NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
NWU-EECS-10-01
February 12, 2010

GAIL: A Design and Implementation of a Constrained Guarded Action Intermediate Language Suitable for Rewrite-Based Optimization

Tim Zwiebel

Abstract

Deploying wireless sensor networks can present many technical challenges. The ABSYNTH Project attempts to address these challenges by presenting the user with an end-to-end system to assist the user. GAIL (Guarded Action Intermediate Language) is an intermediate language for expressing node-level code.

GAIL programs consist of a series of guarded action statements. These statements have a boolean expression, the guard, which triggers its actions when it is true. Programs in GAIL contain descriptions of both the hardware and software. GAIL is designed to be constrained in several ways. First, each guarded action has a constraint to determine when the guards should be evaluated. Second, all variables are statically allocated. Finally, evaluation of the guarded actions is designed to limit where expressions with side effects are allowed to occur. These constraints cause the language to not be Turing-complete. Instead, GAIL programs can be modeled as finite state machines. This property of the language facilitates program analysis at compile time.

GAIL programs are optimized using rewrite rules by rewriting programs into many equivalent programs. Because programs contain both hardware and software descriptions, the optimization process can optimize both hardware and software at the same time. Due to the constrained nature of the language, these rewrite rules are very compact, often taking only a few lines of code.

By using a constrained guarded action language, code for sensor networks can be optimized using very concise rewrite rules. Also, because the language is not Turing-complete, it is possible to statically analyze programs completely in GAIL. This analysis can help determine an efficient equivalent program produced by the rewrite rules at compile time.

This project was supported by the National Science Foundation (NSF) via grant CNS-0721978

Keywords: sensor networks, guarded actions, intermediate language, rewrite optimization

GAIL: A Design and Implementation of a Constrained Guarded Action Intermediate Language Suitable for Rewrite-Based Optimization

Tim Zwiebel

School of Engineering
Northwestern University
Evanston, IL 60201

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

This project is made possible by support from the
National Science Foundation (NSF) via grant CNS-0721978

Keywords: sensor networks, guarded actions, intermediate language, rewrite optimization

Abstract

Deploying wireless sensor networks can present many technical challenges. The ABSYNTH Project attempts to address these challenges by presenting the user with an end-to-end system to assist the user. GAIL (Guarded Action Intermediate Language) is an intermediate language for expressing node-level code.

GAIL programs consist of a series of guarded action statements. These statements have a boolean expression, the guard, which triggers its actions when it is true. Programs in GAIL contain descriptions of both the hardware and software. GAIL is designed to be constrained in several ways. First, each guarded action has a constraint to determine when the guards should be evaluated. Second, all variables are statically allocated. Finally, evaluation of the guarded actions is designed to limit where expressions with side effects are allowed to occur. These constraints cause the language to not be Turing-complete. Instead, GAIL programs can be modeled as finite state machines. This property of the language facilitates program analysis at compile time.

GAIL programs are optimized using rewrite rules by rewriting programs into many equivalent programs. Because programs contain both hardware and software descriptions, the optimization process can optimize both hardware and software at the same time. Due to the constrained nature of the language, these rewrite rules are very compact, often taking only a few lines of code.

By using a constrained guarded action language, code for sensor networks can be optimized using very concise rewrite rules. Also, because the language is not Turing-complete, it is possible to statically analyze programs completely in GAIL. This analysis can help determine an efficient equivalent program produced by the rewrite rules at compile time.

Contents

List of Figures

Chapter 1

Introduction

Wireless sensor networks (WSNs) are a growing technology with vast potential. They consist of small hardware devices, called nodes, that are spread out over an area that is to be monitored. A typical node has a microprocessor, various sensors, a battery, and a radio.

Deploying a wireless sensor network can present many technical challenges. Typically, these systems are programmed at a low level, usually in C or assembly. They often have limited energy resources, such as batteries, making power and energy consumption a major priority. Frequently, they use complex networking protocols to communicate. Due to these challenges, embedded systems experts are often needed to deploy wireless sensor networks.

1.1 The ABSYNTH Project

The goal of the ABSYNTH (Archetype-Based SYNTHesis) Project is to respond to this problem by making it easier to design and implement wireless sensor network applications. The end-to-end system being designed will attempt to reach this goal by assisting users while they design their wireless sensor network.

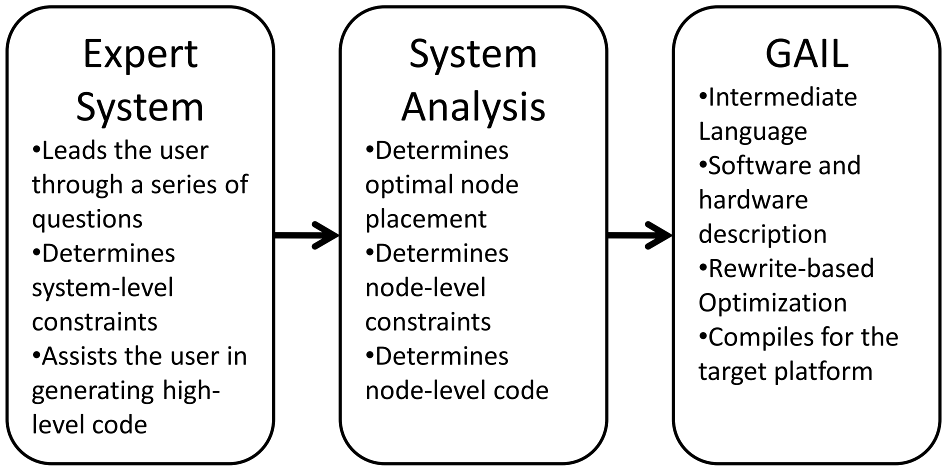

Expert System
•Leads the user through a series of questions
•Determines system-level constraints
•Assists the user in generating high-level code
System Analysis
•Determines optimal node placement
•Determines node-level constraints
•Determines node-level code
GAIL
•Intermediate Language
•Software and hardware description
•Rewrite-based Optimization
•Compiles for the target platform

Figure 1.1: ABSYNTH Project

As seen in Figure 1.1, the user will start with an expert system. The expert system will lead the user through a series of questions that helps the system determine the type of application and the restrictions on the application, such as price, size, and lifetime. It will help the user choose hardware for the application, as well as provide the code for simple applications, or present the user with an easy programming language that is well-suited for the application. At this point, analyses will be run to determine things such as optimal node placement and the node-level constraints required to meet the system-level constraints.

