CoJava: A Unified Language for Simulation and Optimization

Alexander Brodsky[1,2] and Hadon Nash[2]

[1] George Mason Unviersity, Virginia, USA, brodsky@gmu.edu
[2] Adaptive Decisions, Inc., Maryland, USA, hnash@adaptivedecisions.com

Abstract. We have proposed and implemented the language CoJava, which offers both the advantages of simulation-like process modeling in Java, and the capabilities of true decision optimization. By design, the syntax of CoJava is identical to the programming language Java, extended with special constructs to (1) make a non-deterministic choice of a numeric value, (2) assert a constraint, and (3) designate a program variable as the objective to be optimized. The semantics of CoJava interprets a program as an optimal nondeterministic execution path, namely, a path that (1) satisfies the range conditions in the *choice* statements, (2) satisfies the assert-constraint statements, and (3) produces the optimal value in a designated program variable, among all execution paths that satisfy (1) and (2). To run a CoJava program amounts to first finding an optimal execution path, and then procedurally executing it. We have developed a CoJava constraint compiler based on a reduction of the problem of finding an optimal execution trace to a standard symbolic formulation, reinterpreting Java code as a symbolic constraint construction, and solving the resulting optimization problem on an external solver. To demonstrate the power of CoJava, we have implemented a realistic problem in the area of robot arm control in CoJava. The robot arm is constructed using self-contained components implemented as CoJava classes, that model robot's arm movements based on Newton's laws.

1 Introduction

Both numeric simulation and constraint-based optimization are successfully applied in wide variety of domains. This paper is concerned with developing a unified object-oriented (OO) language supporting both simulation modeling and decision optimization.

The Problem: Simulation vs. Optimization Models

In decision optimization problems one often needs to model a real-world process, such as a supply chain, an interaction of physical devices, a chemical process, or activities of a robot. However, describing a process using traditional operations research (OR) modeling, with decision variables, constraints, and objective functions, is quite a challenging task for non-OR professionals, even for those with general computer science and programming skills.

The reason for that challenge is that the elements of an OR model are abstract constraints, which have only an indirect connection to elements of a real-world process. For example, one equation may combine elements from several real-world devices. Also, the notions of order and timing of events are usually not explicit in OR models, which puts additional burden on the modeler. Furthermore, the execution of the optimization is typically a black box for the modeler, with no clear connection to the flow of the real world process. This makes debugging of an optimization model a challenging task. If the optimization fails there is no clear explanation for the failure. Finally, OR models typically lack the modularity of modern object-oriented languages, so they tend to become difficult to maintain over time (like "spaghetti code").

By contrast, simulations are generally well understood by software developers. The elements of a simulation are state variables and state-transitions, which have a clear one-to-one correspondence with elements of a real-world process. Every quantity from the real-world process is represented by a single state variable, so there is little room for confusion. Real-world time and sequence of events correspond to time and sequence in the running simulation in an obvious way. Also, the "cause and effect" progression of the simulation is easy to follow. If the simulation fails, the exact time and place of the failure is reported. Finally, simulation modelers can practice modern object-oriented software engineering. Complex building blocks can be modeled using simpler building blocks, and so on. In fact, modern OO languages have been derived from early simulation systems.

While simulation offers numerous advantages in ease of modeling and debugging, OR modeling has one major advantage. If modeled correctly using a manageable constraint domain such as LP or MILP, an optimization problem can be solved efficiently using existing solvers with sophisticated optimization algorithms. By contrast, no such solvers exist for simulation models. Typically, simulations are optimized by choosing parameters manually. An optimization layer can be added by running a simulation multiple times, with possible heuristics. However, such a search cannot compete with performance of solvers on manageable constraint domains.

Contributions

We have proposed and implemented the language CoJava, which offers both the advantages of simulation-like process modeling in Java, and the capabilities of true decision optimization.

By design, the syntax of CoJava is identical to the programming language Java, extended with special constructs to (1) make a non-deterministic choice of a numeric value, (2) assert a constraint, and (3) designate a program variable as the objective to be optimized.

