DiffGen: A Toolkit for Generating Distributed Protocol Code

Mehwish Nagda
University of Illinois - Urbana Champaign
nagda@uiuc.edu

Christo Frank Devaraj
University of Illinois - Urbana Champaign
devaraj@uiuc.edu

Indranil Gupta
University of Illinois - Urbana Champaign
indy@cs.uiuc.edu

Gul Agha
University of Illinois - Urbana Champaign
agha@cs.uiuc.edu

ABSTRACT

We describe the design and implementation of a toolkit to automatically generate C code for distributed protocols derived from a subclass of differential equations. The toolkit generates compilable C code. The code can be run directly over any socket implementation, as well as within a simulator we have created for debugging and testing. The toolkit also includes functionality for rewriting to generate protocols for higher order systems and a debugging tool to analyze the behavior of the resulting protocol in a user specified computing environment. We expect the toolkit will be useful to researchers and designers of distributed protocols. The advantages would include systematically using natural analogies to design distributed systems, and cutting short the protocol design life cycle.

General Terms

Code Generators, Toolkit Design

1. INTRODUCTION

A subclass of differential equation systems has been shown to be translatable (and equivalent) to useful distributed protocols such as epidemics, endemics, etc [5]. These protocols are probabilistically scalable and reliable. This paper describes a toolkit to automatically generate compilable and runnable C code given an input system of differential equations. Previous toolkits for code generation from high level specifications consist mainly of *Model Driven Architectures* (MDA). However, these toolkits do not allow a simple and systematic way of generating code for the many distributed protocols based on analogies from natural phenomena, for e.g. [9][7][6][2][3][4]. Although these phenomena have been extensively described using differential equations, much of the work of converting from the equations to actual protocols is left to the user. As such, we expect the toolkit will be useful to researchers and designers using natural analogies to design distributed systems by cutting short the protocol design life cycle.

The expected input into the toolkit is from a subclass of systems of differential equations which is then converted into distributed protocol code. The resulting protocols are designed as state machines with probabilistic transitions and actions. In particular, the subclass covered by the toolkit consists of completely partitionable ([5]) polynomial systems of equations. A completely partitionable polynomial system of differential equations is a polynomial system of equations where $\sum_{x \in X} \dot{x} = 0$ (X is the set of all variables) and every term T in an equation has a corresponding $-T$ term in another equation. The protocol is built using a combination of three actions: *flipping*, *one time sampling* and *tokenizing*. The mappings are described in more detail in [5]. However, for the sake of completeness, the protocol actions are summarized below:

- Flipping: A term $-c.x$ in the equation for $\dot{x}$ results in a flipping action. Periodically every node in state x transfers to state y (where $\dot{y}$ has a corresponding $c.x$ term) with a probability proportional to c.
- One Time Sampling: Consider a term $T = -c.x^{i_{x,f_x,T}}.\Pi_{y \in X-\{x\}}.y^{i_{y,f_x,T}}$ in the equation for $\dot{x}$ where X is the set of all variables in the system. If $i_{x,f_x,T} \geq 1$, then this term results in a one time sampling action. Periodically every node in state x samples $i_{x,f_x,T} - 1 + \sum_{y \in X-\{x\}} i_{y,f_x,T}$ random nodes. Consider a sequence S (of variables in X constructed from the term T) where all variables in X are in the sequence a number of times equal to their exponent (except x which has one less) in T. Then the node transfers to state y (where $\dot{y}$ contains the term $+T$) if the states of nodes sampled are in the same order as S and a coin (with a success probability proportional to c) toss succeeds.
- Tokenizing: Note that one time sampling for terms in $\dot{x}$ requires $i_{x,f_x,T} \geq 1$. Tokenizing is used to translate terms of the form $T = -c.\Pi^{y \in X-\{x\}} y^{i_{y,f_x,T}}$ in an equation for $\dot{x}$. Pick a variable $w \in X - \{x\}$ such that $i_{w,f_x,T} \geq 1$ and create appropriate flipping/one time sampling action for nodes in state w. When this flipping/one time sampling action for a node in state w succeeds, instead of changing state, it sends a token to some node in state x. Then the token's recipient changes from state x to y where $\dot{y}$ contains the corresponding $+T$. [5] also deals with constant terms. A constant term is rewritten as the constant times the sum of the state variables (assumed to add up to unity). Then tokenizing is carried out if necessary.