1.2 GAIL

After the system-level code is determined and analyses are performed, the system-level code can be compiled into node-level code written in GAIL (Guarded Action Intermediate Language). The focus of this document is the design and implementation of GAIL. GAIL is an intermediate language based on a guarded action model. In this model, a boolean expression, called a guard, triggers a sets of actions when the guard is true. The guarded action model is similar to Dijkstra's guarded

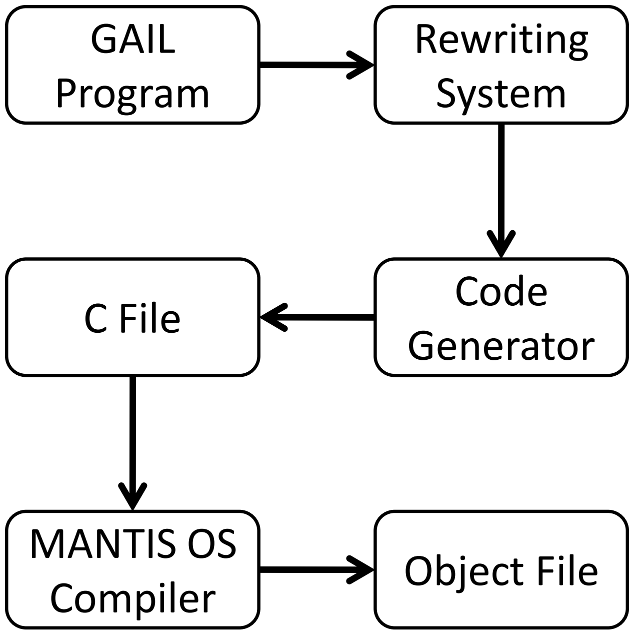


Figure 1.2: GAIL Compilation Process

command language. GAIL is constrained by design in several ways. First, each guarded action has a constraint to determine when the guards should be evaluated. Second, all variables are statically allocated. Finally, evaluation of the guarded actions is designed to limit where expressions with side effects are allowed to occur. These constraints make GAIL programs not Turing-complete and allow them to be modeled as finite state machines, which allows extensive program analysis at compile time. The syntax and semantics of GAIL are described in more detail in Chapter 2.

Figure 1.2 shows the compilation process for GAIL programs. Programs are optimized using the rewriting system. Programs in GAIL describe both the software and hardware used in the application. This means that during the rewriting process, both the hardware and software can be optimized together. The rewrite rules are simple, allowing new optimizations to be easily added to the existing set. Many equivalent programs are produced, and then an objective function is used to determine the one that is best-suited for the application. Next, the code generator compiles

GAIL code into C code, and from there it is compiled for the target platform. Chapter 3 details the rewriting system.

In the current implementation, after an equivalent program is chosen from the rewriting system, it is compiled to C code using the code generator. From there, the Mantis OS compiler compiles the C code to an object file that can be loaded directly onto the hardware. Chapter 4 explains the code generator.

In order to test the concept of GAIL, simpler versions of the rewrite system and the code generator were created. A simple user interface was also created so that others can more easily use GAIL (see Chapter 5). In the current implementation of GAIL, the data types, the communication, and the hardware descriptions were kept simple. The main focus has been on applications that perform simple periodic sampling. Also, the current set of optimizations is very small. To allow more interesting and complex optimizations and analyses, the compilation process will likely be modified so that the equivalent programs generated by the rewriting system are all compiled to object files which can then be used in the analyses. These issues are discussed in further detail in Chapter 6.

Even though the current implementation is a simpler version of the language, GAIL does demonstrate the advantages of being used as an intermediate language. The rewrite system offers the ability to perform joint hardware-software optimization and to automatically explore the trade-offs of calculating expressions in different ways. In addition, the rewrite rules are simple to create, allowing further expansion of the possible optimizations. GAIL programs can be easily analyzed to determine many properties of the program at compile time, which can then be used to optimize the program and make node-level cost estimates.

1.3 Related Work

There is a wide variety of programming languages designed for sensor networks. One such language, SensorScheme, is able to create very compact programs [5, 6]. By using a Scheme-like syntax, GAIL is easy to parse, easy to rewrite subexpressions, and easy to use with existing tools such as those found in PLT Scheme.

Another language, DESAL$^{\alpha}$, is also based on a guarded action model [2]. Like SensorScheme, DESAL$^{\alpha}$ is intended to be used by programmers directly as opposed to GAIL, which is designed as an intermediate language. DESAL$^{\alpha}$ focuses on some of the challenges in wireless sensor networks such as synchronizing periodic events and implementing neighborhood management services like node discovery and health monitoring. Communication in DESAL$^{\alpha}$ is achieved via distributed state sharing in which programs use shared variables within their networks.

Chapter 2

GAIL Syntax and Semantics

GAIL is an intermediate language based on a guarded action model and designed for use in wireless sensor networks. The language is constrained by design in order to allow programs to be analyzed in ways that might not otherwise be possible. It is also designed to facilitate rewrite-based optimization. Programs in GAIL consist of a set of definitions, followed by a set of guarded action expressions.

2.1 Guarded Action Model

In a guarded action model, all statements with side effects are associated with boolean expressions called guards. Instead of evaluating code in the order in which it appears, the statements are evaluated whenever its guard becomes true. Because of this, this type of model is best suited for applications which are event-driven. When events occur, or conditions become met, the actions are taken.

Type	**Description**	**Current Implementation**
boolean	`true` or `false`	`uint8_t`
scalar	a real number	`float`
boolean queue	a collection of booleans	array
scalar queue	a collection of scalars	array
analog input	an analog input signal	hardware
digital input	a digital input signal	hardware
analog output	an analog output signal	hardware
digital output	a digital output signal	hardware

Figure 2.1: Types in GAIL

2.2 Definitions

The first part of GAIL programs are the definitions. The definitions describe the variables as well as the hardware that is used in the program. All variables are given initial values when they are defined. There are eight data types in GAIL. These types are listed in Figure 2.1.

In order to facilitate analyzing memory usage, queues are statically allocated, and therefore can become full. Each queue has a policy associated with it to determine the action that occurs when an item is added to a queue that is full. If the policy is `DISPLACE`, then an existing item is displaced in favor of the new item, but if the policy is `DROP`, the new item will not be added and the queue will remain unchanged. Listed below are the definitions that are possible in GAIL.

`(define-bv name ival)`
Defines a variable for a boolean value.
`name` - the name of the variable, starting with "bv_".
`ival` - the initial value (a boolean).

`(define-sv name ival)`
Defines a variable for a scalar value.
`name` - the name of the variable, starting with "sv_".
`ival` - the initial value (a scalar).

`(define-bq name capacity ival policy)`
Defines a variable for a boolean queue. Queues use zero-based indexing.
`name` - the name of the variable, starting with “bq_”.
`capacity` - the maximum size of the queue.
`ival` - the initial value (a boolean queue).
`policy` - either `DISPLACE` or `DROP`.

`(define-sq name capacity ival policy)`
Defines a variable for a scalar queue. Queues use zero-based indexing.
`name` - the name of the variable, starting with “sq_”.
`capacity` - the maximum size of the queue.
`ival` - the initial value (a scalar queue).
`policy` - either `DISPLACE` or `DROP`.

`(define-ai hw func)`
Defines a block of hardware with an analog signal to be read as an input.
`hw` - the description of the hardware.
`func` - a string that is the name of a C function that will return the value as a scalar.
Example: `"getLight"` refers to `scalar_t getLight()`.