A CoJava program defines a set of nondeterministic execution paths, each being a program run with specific selection of values in the choice statements. The semantics of CoJava interprets a program as an optimal nondeterministic execution path, namely, a path that (1) satisfies the range conditions in the *choice*

statements, (2) satisfies the assert-constraint statements, and (3) produces the optimal value in a designated program variable, among all execution paths that satisfy (1) and (2). To run a CoJava program amounts to first finding an optimal execution path, and then procedurally executing it.

To optimize a process, each real-world device or facility is modeled, tested and debugged in pure Java as a class of objects with private state and public methods which change the state. A process is described as a method of a separate class, which invokes methods of the model objects passing non-deterministic choices for arguments, and which designates an optimization objective.

For model developers, it appears as if the program has simply followed a single execution path which coincidentally produces the optimal objective value. Since CoJava builds on software developers' existing skills, the learning curve for them is minimal. For OR professionals, CoJava enables development of decision models in the most natural way, which preserves the one-to-one correspondence with the components of a real-world process. Moreover, it provides OR modelers with the powerful OO language features, and permits complex models to be organized into self-contained business objects and processes, which reduces development time and allows easy extensibility.

To be able to find an optimal execution path for a CoJava program, we have developed a reduction to a standard constraint optimization formulation. Constraint variables represent values on program variables that can be created at any state of a non-deterministic execution. Constraints encode transitions, triggered by CoJava statements, from one program state to the next, and also capture conditions in the assert statements.

The reduction and the implementation are made under three simple restrictions, explained in the paper. Namely, that (1) Boolean exit condition in loops can not involve non-deterministic values, (2) recursive method calls cannot be made from a non-deterministic conditional statement, and (3) an `objective` parameter requested to be optimized cannot appear in a non-deterministic conditional statement.

The restrictions insure that (1) the length of any execution trace, and thus a number of values generated, are not dependant on non-deterministic choices, and (2) any non-deterministic `choice` statements, and Boolean conditions on non-deterministic values in `assert` statements have a unique corresponding `objective` parameter, which needs to be optimized.

We have developed a CoJava constraint compiler, which is based on the reduction of the optimization problem to a standard formulation. The compiler operates by first translating the Java program into a very similar Java program in which the primitive numeric operators and data types are replaced by symbolic constraint operators and data types. This intermediate java program functions as a constraint generator. This program is compiled and executed to produce a symbolic decision problem. The decision problem is then submitted to an external optimization solver.

To demonstrate the power of CoJava, a realistic problem in the area of robot arm control has been implemented. The robot arm is constructed using self-

contained components implemented as CoJava classes, that model robot's arm movements based on Newton's laws.

Related Work

CoJava addresses the goals of constraint modeling and object oriented simulation. Object oriented simulation has traditionally been approached through procedural object oriented languages, such as Smalltalk [**?**] and Java [**?**]. These languages start with a syntax for variable assignment and add support for modular organization of procedures. There are many specialized object oriented simulation languages such as Simula [**?**] and ModSim [**?**], an there are simulation environments layered on top of existing object oriented languages such as Silk [**?**] and JWarp [**?**]. These languages allow complex models to be constructed and maintained effectively, but lack support for systematic optimization.

Constraint modeling has traditionally been approached through specialized constraint modeling languages, such as AMPL [**?**] and GAMS [**?**]. These languages start with a syntax for equations and layer additional support for organizing equations and other constraints. They enable systematic optimization, but they require explicit definition and maintenance of equations and other constraints.

Constraint programming languages, such as OPL [**?**] and CLP [**?**], allow developers to specify strategies for solving optimization problems. Certain constraint programming languages provide support for object oriented modeling as well. However, the focus of constraint programming is on solving optimization problems rather than modeling such problems.