As an example consider the completely partitionable differential equation system $\dot{x} = -\beta xy + \alpha z$, $\dot{y} = \beta xy - \gamma y$, and $\dot{z} = \gamma y - \alpha z$. This system represents an endemic disease such as the common cold. The transformation discussed in [5] will result in a flipping action (corresponding to term $-\alpha z$ in $\dot{z}$) for nodes in state z, a flipping action (corresponding to term $-\gamma y$ in $\dot{y}$) for nodes in state y and a one time sampling action (corresponding to term $-\beta xy$ in $\dot{x}$) for nodes in state x. The mappings are described in more detail in [5].

2. TOOLKIT DESIGN

Toolkits that enable code generation from higher level architectures are already available. Many of them are classified as *Model Driven Architectures* (MDA). Major aspects of MDA include UML-based modeling, transformation between an application's overall design models and the models that are specific to the underlying computing architecture like EJB and generation of code in a specific language. Currently, more than forty such tools are available [1]. Some of these tools like ArcStyler from Interactive Objects Software, GmbH, XDE from IBM Rational, and OptimalJ from Compuware Corp include the UML modeler, the transformation engine and the code generation. Others like Codagen Technology Corps Architect and Telelogics Tau take input from existing UML modeling tools such as Rational Rose or No Magic Incs MagicDraw UML and then generate code from them.

However, these toolkits do not allow a simple and systematic way of generating code for the many distributed protocols based on analogies from natural phenomena, for e.g. [9][7][6][2][3][4]. Although these phenomena have been extensively described using differential equations, much of the work of converting from the equations to actual protocols is left to the user.

The main objective behind this toolkit is to allow input in the form of differential equations and use them to generate scalable, fault-tolerant and reusable protocols with well-defined interfaces. The outputted protocol code is parameterized to allow reuse of the basic flipping, one time sampling and tokenizing functions and also help in easy inclusion of other possible future mappings of differential equation terms to actions. By defining standard interfaces, users can port the protocol to different underlying communication mechanisms, membership protocols, and compose them together to get more complex protocols. Further, the toolkit is designed to be portable and extensible. We would like easy code generation for languages other than C as well as easy integration into other toolkits such as those that compose various protocols together. Currently, we support a subset of differential equations. However, in the future we would like to have the ability to extend the toolkit's functionality to a larger set of equations if necessary. To that end, the input is first converted into an internal representation, *diffIR*. C code is then generated from this internal representation.

diffIR contains a list of differential equations in the system. Each differential equation, $\dot{x_i}$, contains a list of positive and negative terms as well as the differential equation variable, x_i. Each term in *diffIR* contains the constant in the term, the set of variables involved in the term and their exponents, and a reference to the matching negative or positive term in the differential equation system.

The input semantics requires each differential equation in the input system to be separated by a newline. An equation in the system of n equations is in the form $D[x_i, order] = f(x_1, x_2, ..., x_i, ..., x_n)$ where x_i is the variable that is differentiated and *order* is the order of differentiation. An example input equation's right hand side would be in the form $0.3 * x * z - 0.3 * x * y$. Each term in the equation starts with a constant value which may be positive or negative. Variables in the term are expected to be separated by a ' * '. Exponents for each variable follow the ' ^' character. A variable with no exponent defined has an exponent of 1.

Apart from the differential equation system, we require that the user provides application specific functions implementing the communication mechanism between nodes (send and receive) and for choosing a random node from the group. We provide some default functions to the user for common scenarios, including a socket implementation for testing.

The internal representation expects a first order system of equations. However, the toolkit allows higher order systems as input. Higher order system of linear equations are useful and have been used to describe a technique for parallel load balancing based on the heat conduction phenomena[9]. The rewriting module takes higher order equations from the parser (described above) and re-writes them as first order equations by introducing additional variables for each intermediate order into the system as described in [8][5]. The rewriting module then outputs the corresponding *diffIR*.

The toolkit then generates code given the *diffIR* and user-defined functions. The user defined functions are output as-is. The header file and core C file are based on standard functions of flipping, one time sampling and tokenizing that are independent of the differential equation system. Timer events are used to wake nodes up to carry out protocol actions. Each timer event is a structure containing the time in the future when the event should fire and other data pertinent to the particular timer event. Each node sets a timer event for each negative term (including terms from other equations for tokenizing actions) in the differential equation it is simulating currently. For example, a node in state x will set a timer for each negative term in the differential equation for $\dot{x}$. Timer events for different terms in the same equation happen at staggered times within a protocol period. Every timer event is refreshed to happen one protocol period later upon firing.