`(define-di hw func)`
Defines a block of hardware with a digital signal to be read as an input.
`hw` - the description of the hardware.
`func` - a string that is the name of a C function that will return the value as a scalar.
Example: `"getLight"` refers to `scalar_t getLight()`.

`(define-ao hw func)`
Defines a block of hardware with an analog signal to be written to as an output.
`hw` - the description of the hardware.
`func` - a string that is the name of a C function that will write a scalar value.
Example: `"setLEDs"` refers to `void setLEDs(scalar_t value)`.

`(define-do hw func)`
Defines a block of hardware with a digital signal to be written to as an output.
`hw` - the description of the hardware.
`func` - a string that is the name of a C function that will write a scalar value.
Example: `"setLEDs"` refers to `void setLEDs(scalar_t value)`.

2.3 Guarded Actions

The second part of GAIL programs is a set of guarded actions. Guarded actions contain a guard, a set of actions, and a constraint. The constraint determines when the guard is evaluated. The set of actions is evaluated whenever the guard evaluates to `true`. Below is the definition for guarded actions.

`(guarded-action name constraint guard (action ...))`

`name` - the name of the guarded action. It is a string surrounded by quotes that matches "[a-zA-Z][0-9a-zA-Z_]*".

`constraint` - tells when the guard should be evaluated.

`guard` - a boolean expression that triggers the actions when `true`.

`(action ...)` - a list of one or more actions. An action is a expression with side effects. The side effects can only be at the top level of actions. When a guard evaluates to `true`, all of the expressions without side effects in the actions are calculated, then the expressions with side effects are evaluated (in order).

2.3.1 Guards

Guards in GAIL are boolean expressions. They also must be free of side effects. This means that the state of the program can only be changed when a guard is true and the actions occur. Boolean expressions can be `true` or `false`, the name of a boolean variable, or an expression that returns a boolean.

2.3.2 Constraints

In GAIL, guarded action expressions contain a constraint that describes when the guard is to be evaluated. By doing this, it is ensured that the guards are only evaluated when necessary. The most simple constraint is the `time-constraint`, which evaluates the guard periodically and is well-suited for periodic sampling. Below is a full list of all of the constraints in GAIL.

(time-constraint offset period)
A constraint that the guard be evaluated periodically.
`offset` - the offset in the cycle in milliseconds.
`period` - the total period of the cycle in milliseconds.
Example: `(time-constraint 15 100)` would result in the guard being evaluated at time t = 15ms, 115ms, 215ms...

(time-constraint offset period refill)
Not yet implemented.
A constraint that the guard be evaluated periodically when refill is true. When refill is false, the guard will not be evaluated.
`offset` - the offset in the cycle in milliseconds.
`period` - the total period of the cycle in milliseconds.
`refill` - a boolean variable. The guard will only be evaluated periodically when refill is true. This is used to register and unregister periodic constraints, so that the node can enter a low power mode when it is known that the guarded action does not need to be evaluated.

(continuous-constraint)
Evaluates the guard whenever possible. This will likely result in inefficient code, but may produce useful hardware optimizations during the rewrite phase of compiling.

(variable-constraint name)
Not yet implemented.
A constraint that the guard be evaluated when a variable changes.
`name` - the name of the variable.
Example: `(variable-constraint sv_temp)` would result in the guard being evaluated every time an action is performed on `sv_temp`.

(guard-constraint name)
A constraint that this guard be evaluated after the target guard is evaluated to `true`. This guard will not be evaluated until after the target guarded action's set of actions is finished being evaluated.
`name` - the name of the target guarded action.
Example: `(guard-constraint "sense_and_send")` would result in the guard being evaluated every time `sense_and_send`'s guard is true.

(message-sent)
Not yet implemented.
A constraint that the guard be evaluated when a message is sent.

(message-received)
A constraint that the guard be evaluated when a message is received.

`(message-deleted)`
Not yet implemented.
A constraint that the guard be evaluated when a message is deleted.

`(interrupt-constraint)`
Not yet implemented.
A constraint that the guard be evaluated when an interrupt is triggered.

2.3.3 Actions

As previously mentioned, actions are the only place in a GAIL program where the state of the program can be changed. All side effects must occur inside of actions. Furthermore, side effects must be at the top level of actions. This means that when each action is evaluated, exactly one change is made, and it is the last thing to be done. Additional rules regarding the evaluation of guarded actions are discussed in Section 2.4. Listed below are the actions available in GAIL (not including the actions used for communication).

`(set var val)`
Sets the value of a variable.
`var` - the name of the variable.
`val` - the value to which var will be set.

`(set queue i val)`
Sets the ith item in the queue to `val`.
`queue` - the name of the queue.
`i` - the index in the queue.
`val` - the value to which the item will be set.

`(set-queue queue val)`
Sets the value of a queue.
`queue` - the name of the queue.
`val` - the value to which `queue` will be set.

`(set-range dst-queue src-queue i)`
Copies a queue to another part of another queue.
`dst-queue` - the destination queue.
`src-queue` - is the source queue.
`i` - the index in `dst-queue` where the copy operation should start.
Example:

```
   (define-sq sq_dst 5 (0 0 0 0 0))
   (define-sq sq_src 2 (1 2))
   (set-range sq_dst sq_src 2) ⇒ sq_dst=(0 0 1 2 0)
```

`(set-policy queue pol)`
Sets the policy of the queue.
`queue` - the queue whose policy is to be changed.
`pol` - the policy (either `DROP` or `DISPLACE`).

`(push-front queue val)`
Adds an item to the beginning of a queue.
`queue` - the queue.
`val` - the value to add.

`(pop-front queue)`
Removes an item from the beginning of a queue.
`queue` - the queue.

`(push-back queue val)`
Adds an item to the end of a queue.
`queue` - the queue.
`val` - the value to add.

`(pop-back queue)`
Removes an item from the end of a queue.
`queue` - the queue.

`(write output s-exp)`
Writes a scalar expression to an analog or digital output.
`output` - the analog or digital output expression.
`s-exp` - the scalar expression.

2.3.4 Communication

Communication in GAIL is achieved with special constraints, expressions, and actions. In the current implementation, communication involves sending and receiving messages using a single-hop unicast model. This means that only nodes in range of the sender will be able to receive messages. All messages sent or received over different interfaces, such as the radio or serial port, are considered to be the same type. All messages have a source, a destination, a tag, an interface, and data. The source is the node ID of the sender while the destination is the node ID of the targeted recipient. Node IDs are stored in non-volatile flash memory prior to loading and executing GAIL programs. The tag is a scalar expression associated with the message, often used in GAIL programs to specify the type of message.

When messages are received, regardless of the interface through which they were received, they are stored in a special message queue. While there are available messages, the first message in the queue is chosen to be the current message. All expressions then act upon the current message. This means that no additional messages can be handled until the ones before it have been dealt with. Below are the expressions in GAIL associated with communication.