In recent years, there has been considerable interest in languages that combine constraints with object oriented programming. These languages are motivated by the need for modular construction of optimizable models. Languages such as Cob [**?**] and Siri [**?**] add object oriented modeling constructs, such as inheritance and encapsulation, to an equational syntax. These languages provide a very clean representation for steady-state optimization problems, but they don't model state changes in the direct way that procedural languages do.

Other languages combine constraints with object oriented procedures in a "hybrid" fashion, maintaining program states and constraints side by side. Some languages, such as CCUBE [**?**] and Lyric [**?**] add explicit constraint objects to a procedural language, allowing procedures for building and querying a database of constraints. Other languages such as ThingLab [**?**] and Kaleidoscope [**?**] impose both state changes and constraints on the same object attributes. These languages provide separate procedure execution semantics and constraint solving semantics which interact during program execution.

The languages most closely related to CoJava are those that translate procedural algorithms into declarative constraints. These languages unify procedural and constraint semantics, such that the same program statements determine both interpretations. The language Modelica [**?**] supports unified models, which can define both simulation and optimization problems. Modelica models are translated into equations, which may in turn be solved by an optimizer or sorted

and compiled into an efficient sequential procedure. Within pure functions (functions without side effects), Modelica can also translate procedural algorithms into constraint equations. Within these functions, Modelica gains the advantage of specifying constraints using familiar procedural operations and flow of control. However, Modelica is fundamentally an equational language, and it supports procedural algorithms only in this limited context.

By contrast, CoJava has a thoroughly procedural object oriented syntax and semantics, (which is in fact identical to that of Java). CoJava presents the developer with no visible boundary between procedures and constraints. Familiar procedural operations and flow of control can be used uniformly throughout an entire model, or even throughout an entire software system. Our philosophy is to minimize the learning curve for developers, and to minimize the "impedance mismatch" between procedures and constraints, by conforming to a single well understood syntax and semantics. CoJava gives developers the flexibility to move model components freely back and forth between procedural algorithms and declarative optimization models. We believe this capability is unique to CoJava.

2 Syntax and Dual Semantics of CoJava

Syntax and Procedural Semantics

By design, the syntax of CoJava is identical to that of the Java programming language, extended with one special library class, and a few restrictions on how its methods can interact with the rest of the program. CoJava has two complementary semantics: the regular *procedural* semantics of Java, which can be used for a *simulation* process, and an additional decision *optimization* semantics.

More specifically, CoJava adds the following special class to Java:

```
public class DecisionMaker {
  public double choice(double min, double max) {...}
  public double checkMinObjective(double objective) {...}
  public double checkMaxObjective(double objective) {...}
}
```

CoJava's procedural semantics is identical to the standard Java semantics, where the semantics of the `DecisionMaker` class methods are as follows. The method `choice(min,max)` returns a single specific value between `min` and `max`. Note that the value for `min` may be negative infinity, and for `max` positive infinity. This method may be implemented by the user, and an optional default implementation (e.g., random selection, using a particular distribution) is provided by the system.

The methods `checkMinObjective` and `checkMaxObjective`, in the procedural semantics, do nothing but output the value of the parameter `objective`. Sometimes we'll be using the name `checkObjective` to refer to either of the two methods.

CoJava program can also use the Java command `assert(booleanCondition)` with the standard procedural semantics, namely the program will report an error if the `booleanCondition` is not satisfied.

We will explain the optimization semantics, and the limitation on how the methods of DecisionMaker interact with the rest of the program in separate subsection. To first understand the concepts intuitively, we start with an example.

