass has actually changed the expression being simplified. Adding additional checks could eliminate a number of unnecessary clean-up passes. Some care is required, however, because the passes actually update expressions in place so cloning would be required. Furthermore, equality checking of expressions is expensive (recursive tree comparison). A better approach might be to use a cheap hashcode to eliminate some, but not all, redundant clean-up passes.

Third, δModelica makes very naive use of Octave. Instead of spawning a separate Octave process each time a numerical solution is required, it would be nicer to maintain a pool of available instances to avoid the start-up overhead. It might also be worthwhile to investigate using the Java Native Interface (JNI) to connect to the Octave C++ API directly.

Fourth, while finding all solutions is basically impossible because of the reliance on numerical solvers for hard subproblems, it would be nice to do a better job of retaining multiple solutions to algebraically solved subproblems.

Fifth, better initial guesses should be provided to the numerical solver based on information about other equations. The current implementation always starts with 0 as the initial solution and it might be possible to do better.

Finally, we hope to integrate the lessons learned from building δModelica into the new McLab project being developed in the Sable lab. McLab is to be a compiler framework for scientific programming languages and non-causal equation solving can be quite valuable to scientists.

Acknowledgments. We gratefully acknowledge the help of Hans Vangheluwe, who proposed this investigation and provided useful references.

References

1. Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford: Introduction to Algorithms, Second Edition. Sect. 22.5. MIT Press, Cambridge, MA (2001)
2. Dinic, E.A.: Algorithm for solution of a problem of maximum flow in a network with power estimation. Soviet Math. Dokl., 11, 1277-1280 (1970)
3. Indrani, A.V.: Some issues concerning Computer Algebra in AToM^3. Technical Report, School of Computer Science, McGill University, Montreal, QC (2003)
4. Vangheluwe, Hans, Sridharan, Bhama, Indrani, A.V.: An algorithm to implement a canonical representation of algebraic expressions and equations in AToM^3. Technical Report, School of Computer Science, McGill University, Montreal, QC (2003)
5. Xu, Weigao "Steven": The Design and Implementation of the μModelica compiler. MSc. Thesis. School of Computer Science, McGill University, Montreal, QC (2005)