`(send interface dst tag (item ...))`
Sends a message.
`interface` - the interface over which to send the message (e.g. SERIAL, RADIO, LOOPBACK, etc.). The interface is a scalar expression and is dependent on the target platform. A macro is usually used to determine the appropriate scalar for the desired interface.
`dst` - the destination of the message.
`tag` - a scalar expression to identify the message.
`(item ...)` - a list of zero or more booleans, scalars, and/or queues.

`(get-msg-tag)`
Returns the tag of the current message.

`(get-msg-b i)`
Returns a boolean from the i^{th} item in the current message.
`i` - the index in the message.

`(get-msg-s i)`
Returns a scalar from the i^{th} item in the current message.
`i` - the index in the message.

`(get-msg-bq i)`
Returns a boolean queue from the i^{th} item in the current message.
`i` - the index in the message.

`(get-msg-sq i)`
Returns a scalar queue from the i^{th} item in the current message.
`i` - the index in the message.

`(delete-message)`
Deletes the current message, permanently destroying it.

There are two reserved words for expressing the destination of a message. They are listed below.

`ID`
The node id.

`ALL`
A special destination, that corresponds to broadcasting to all nodes in the network. Since the current implementation only supports single-hop communication, this destination corresponds to all nodes in range of the radio.

2.3.5 Expressions Without Side Effects

Logical Expressions

The following expressions return booleans.

`(and b1 b2)`
Returns the logical and of two boolean expressions.
`b1` and `b2` - boolean expressions.

`(or b1 b2)`
Returns the logical or of two boolean expressions.
`b1` and `b2` - boolean expressions.

`(xor b1 b2)`
Returns the logical xor of two boolean expressions.
`b1` and `b2` - boolean expressions.

`(not b1)`
Negates a boolean expression.
`b1` - a boolean expressions.

Scalar Expressions

The following expressions return scalars.

`(+ s1 s2)`
Adds two scalar expressions.
`s1` and `s2` - scalar expressions.

`(- s1 s2)`
Subtracts `s2` from `s1`.
`s1` and `s2` - scalar expressions.

`(* s1 s2)`
Multiplies two scalar expressions.
`s1` and `s2` - scalar expressions.

`(/ s1 s2)`
Divides `s2` into `s1`.
`s1` and `s2` - scalar expressions.

`(abs s1)`
The absolute value of a scalar expression.
`s1` - a scalar expression.

`(> s1 s2)`
Tests if `s1` is greater than `s2`. Returns a boolean.
`s1` and `s2` - scalar expressions.

`(>= s1 s2)`
Tests if `s1` is greater than or equal to `s2`. Returns a boolean.
`s1` and `s2` - scalar expressions.

`(< s1 s2)`
Tests if `s1` is less than `s2`. Returns a boolean.
`s1` and `s2` - scalar expressions.

`(<= s1 s2)`
Tests if `s1` is less than or equal to `s2`. Returns a boolean.
`s1` and `s2` - scalar expressions.

`(= s1 s2)`
Tests if `s1` and `s2` are the same. Returns a boolean.
`s1` and `s2` - scalar expressions.

`(!= s1 s2)`
Tests if `s1` and `s2` are different. Returns a boolean.
`s1` and `s2` - scalar expressions.

Queue Expressions

The following are expressions that apply to queues.

`(head queue)`
Returns the item at the front of the queue.
`queue` - the queue.

`(tail queue)`
Returns the item at the back of the queue.
`queue` - the queue.

`(get queue i)`
Returns the item at the ith position in the queue.
`queue` - the queue.
`i` - the index of the item. (zero-based indexing)

`(get-range queue i j)`
Returns a queue containing the ith through the jth items in the original queue.
`queue` - the queue.
`i` - the starting index. (zero-based indexing)
`j` - the ending index. (zero-based indexing)
Example: `(get-range (1 2 3) 0 1)` returns `(1 2)`

`(empty? queue)`
Returns `true` if the queue is empty and `false` otherwise.
`queue` - the queue.

`(size queue)`
Returns the number of items in the queue.
`queue` - the queue.

`(max s-queue)`
Gets the maximum value of a scalar queue.
`queue` - the queue.

`(min s-queue)`
Gets the minimum value of a scalar queue.
`queue` - the queue.

If Expressions

Conditionals are supported in GAIL through guards and through `if` expressions.

`(if boolean tb fb)`
If `boolean` is `true`, returns `tb`, otherwise returns `fb`. `tb` and `fb` are either boolean expressions, scalar expressions, or queue expressions, and they most both be the same type. Either `tb` or `fb` can be `()`, but not both.
`boolean` - the test.
`tb` - the true branch.
`fb` - the false branch.

Input Expressions

The following expression allows reading scalar values from hardware.

`(read input)`
Reads an analog or digital input and returns a scalar.
`input` - the analog or digital input expression.

Hardware Expressions

The following are expressions that describe hardware. These expressions are never evaluated in the software. They represent either analog or digital signals. The following expressions were created as a proof of concept. The entire set of hardware operations has not yet been determined.

`(a+ a1 a2)`
Add `a1` and `a2` with analog hardware.
`a1` and `a2` - analog input expressions.

`(a- a1 a2)`
Subtract `a2` from `a1` with analog hardware.
`a1` and `a2` - analog input expressions.

`(a* a1 a2)`
Multiply `a1` and `a2` with analog hardware.
`a1` and `a2` - analog input expressions.

`(d+ d1 d2)`
Add `d1` and `d2` with digital hardware.
`d1` and `d2` - digital input expressions.

`(d- d1 d2)`
Subtract `d2` from `d1` with digital hardware.
`d1` and `d2` - digital input expressions.

`(d* d1 d2)`
Multiply `d1` and `d2` with digital hardware.
`d1` and `d2` - digital input expressions.

Analog/Digital Conversion Expressions

The following are expressions for converting analog signals to digital ones and vice versa.

`(adc a-exp)`
Converts an analog expression to a digital expression using an analog-to-digital converter.
`a-exp` - the analog expression.

`(dac d-exp)`
Converts a digital expression to an analog expression using a digital-to-analog converter.
`d-exp` - the digital expression.

2.4 Evaluation

The evaluation of guarded actions is constrained in a very specific way to facilitate the analysis of GAIL programs. The following rules define the evaluation process.

1. Guards are evaluated according to their constraints. Internally, they are placed in a queue and evaluated first come, first served.
2. When a guard is evaluated to `true`, the entire set of actions is evaluated immediately. No other guards are evaluated until the actions have been evaluated.
3. Hardware is sampled once for the entire set of actions, so setting two different variables to the current value of a sensor will guarantee that both variables are equal.
4. Expressions with side effects must be at the top level of an action.
5. When evaluating a set of actions, all expressions that do not have side effects are evaluated before those that do.
6. Expressions that have side effects are evaluated in order.