Example

As an illustration of the syntax and semantics of CoJava, consider the following fragment of a CoJava program, which models the production of two products, using one raw material drawn from a limited inventory.

```
double inventory = 50.0;
double qtyProduct1 = decisionMaker.choice(0.0, 20.0);
double qtyProduct2 = decisionMaker.choice(0.0, 20.0);
double materialUsed = 2.0 * qtyProduct1 + 1.0 * qtyProduct2;
double inventory = inventory - materialUsed;
assert inventory >= 0.0;
double profit = 5.0 * qtyProduct1 + 1.0 * qtyProduct2;
decisionMaker.checkMaxObjective(profit);
```

Intital `inventory` of the raw material, which is consumed in production, is 50.0. Then, using the method `choice`, the quantity to produce for product 1 and 2 is selected, each restricted to be between 0.0 and 20.0. The amounts 2.0 and 1.0 of the raw material are needed to produce 1 unit of product 1 and 2, respectively. Therefore, `2.0 * qtyProduct1 + 1.0 * qtyProduct2` is the amount of raw material to be used. Then, `inventory` is updated to reflect the materials consumed. The `assert` statement verifies that the `inventory` remains non-negative, and halts the program otherwise. The `profit` per unit for products 1 and 2 are 5.0 and 1.0 respectively, and therefore, the total `profit` is `5.0 * qtyProduct1 + 1.0 * qtyProduct2`. Finally, program outputs the `profit` using the method `checkMaxObjective`.

If we were to compile and run the example procedure, particular quantities would be chosen for each product arbitrarily and a particular value would be computed for `profit`. This `profit` would typically not be optimal.

In addition to the standard *procedural* semantics, we propose a decision *optimization* semantics. For the production example, the optimization semantics will permit quantities of the products to be determined, which are optimal in terms of `profit`. More specifically, in the optimization semantics, the method `choice` is interpretted as a non-deterministic selection of a value from a range. A series of specific selection from each `choice` statment will define an execution trace in a non-deterministic program. Thus, a program with `choice` statements defines a set of possible execution traces, some of which may fail by not satisfying an `assert` statement. Some execution traces will reach the `checkObjective` statement.

The decision optmization semantics of CoJava interprets each program as an optimization problem: the search space is the set of all execution traces that reach the `checkMaxObjective` statement; the objective function is the value passed to the `checkObjective` method. In our example, a solution to the optimization problem will give quantities of products that maximize the `profit`.

Unfortunately, even for the simple production example, there are infinitely many execution traces, because values in non-deterministic choices range over a continuous interval. Fortunately, it is possible to reduce the optimization problem over the set of all execution traces to a standard constraint optimization form in terms of constraint variables, constraints and the objective function. Intuitively, this is done by encoding, with constraint variables, the states of program execution, and by encoding with constraints valid transitions from one program state to the next state in an execution trace. To illustrate, we show constraint encoding for the production example.

Initial program state is encoded by the empty conjunction CS of constraints, i.e., TRUE. After the the first assignment, the new program state is represented by a constraint variable $inventory_1$, which represents a value in the program variable `inventory` after the assignment is made. The constraint $inventory_1 = 50.0$ is added to the constraint store CS.

Assignments with the `choice` method on the right hand side are encoded as range constraints, within which a non-deterministic choice can be made. Thus, after the two `choice` statements, the constraints

$$0.0 \leq qtyProduct1_1 \leq 20.0$$
$$0.0 \leq qtyProduct2_1 \leq 20.0$$

are added to the constraint store CS. Note that two additional constraint variables have been created.

The forth and the fifth assignments introduce two additional equations to be added to CS

$$materialUsed_1 = 2.0 * qtyProduct1_1 + 1.0 * qtyProduct2_1$$
$$inventory_2 = inventory_1 - materialUsed$$

Notice the new constraint variable $materialUsed_1$. Also, note that a new constraint variable $inventory_2$ was created for the program variable `inventory`, which is necessary to represent the value of `inventory` in the new program state.

As the assert statement is encoded, a single additional inequality $inventory_2 > 0.0$ is added to the constraint store.