Code for flipping and one time sampling are written in the form of four C functions (*flipping*, *ots*, *tok_flipping* and *tok_ots*) that are parameterized thus making them fixed between executions. The Java toolkit writes out a function called *schedule_timer_event* which is called whenever a timer event needs to be scheduled. The function is passed a *payload* which contains information about the node requesting the timer event, its state and the particular term to be handled. *schedule_timer_event* after checking if there has not been a change of state in the meantime, calls *flipping*, *ots*, *tok_flipping* or *tok_ots* depending on the kind of term and parameters are passed appropriately. Note that this calling code is written out by the toolkit and the parameters will be known during translation. *flipping* just tosses a coin and decides to change state. *ots* requests states of as many random nodes as required (by calling *get_random_state*). *tok_flipping* and *tok_ots* do the same except that instead of changing state, they send a token.

The messaging subsystem is built out of an interface that has two methods *send_msg* and *receive_msg*. The functions defined in the above paragraph assume an implementation

of this interface. We provide some example implementations of the messaging interface. One implementation meant for the provided simulator simulates the distributed system as a discrete event simulation. The send/receive functions just add and remove a new event on an event queue. We also provide a UDP socket implementation with a well defined server where the nodes register their UDP socket addresses (IP address + port number). The server assigns node addresses as and when nodes join. When a pre-decided number of nodes have joined, the server sends all nodes a *start* message. The nodes start executing protocol actions as described in earlier paragraphs. When a message needs to be sent, *send_msg* checks to see if the destination node's address (socket address) is in the local cache. If so then the message is sent over UDP to the destination socket. If not then the server is contacted for the address (which is cached) and the message is sent. Nodes signal state changes to the server. We use blocking socket receives and hence, the client code spawns a thread for listening to datagrams on the socket and this thread calls *receive_msg* when there is a message to receive. Semaphores are used for mutually exclusive access between threads to shared data (such as node state). The sockets implementation is completely asynchronous and provides a very valuable technique to build and analyze distributed protocols from differential equations. Moreover by implementing different *send_msg* and *receive_msg* functions, the code can be easily ported to any other communication system (such as radio communication using *motes* in wireless sensor networks).

The toolkit comes packaged in a GUI with functionality to edit, view and create new files, generate code and analyze the outputted protocol. The toolkit allows users to generate code both for deployment as well as testing and analyzing on a provided simulator. The simulator allows users to specify environment parameters such as the number of processes that crash-stop or crash-recover, clock drifts, percentage of messages dropped, and the number of nodes to simulate on. The simulator is based on a discrete event simulation model and assumes a FIFO message queue.

3. CONCLUSION

It has been shown in [5] that a subclass of differential equation systems can be translated into equivalent probabilistic distributed protocols. This transformation is done by converting negative terms in the equations into one of two protocol actions (*flipping* and *one time sampling*) augmented by *tokenizing* if necessary. We have implemented a toolkit in Java that can take a system of differential equations (completely partitionable and polynomial) and obtain the corresponding protocol code in C. The toolkit can be parameterized to obtain protocol implementation to run over sockets or over a simple C simulator. The toolkit can also rewrite arbitrary differential equations in the form required by the algorithm in [5]. A wide range of natural systems whose behavior can be specified as differential equations can be imitated on a distributed system using the proposed toolkit. Our implementation has been tested for behaviors such as probabilistic consensus using the Lotka-Voltera model of competition.

4. REFERENCES

[1] J. Ambrosio. Tools for the code generation. *http://www.adtmag.com/article.asp?id=7850*, 2004.

[2] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency*, 1999.

[3] A. Das, I. Gupta, and A. Motivala. Swim: Scalable weaklyconsistent infection-style process group membership protocol. *Proc. of The International Conference on Dependable Systems and Networks (DSN 02)*, pages 303–312, June 2002.

[4] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. *Proceedings of the Sixth Symposium on Principles of Distributed Computing*, pages 1–12, August 1987.

[5] I. Gupta. On the design of distributed protocols from differential equations. *Proceedings of PODC, Newfoundland, Canada*, 2004.

[6] I. Gupta, A. Kermarrec, and A. Ganesh. Efficient epidemic-style protocols for reliable and scalable multicast. *21st Symposium on Reliable Distributed Systems (SRDS 2002)*, pages 180–189, October 2002.

[7] D. Oppen and Y. Dalal. The clearinghouse: A decentralized agent for locating named objects in a distributed environment. Technical report, Xerox Corp, 1981.

[8] S. Strogatz. *Nonlinear Dynamics and Chaos*. Perseus Books Publishing Company, Cambridge, Massachusetts, 2000.

[9] E. W. Weisstein. Heat conduction equation. *MathWorld–A Wolfram Web Resource, http://mathworld.wolfram.com/HeatConductionEquation.html*, 2004.