Because of these rules, the order in which the actions are evaluated does not matter, except in the case where multiple actions change the same variable. Furthermore, because there is no form of explicit iteration or recursion, and because all variables are allocated statically, a program can be thought of as a finite state machine. This property makes it possible to analyze GAIL programs at compile time. These analyses are discussed in more detail in Section 3.4.

2.5 Examples

The following are examples of GAIL programs that demonstrate the language and its usage.

2.5.1 Task 1 Example

The Task 1 example was created for the ABSYNTH Project for use in a study to determine the difficulty of programming a wireless sensor network using various languages [1]. The tasks are intended to be similar to real deployed sensor network applications.

Task 1: Sample light and temperature every 2 seconds from all the nodes in the network. Transmit the samples with their node identification numbers to the base station.

```
((MICAZ_SENSORS)
 ((guarded-action "sense_and_send"
                  (time-constraint 0 2000)
                  true
                  ((send RADIO 0 0 (ID
                                    (read (adc ai_light))
                                    (read (adc ai_temp))))))))
```

2.5.2 Task 3 Example

The Task 3 example is from the same study as the Task 1 example in the previous section.

Task 3: Sample temperature every 2 seconds from all the nodes in the network. Transmit the node identification numbers and the most recent temperature readings from nodes where the current temperature exceeds 1.1 times the maximum temperature reading during the preceding 10 seconds.

```
((MICAZ_SENSORS
  (define-sq sq_samples 6 (0 0 0 0 0) DISPLACE))
 ((guarded-action "sense"
                  (time-constraint 0 2000)
                  true
                  ((push-front sq_samples (read (adc ai_temp)))))
  (guarded-action "send"
                  (guard-constraint "sense")
                  (> (head sq_samples)
                     (* 1.1 (max (get-range sq_samples 1 5))))
                  ((send RADIO 0 0 (ID
                                    (head sq_samples)))))))
```

2.5.3 Lights Example

The following is an example program that both reads inputs and writes to outputs. Every three seconds, it reads a light sensor and adds 15 to the raw data value, it sends the previous data sample to the root node (node 0), and it uses the LEDs to count in binary up to seven before looping back to zero. According to rule 5 in the Section 2.4, the value for `sv_light` that will be sent in the last action is calculated before `sv_light` is modified in the first action. This means that each time a message is sent, it is actually the previous data sample that is being sent. If the goal were to send the current data sample, the entire expression that calculates the new `sv_light` would be used instead. Because of rule 3, hardware is only sampled once, so any values that use this expression are guaranteed to be the same.

```
((MICAZ_SENSORS
  (define-sv sv_light 0)
  (define-sv sv_leds 0))
 ((guarded-action "test"
                  (time-constraint 0 3000)
                  true
                  ((set sv_light (+ (read (adc ai_light)) 15))
                   (set sv_leds (if (> sv_leds 7) 0 (+ sv_leds 1)))
                   (write do_leds sv_leds)
                   (send RADIO 0 0 (sv_light))))))
```

2.5.4 ACM Example

The following example is a simplified version of an autonomous crack monitoring (ACM) application [3, 4]. In the example, a light sensor is used in place of a crack-measuring sensor. Each node starts in “state 0”, where the node is waiting for the sensor to exceed a threshold value of 200. As soon as a node detects a value above 200, it stores the value, switches to “state 1”, and sends a message to all nodes in the network to also switch to “state 1”. Once in “state 1”, the node collects samples for three seconds at ten hertz. After it has collected the 30 samples (31 for the node that triggered the event), the nodes return to “state 0”.

```
((MICAZ_SENSORS
  (define-sq sq_samples 31 () DISPLACE)
  (define-sv sv_light 0)
  (define-sv sv_state 0)
  (define-sv sv_count 0))
 ((guarded-action "poll"
                  (continuous-constraint)
                  (and (= sv_state 0) (> (read (adc ai_light)) 200))
                  ((set sv_state 1)
                   (set sv_count 0)
                   (push-back sq_samples (read (adc ai_light)))
                   (send RADIO ALL 1 ())))
  (guarded-action "sample"
                  (time-constraint 0 100)
                  (= sv_state 1)
                  ((push-back sq_samples (read (adc ai_light)))
                   (set sv_count (+ sv_count 1))))
  (guarded-action "stop_sampling"
                  (guard-constraint "sample")
                  (> sv_count 30)
                  ((set sv_state 0)
                   (send RADIO 0 2 (sq_samples))
                   (set-queue sq_samples ())))
  (guarded-action "remote_trigger"
                  (message-received)
                  (= (get-msg-tag) 1)
                  ((set sv_state 1)
                   (delete-message)))))
```

Chapter 3

Rewrite-Based Optimization

As previously mentioned, GAIL is designed with rewrite-based optimization in mind. A tool called PLT Redex, included in PLT Scheme, is used to perform joint hardware-software optimization through the use of rewrite rules. In PLT Redex, these rewrite rules are called reduction cases and a set of reduction cases is called a reduction relation.

3.1 PLT Redex Example

Figure 3.1 shows a simple example of a language grammar, similar in style to BNF (Backus-Naur Form). Expressions in the example language, called simple-lang, are either numbers, or expressions composed of the addition or subtraction of other expressions. The nonterminal, `C`, represents a context and matches any expression in simple-lang that has the word `hole` replacing some part of the expression. `C` will be important later when applying reduction relations because it will define where rewrites are allowed to take place.

Reduction cases consist of a pattern to match, and a term with which to replace the pattern. In this way, optimizations can be made by identifying inefficient expressions, and replacing them

```
(define-language simple-lang
  (exp number
       (+ exp exp)
       (- exp exp))
  (C hole
     (any ... C any ...)))
```

Figure 3.1: Example of a Language Definition

```
(define rewrite
  (reduction-relation
   simple-lang
   (--> (in-hole C (+ number_1 number_2))
        (in-hole C ,(+ (term number_1) (term number_2)))
        "addition")
   (--> (in-hole C (- number_1 number_2))
        (in-hole C ,(- (term number_1) (term number_2)))
        "subtraction")))
```

Figure 3.2: Example of a Reduction Relation

with optimized versions of the expressions. Figure 3.2 shows an example of a reduction relation for simple-lang with two reduction cases. These example reduction cases allow the addition or subtraction of two numbers to be rewritten as a single number which is the answer. In these cases, `number_1` and `number_2` are numbers, and are not necessarily distinct. As previously mentioned, `C` matches any expression in simple-lang that has the word `hole` replacing some part of the expression, so `(in-hole C (+ number_1 number_2))` matches a `C` where the word `hole` replaces an expression that matches `(+ number_1 number_2)`. The comma in the replacement terms indicate that the following expression is to be evaluated as a Scheme expression, which in this example means that the + or − procedure will be called, and the result will be placed in the `hole`. The `term` keyword is similar to a quote in Scheme, in that it prevents its argument from being evaluated. In the example, it uses the numbers bound when the pattern was matched