Similarly, the next assignment is encoded by adding the equation

$$profit_1 = 5.0 * qtyProduct1_1 + 1.0 * qtyProduct2_1$$

The constraint store CS is now:

$$
\begin{array}{l}
CS = inventory_1 = 50.0 \wedge \\
0.0 \leq qtyProduct1_1 \leq 20.0 \wedge \\
0.0 \leq qtyProduct2_1 \leq 20.0 \wedge \\
materialUsed_1 = 2.0 * qtyProduct1_1 + 1.0 * qtyProduct2_1 \wedge \\
inventory_2 = inventory_1 - materialUsed \wedge \\
inventory_2 > 0.0 \wedge \\
profit_1 = 5.0 * qtyProduct1_1 + 1.0 * qtyProduct2_1
\end{array}
$$

Finally, the `checkMaxObjective(profit)` statement is encoded as the decision optimization problem:

$$maximize profit_1 s.t. CS$$

In fact, if we ran this problem using an LP solver, we would get the answer $qtyProduct1_1 = 20.0$ and $qtyProduct2_1 = 10.0$.

Restrictions on DecisionMaker Class Methods

Certain restrictions on how the methods `choice(...)`, `checkObjective`, and the command `assert` interact with the rest of CoJava program are imposed to make the optimization semantics well-defined and computatable.

More specifically, the purpose of the restrictions is to control the size of the set of values that are computed during any execution trace of the program. As long as the program computes a predictable number of values, we can identifiy a correspondence between values across all program traces.

To formulate the restrictions, we use the notion of *non-deterministic values*, or ND-values for short, which we define recursively as follows.

- The output of a `choice` method is ND-value
- A variable is an ND-value, if it appears on the left-hand side of an assignment with an ND-value on the right-hand side.
- A variable is an ND-value, if it appears on the left-hand side of an assignement that appears in the THEN or ELSE part of a conditional statement, where the Boolean condition is an ND-value.
- The result of an arithmetic or Boolean operation on one or more ND-values is an ND-value.

We also say that a conditional statement is ND, if its Boolean condition is ND. A LOOP statement is ND, if its exit Boolean condition is ND. A method call is ND, if it is done from within an ND conditional statement. We also say that a variable, expression, conditional statement etc. are *deterministic* to mean the negation of being *non-deterministic*.

The following simple restrictions are imposed in order to make the number of values computed by the program independent of the non-determinstic choices, and associate at most one `checkObjective` method call with each `choice` call and `assert` statement.

- No ND loops
- No ND recursive method calls
- No ND calls for `checkObjective`.

The first restriction controls the number of iterations of each loop. As long as the loops exit conditions are deterministic, the loop will continue for a determinstic number of iterations. If a loop executed a non-determinstic number of iterations, it would compute a non-determistic number of values.

The second restriction controls the depth of recursive calls. By prohibiting recursive calls within non-determinstic conditionals, we prevent the depth of recursion from depending on non-determinstic choices. We do allow arbitraty non-recursive method calls, whether or not they are deterministic, and also recursive method calls as long as they are deterministic.

The third restriction, namely that no `checkObjective` is called from a ND conditional statement, makes sure that per given input to the program, (1) all `checkObjective` method calls have a total ordering, which is deterministic, and (2) that every execution trace that goes through a specific `choice` or `assert` statement will deterministically "continue" to a unique "nearist" `checkObjective` call (if there is any). In this case, we say that such a `choice` or `assert` statement is in the *scope* of that nearest `checkObjective` call.

For example,...

We are now ready to define optimization semantics in a general way.

Optimization Semantics

The decision *optimization* semantics interprets the method `choice` as a non-deterministic selection of a value from a range. A series of specific selections from each `choice` invocation will define an execution trace in a non-deterministic program. Thus, a program with `choice` method invocations defines a set of possible execution traces, some of which may fail by not satisfying the condition in an `assert` statement. Some execution traces will reach a given `checkMinObjective` or `checkMaxObjective` statement.

Case 1: Single checkObjective Call, as Last Program Statement We assume here that all the restrictions outlined are satisfied. Given a CoJava program P, input I, and the `checkObjective(v)` statement S as the last program statement, we denote by ET the set of all execution traces e that reach S. For a particular execution trace $e \in ET$, we denote by $v(e)$ the value of the program varialbe v at the statement S.