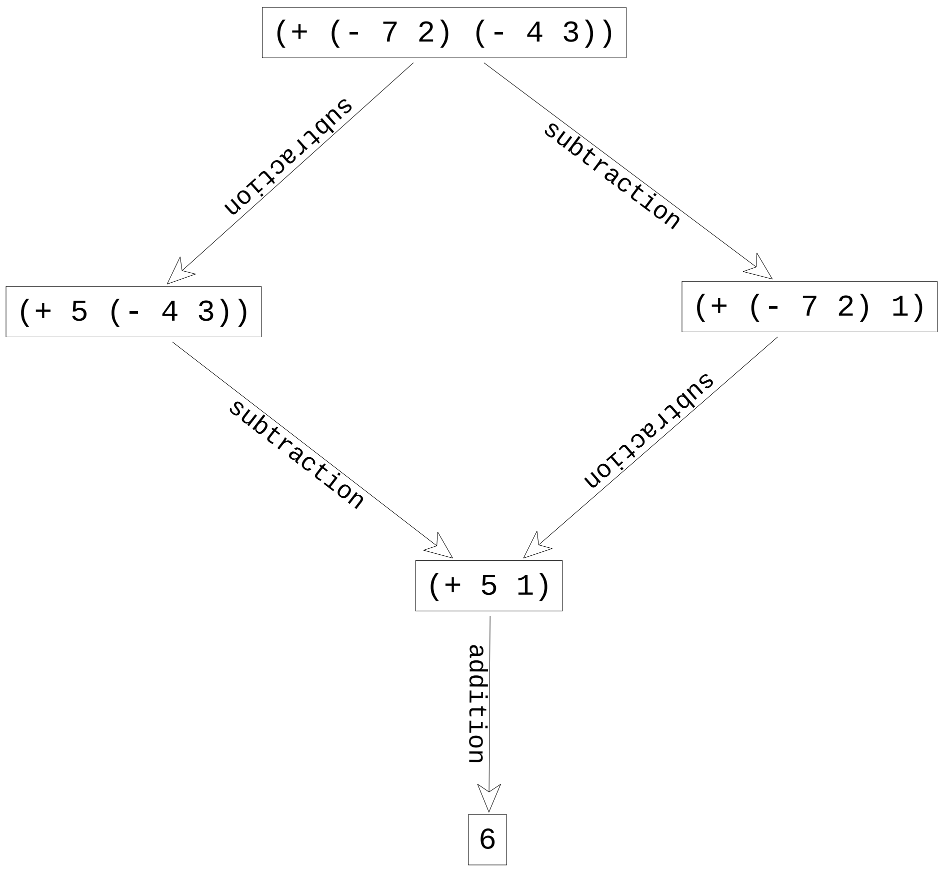


Figure 3.3: Example Graph Using `traces`

rather than looking for global variables. Reduction cases can also have a name and/or side conditions that must hold true for the reduction to be applied. For example, if simple-lang was expanded to include division, the rewrite rule might have a side condition that the second number wasn't zero to avoid a divide-by-zero error.

Finally, a useful tool to test the language and its rewrite rules is the `traces` command, which produces a graph of all possible reductions using the given reduction relation until no new expressions can be produced. For example, `(traces rewrite (term (+ (- 7 2) (- 4 3))))` results in the graph shown in Figure 3.3. In this example, the subtraction expressions can be replaced with their results and then the addition can be simplified.

3.2 Joint Hardware-Software Optimization

The use of rewrite rules combined with the inclusion of hardware description in GAIL makes joint hardware-software optimization possible. By simply adding rewrite rules, software and hardware can be easily interchanged. Rewrite rules can also be used to change software into equivalent software, or hardware into equivalent hardware. The overall purpose of the rewriting system is to produce many equivalent programs and use the objective function to analyze their differences to determine the one best suited for the application.

One of the major advantages to using rewrite-based optimization is the size and simplicity of the optimizations. New optimizations can be added by creating new rewrite rules, which can be written in just a few lines of code.

3.2.1 GAIL Grammar and Rewrite Rules

Below is the formal grammar used to specify GAIL in PLT Redex.

```
(define-language gail
  (prog ((var-def ...)
         (ga ga ...)))

  (var-def (define-bv b-var boolean)
           (define-bq bq-var number b-queue policy)
           (define-bq bq-var number () policy)
           (define-sv s-var scalar)
           (define-sq sq-var number s-queue policy)
           (define-sq sq-var number () policy)
           (define-ai ai-exp string)
           (define-di di-exp string)
           (define-ao ao-exp string)
           (define-do do-exp string))

  (policy DISPLACE DROP)

  (ga (guarded-action string
                      constraint
                      b-exp
```

```
                    (action action ...)))

  (constraint (time-constraint number number)
              (time-constraint number number b-var)
              (continuous-constraint)
              (variable-constraint d-var)
              (guard-constraint string)
              (message-sent)
              (message-received)
              (message-deleted)
              (interrupt-constraint))

  (ai-var (variable-prefix ai_))
  (di-var (variable-prefix di_))
  (ao-var (variable-prefix ao_))
  (do-var (variable-prefix do_))

  (ai-proc (dac di-exp)
           (a+ ai-exp ai-exp)
           (a- ai-exp ai-exp)
           (a* ai-exp ai-exp))
  (di-proc (adc ai-exp)
           (d+ di-exp di-exp)
           (d- di-exp di-exp)
           (d* di-exp di-exp))
  (ao-proc (dac do-exp))
  (do-proc (adc ao-exp))

  (ai-exp ai-var
          ai-proc)
  (di-exp di-var
          di-proc
          number)
  (ao-exp ao-var
          ao-proc)
  (do-exp do-var
          do-proc
          number)

  (input-exp ai-exp
             di-exp)

  (output-exp ao-exp
              do-exp)

  ;needed for variable constraints
  (d-var b-var
         s-var
```

```
       bq-var
       sq-var)

;needed for messages
(d-exp b-exp
       s-exp
       bq-exp
       sq-exp
       ())

(boolean true false)

(b-var (variable-prefix bv_))

(b-exp boolean
       b-var
       b-proc)

(b-proc (head bq-exp)
        (tail bq-exp)
        (get bq-exp s-exp)
        (empty? bq-exp)
        (empty? sq-exp)
        (> s-exp s-exp)
        (>= s-exp s-exp)
        (< s-exp s-exp)
        (<= s-exp s-exp)
        (= s-exp s-exp)
        (!= s-exp s-exp)
        (and b-exp b-exp)
        (or b-exp b-exp)
        (xor b-exp b-exp)
        (not b-exp)
        (get-msg-b s-exp)
        (if b-exp b-exp b-exp))

(scalar number ID)

(s-var (variable-prefix sv_))

(s-exp scalar
       s-var
       s-proc
       (read input-exp))

(s-proc (head sq-exp)
        (tail sq-exp)
        (get sq-exp s-exp)
        (size bq-exp)
        (size sq-exp)
        (+ s-exp s-exp)
        (- s-exp s-exp)
        (* s-exp s-exp)
        (/ s-exp s-exp)
        (abs s-exp)
        (max sq-exp)
        (min sq-exp)
        (get-msg-tag)
        (get-msg-s s-exp)
        (if b-exp s-exp s-exp))

(b-queue (boolean boolean ...))

(bq-var (variable-prefix bq_))

(bq-exp b-queue
        bq-var
        bq-proc)

(bq-proc (get-range bq-exp s-exp s-exp)
         (get-msg-bq s-exp)
         (if b-exp bq-exp bq-exp)
         (if b-exp () bq-exp)
         (if b-exp bq-exp ()))

(s-queue (scalar scalar ...))

(sq-var (variable-prefix sq_))

(sq-exp s-queue
        sq-var
        sq-proc)

(sq-proc (get-range sq-exp s-exp s-exp)
         (get-msg-sq s-exp)
         (if b-exp bq-exp bq-exp)
         (if b-exp () bq-exp)
         (if b-exp bq-exp ()))

(action (set b-var b-exp)
        (set s-var s-exp)
        (set bq-var s-exp b-exp)
        (set sq-var s-exp s-exp)
        (set-queue bq-var bq-exp)
        (set-queue bq-var ())
        (set-queue sq-var sq-exp)
```

```
          (set-queue sq-var ())
          (set-range bq-var bq-exp s-exp)
          (set-range sq-var sq-exp s-exp)
          (set-policy bq-var policy)
          (set-policy sq-var policy)
          (push-front bq-var b-exp)
          (push-front sq-var s-exp)
          (pop-front bq-var)
          (pop-front sq-var)
          (push-back bq-var b-exp)
          (push-back sq-var s-exp)
          (pop-back bq-var)
          (pop-back sq-var)
          (write output-exp s-exp)
          comm)

  (comm (send s-exp dst s-exp (d-exp ...))
        (delete-message))

  (dst s-exp
       ALL)

  ;define context for rewrite rules (not part of grammar):
  (C hole
     (any ... C any ...))
  )
```