The decision optimization semantics interprets the program as a decision optimizatin problem OP:

$$Optimize \;\; v(e) \; s.t. \;\; e \in ET$$

where *Optimize* stands for *Minimize* in the case of `checkMinObjective` and for *Maximize* in the case of `checkMaxObjective`.

Note that a solution to this problem may not be unique, as more than one execution trace $e \in ET$ may have the minimal/maximal $v(e)$. An optimal execution trace e defines the values for each execution of a `choice` method.

An execution of the program P according to optimization semantics is viewed as a regular procedural execution where the values returned by each `choice` statement are those corresponding to an optimal execution trace e.

Case 2: No checkObjective in the program In this case, the decision optimization semantics interprets the program as a satisfaction problem SP: Find $e \in ET$, where ET is the set of all execution traces that reach a special `NOOP` method (which does nothing) at the end of the program. We call such execution traces *valid*.

An execution of the program P according to optimization semantics is viewed as a regular procedural execution where the values returned by each `choice` statement are those corresponding to a valid execution trace e.

Case 3: One or More checkObjective Calls According to Restrictions We do not discuss this general case in detail in this paper, nor was it implemented. Here we only provide a general idea. Because of the restrictions, (1) all `checkObjective` method calls have a total ordering, which is deterministic, and (2) that every execution trace that goes through a specific `choice` or `assert` statement will deterministically "continue" to a unique "nearest" `checkObjective` call (if there is any). In this case, we say that such a `choice` or `assert` statement is in the *scope* of that nearest `checkObjective` call.

The idea here, is to consider an execution as split into sections, in the order of `checkObjective` calls, each with the `choice` and `assert` statements in its scope, and apply Case 1 on all but the last section, and Case 2 on the last.

2.1 Generation of Standard Constraint Optimization Formulation

In this subsection we briefly describe how a decision optimization problem in terms of constraint variables, constraints and an objective function is formulated, so that it would be equivalent to the optimization problem in the *optimization* semantics of CoJava. To do that, we conceptually describe a modified Java program that generates symbolic constraints, to be used in the optimization problem.

First, we introduce symbolic expessions types, for *arithmetic numeric* and *Boolean* types. This is done by implementing `symbExpression` class. This class has methods which are their arithmetic counterparts: add, subtract, multiply etc., that construct more complex arithmetic symbolic expressions from simpler ones.

Similarly, symbolic atomic constraints correspond to inequalities or equations between two symbolic arithmetic expressions. Symbolic constraints are either atomic, or constructed using Boolean operators of simpler symbolic constraints in the standard fashion.

The Java code of the CoJava program is modified as follows. All numeric and Boolean types are replaced with their symbolic counterparts, and so are the operators. Some program statements extend CS (i.e., conjuncts it with additional constraints). Initially, constraint store CS is empty. Then:

- For assignement statement of the form `v = AE`, where AE is an arithmetic expression, a new constraint variable v_{new} is generated for the program variable v, a symbolic arithmetic expression *Symb(AE)* is created for the RHS `AE`, and the constraint $v_{new} = Symb(AE)$ is added to the constraint store CS. Also, a program variable v is converted to the type `symbolic expression` in the modified program.
- A conditional statement of the form

  `if C { S1 } else { S2 }`

  where `C` is a non-deterministic (ND) Boolean condition, and `S1` and `S2` are statements, is first replaced with two conditional statements

  `if C { S1 }; if not C { S2 }`

- For a conditional statement of the form `if C { S }` , including those generated from `if ... else ...` statement, the following constraints are added to CS:

  $$\begin{array}{rcl} symb(C) & \longrightarrow & Constraints(S) \\ \neg symb(C) & \longrightarrow & Equalities \end{array}$$

  where *Constraints(S)* stand for the constraints that would be generated for the statement `S` if it was executed unconditionally, and *Equalities* is the set of equality constraints of the form $v_{new} = v_{old}$, where v_{new} is the newest constraint variable v_{new} generated for a program variable `v` in *Constraints(S)*, and v_{old} is the last constraint variable for `v` before the conditional statement.
- For an assignment `v = choice(min,max)`, a constraint $min \leq v_{new} \leq max$ is generated and added to CS
- For an `assert(C)` statement a constraint *symb(C)* is generated and added to CS

For Case 1 of optimization semantics, where `checkObjective(objective)` call is the last statement of the program, the optimization problem constructed is:

$$Optimize \quad objective_{current} \quad s.t. CS$$

where *Optimize* stand for *Minimize* in the case of `checkMinObjective` and *Maximize* in the case of `checkMaxObjective`, and $objective_{current}$ is the most recent constraint variable for the program variable `objective`.

3 Overview of Implementation

We have developed a *constraint compiler* for the the CoJava language. The constraint compiler translates a nondeterminstic simulation procedure into an equivalent decision problem. The input is a program in the CoJava (our restricted version of Java). The resulting decision problem consists of a set of equations and inequalities in the mathematical modeling language AMPL.

The overall flow of the costraint compiler is as follows: First, a simulation procedure is made nondeterminstic by initializing it with values from the nondeterminstic choice library, and designating its output as an objective value. This requires no change to the procedure itself, only to its parameters and return value. Next, the procedure is transformed to create a constraint generator procedure. This involves uniformly converting all of its numeric data types to symbolic expression data types. Next, the constraint generator is compiled and executed (using a standard java compiler). The result generated by this procedure is a set of symbolic expression data structures, represent the nondeterministic output of the simulation procedure. Finally, these symbolic expressions are translated into a standard constraint proramming language such as AMPL.

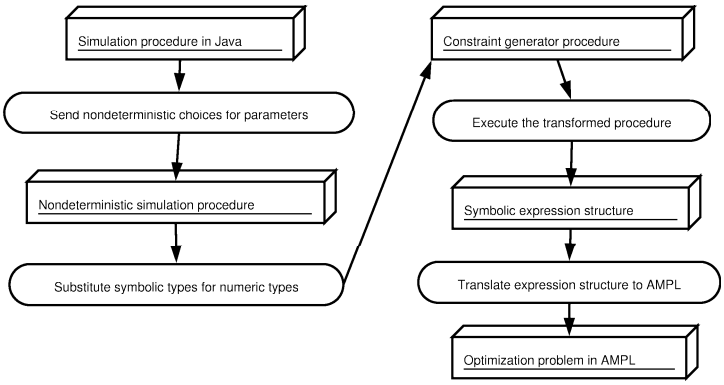

Fig. 1. Diagram of the Constraint Compiler

4 Realistic Decision Problem

In this section, we present a realistic problem to demonstrate the capabilities of the unified language. We show how the unified langauge can be used to model a complete decision problem, and how the decision problem can be solved using an existing solver.

We have selected a problem in the area of robot arm control. Basically, we simulate the acceleration of a robot arm under the forces of gravity and several

motors. This problem includes an element of time, so it illustrates the relationship between a sequential process and the immutable constraints that describe it. The objective for this decision problem is to drive the robot arm as close as possible to a certain final position and velocity.

We have modeled this optimization problem with a non-linear optimizer in mind. We have avoided the use of conditional statements so as not to create a highly combinatorial problem for the optimizer. We were able to restrict the numeric formulas to second degree polynomials throughout the simulation. The simulation procedure integrates Newton's equations of motion by iteratively updating velocities based on current positions, and updating positions based on current velocities. After every few timesteps, new forces are chosen for the motors driving the robot arm.