Below is the current set of rewrite rules used by GAIL as well as a metafunction for additional processing while some of the rewrite rules are being applied. These rewrite rules allow many optimizations to be expressed very concisely. The metafunction removes redundant conversions by performing substitutions in a similar manner to the rewrite rules.

```
(define rewrite
  (reduction-relation
   gail
   (--> (in-hole C (+ number_1 number_2))
        (in-hole C ,(+ (term number_1) (term number_2)))
        "add-constant")
   (--> (in-hole C (- number_1 number_2))
        (in-hole C ,(- (term number_1) (term number_2)))
        "subtract-constant")
   (--> (in-hole C (* number_1 number_2))
        (in-hole C ,(* (term number_1) (term number_2)))
        "multiply-constant")
```

```
(--> (in-hole C (/ number_1 number_2))
     (in-hole C ,(/ (term number_1) (term number_2)))
     (side-condition (not (zero? (term number_2))))
     "divide-constant")
(--> (in-hole C (/ number_1 0))
     error
     "divide-by-zero")
(--> (in-hole C (* s-exp_1 s-exp_2))
     (in-hole C (* s-exp_2 s-exp_1))
     "multiplicative-commutativity")
(--> (in-hole C (+ s-exp_1 s-exp_2))
     (in-hole C (+ s-exp_2 s-exp_1))
     "additive-commutativity")
(--> (in-hole C (+ (* s-exp_1 s-exp_2) (* s-exp_1 s-exp_3)))
     (in-hole C (* s-exp_1 (+ s-exp_2 s-exp_3)))
     "factor-multiply-addition")
(--> (in-hole C (- (* s-exp_1 s-exp_2) (* s-exp_1 s-exp_3)))
     (in-hole C (* s-exp_1 (- s-exp_2 s-exp_3)))
     "factor-multiply-subtraction")
(--> (in-hole C (+ (/ s-exp_1 s-exp_2) (/ s-exp_3 s-exp_2)))
     (in-hole C (/ (+ s-exp_1 s-exp_3) s-exp_2))
     "factor-divide-addition")
(--> (in-hole C (- (/ s-exp_1 s-exp_2) (/ s-exp_3 s-exp_2)))
     (in-hole C (/ (- s-exp_1 s-exp_3) s-exp_2))
     "factor-divide-subtraction")
(--> (in-hole C (* s-exp_1 1))
     (in-hole C s-exp_1)
     "multiplicative-identity")
(--> (in-hole C (+ s-exp_1 0))
     (in-hole C s-exp_1)
     "additive-identity")
(--> (in-hole C (d+ di-exp_1 di-exp_2))
     (reduce (in-hole C (adc (a+ (dac di-exp_1) (dac di-exp_2)))))
     "dac+")
(--> (in-hole C (a+ ai-exp_1 ai-exp_2))
     (reduce (in-hole C (dac (d+ (adc ai-exp_1) (adc ai-exp_2)))))
     "adc+")
(--> (in-hole C (d- di-exp_1 di-exp_2))
     (reduce (in-hole C (adc (a- (dac di-exp_1) (dac di-exp_2)))))
     "dac-")
(--> (in-hole C (a- ai-exp_1 ai-exp_2))
     (reduce (in-hole C (dac (d- (adc ai-exp_1) (adc ai-exp_2)))))
     "adc-")
(--> (in-hole C (d* di-exp_1 di-exp_2))
     (reduce (in-hole C (adc (a* (dac di-exp_1) (dac di-exp_2)))))
     "dac*")
(--> (in-hole C (a* ai-exp_1 ai-exp_2))
     (reduce (in-hole C (dac (d* (adc ai-exp_1) (adc ai-exp_2)))))
```

$$\begin{tikzcd}\boxed{A} \arrow[r, <->, "\text{multiplicative-commutativity}", "\text{multiplicative-commutativity}"'] & \boxed{B}\end{tikzcd}$$

Figure 3.4: Equivalent GAIL Programs - Task 3 Example
(A and B are GAIL programs. They are listed in the text.)

```
        "adc*")
   ))



;reduces redundant conversions
(define-metafunction gail
  reduce : any -> any  ;reduce takes any and returns any
  ;these substitutions are tried in order
  ((reduce (adc (dac any))) (reduce any))
  ((reduce (dac (adc any))) (reduce any))
  ((reduce (any ...)) ((reduce any) ...))
  ((reduce any) any)
  )
```

3.3 Examples

The following examples demonstrate the results of applying the current set of rewrite rules on the examples from Section 2.5.

3.3.1 Task 1 Example

Due to the simple nature of the Task 1 example, it cannot be rewritten using the current set of rewrite rules.