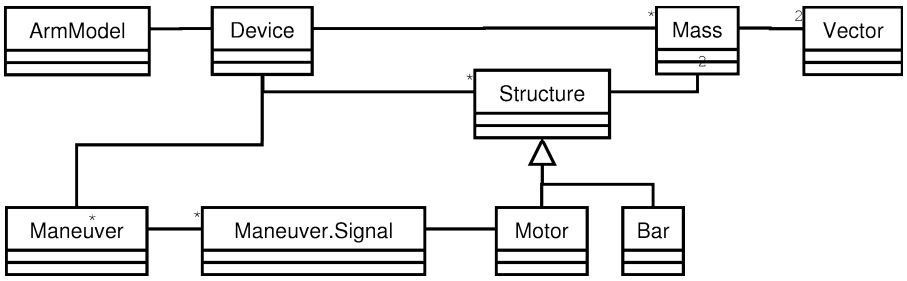

Fig. 2. Class Diagram of the Simulator

The simulation procedure is implemented as a small hierarchy of object classes, depicted in Figure 2. You can see that we have built a very small linear algebra library, a small mechanical physics library, and a simple discrete time simulation library. These software components are assembled together to produce a complete simulation procedure, and then to define a complete decision problem. Most of the advantages of object oriented design are available within the unified language, including encapsulation, visibility control, inheretence, and limited polymorphism.

For our decision problem, we have assembled a very simple robot arm, although more elaborate arms can be assembled easily. We have limited the robot arm to three point masses, three elastic bars, and two linear motors. Also, we have limited its motion to two dimensions. This is accomplished simply by positioning all of the masses in a single plane. Our symbolic expression library is able to detect that all z-axis coordinates evaluate to the constant zero, and to omit them completely from the optimization problem. In order to limit the decision problem to a manageable number of constraints, we limit the simulation to a four of timesteps. Figure 3 in Appendix depicts the simple robot arm.

Figure 4 is an excerpt of the simulation procedure in CoJava. This procedure is standard Java source code, and can be compiled an run using any Java com-

piler. As a simulation procedure, the program can be run with arbitrary choices (which may be made by hand) for the motor forces. The result is a single simulation history and a single objective value for each choice made. The objective is to minimize the sum of square error in final position and velocity.

For our simulation procedure, the constraint compiler produces 229 constraints and 226 contraint variables. We did not include an excerpt from the constraints in AMPL format due to space limitation. There, the constraint variables `distance2_17` and `speed2_1` represents the objective function, and the constraint variables `range_1`, `range_2`, `range_3`, `range_4` represent the decision variables. Both MINOS and SNOPT did successfully solve this optimization problem.

5 Conclusions and Future Work

We presented a unified language with complementing procedural and optimization semantics. Some questions, such as how to generalize CoJava to serve as a general computational paradigm, how to extend it with intelligent debugging capability, how to add CP facilities, and how to extend underlying constraint solvers remain for future research.

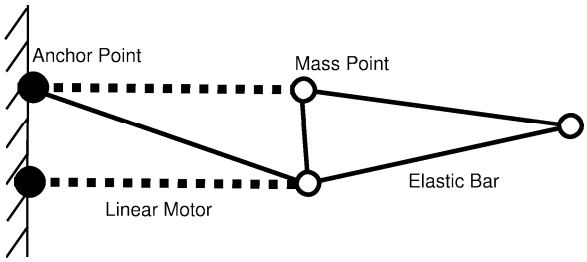

Fig. 3. Diagram of the Modeled Robot Arm

```
/**
 * Advance the physical simulation by one timestep.
 */
public void advance(double dt)
{
    for (int i = 0; i < structures.size(); i = i + 1) {
        Structure structure = (Structure) structures.get(i);
        structure.push(dt);
    }
    for (int i = 0; i < masses.size(); i = i + 1) {
        Mass mass = (Mass) masses.get(i);
        mass.move(dt);
    }
}

/**
 * Return the square error in final speed and position.
 */
public double getSuccess()
{
    Vector target = new Vector(0.25, 8.0, 0);
    Vector zero = new Vector(0, 0, 0);
    double distance2 = hand.position.length2(target);
    double speed2 = hand.velocity.length2(zero);
    return distance2 + speed2;
}
```

Fig. 4. Excerpt of the Arm Simulation