3.3.2 Task 3 Example

Figure 3.4 shows the Task 3 example after applying the rewrite rules. The only rule that was applied changed the order of the parameters of the multiply operation in the program. This will

```
(--> (in-hole C (+ (read di-exp_1) scalar_1))
     (in-hole C (read (d+ di-exp_1 scalar_1)))
     "read d+")
```

Figure 3.5: New Rewrite Rule - Lights Example

not affect the program's performance, but helps demonstrate the idea of rewrite rules using a trivial case. Listed below are the equivalent programs.

```
A: ((MICAZ_SENSORS
     (define-sq sq_samples 6 (0 0 0 0 0) DISPLACE))
    ((guarded-action "sense"
                     (time-constraint 0 2000)
                     true
                     ((push-front sq_samples (read (adc ai_temp)))))
     (guarded-action "send"
                     (guard-constraint "sense")
                     (> (head sq_samples)
                        (* 1.1 (max (get-range sq_samples 1 5))))
                     ((send RADIO 0 0 (ID
                                       (head sq_samples)))))))

B: ((MICAZ_SENSORS
     (define-sq sq_samples 6 (0 0 0 0 0) DISPLACE))
    ((guarded-action "sense"
                     (time-constraint 0 2000)
                     true
                     ((push-front sq_samples (read (adc ai_temp)))))
     (guarded-action "send"
                     (guard-constraint "sense")
                     (> (head sq_samples)
                        (* (max (get-range sq_samples 1 5)) 1.1))
                     ((send RADIO 0 0 (ID
                                       (head sq_samples)))))))
```

3.3.3 Lights Example

In order to demonstrate creating a rewrite rule, assume that the energy consumption of a GAIL program can be lowered by replacing the addition of a sensor and a number with a special piece of hardware that will do the same thing. Figure 3.5 shows a rewrite rule that could make such an

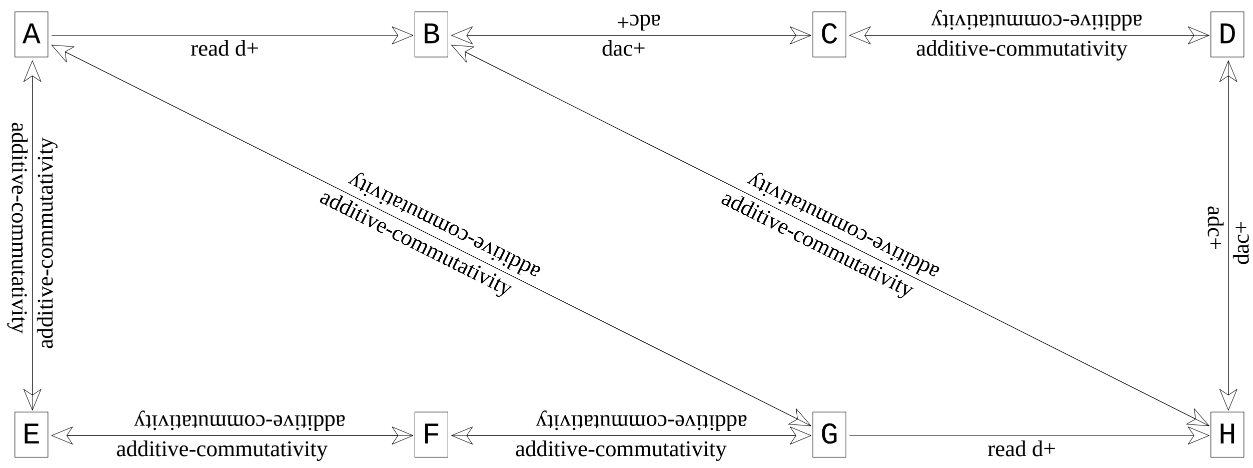


Figure 3.6: Equivalent GAIL Programs - Lights Example
(A-H are GAIL programs. A and B are listed in the text.)

optimization. Notice that adding this optimization takes only a few lines of code. Also notice that because of the additive-commutativity rule, the rule only needs to be written for one case. Figure 3.6 shows the Lights example after applying the rewrite rules including the new one, called "read d+". Listed below is the original program (Program A), and one with the optimization applied (Program B).

```
A: ((MICAZ_SENSORS
     (define-sv sv_light 0)
     (define-sv sv_leds 0))
    ((guarded-action "test"
                     (time-constraint 0 3000)
                     true
                     ((set sv_light (+ (read (adc ai_light)) 15))
                      (set sv_leds (if (> sv_leds 7) 0 (+ sv_leds 1)))
                      (write do_leds sv_leds)
                      (send RADIO 0 0 (sv_light))))))

B: ((MICAZ_SENSORS
     (define-sv sv_light 0)
     (define-sv sv_leds 0))
    ((guarded-action "test"
                     (time-constraint 0 3000)
                     true
                     ((set sv_light (read (d+ (adc ai_light) 15)))
(--> (in-hole C (guarded-action string
                                (continuous-constraint)
                                SNW-exp
                                (action_1 action_2 ...)))
     (in-hole C (guarded-action string
                                (interrupt-constraint)
                                (SNW-replace SNW-exp)
                                (action_1 action_2 ...)))
     "shake-n-wake")
```

Figure 3.7: New Rewrite Rule - ACM Example

```
(SNW-exp (> (read input-exp) scalar)
         (any ... SNW-exp any ...))
```

Figure 3.8: New Pattern - ACM Example

```
                        (set sv_leds (if (> sv_leds 7) 0 (+ sv_leds 1)))
                        (write do_leds sv_leds)
                        (send RADIO 0 0 (sv_light))))))
```

3.3.4 ACM Example

To demonstrate a more complex rewrite rule, the ACM Example will be used. In this application, a device called the “Shake ‘n’ Wake” was developed to determine when the value of the crack sensor exceeded a threshold [7]. This device allows the node conserve energy by entering a low power mode until it receives an interrupt from the device.

To start, a new rule is created (see Figure 3.7). A guarded action with a continuous constraint and a comparison of a sensor and a scalar somewhere in the guard will trigger the rule. The continuous constraint will be replaced with an interrupt constraint and the comparison will be replaced by `true`. It is important to note that interrupt constraints have not yet been implemented, so in a future version, more information would likely need to be supplied to the constraint (see Section 6.3 for more details).

```
;replace shake-n-wake comparison with true
(define-metafunction gail
  SNW-replace : any -> any  ;SNW-replace takes any and returns any
  ;these substitutions are tried in order
  ((SNW-replace (> (read input-exp) scalar)) true)
  ((SNW-replace (any ...)) ((SNW-replace any) ...))
  ((SNW-replace any) any))
```

Figure 3.9: New Metafunction - ACM Example

$$\begin{tikzcd}
\boxed{A} \arrow[r, "\text{shake-n-wake}"'] \arrow[d, leftrightarrow, "\text{additive-commutativity}"', "\text{additive-commutativity}"] & \boxed{B} \arrow[d, leftrightarrow, "\text{additive-commutativity}"', "\text{additive-commutativity}"] \\
\boxed{C} \arrow[r, "\text{shake-n-wake}"'] & \boxed{D}
\end{tikzcd}$$

Figure 3.10: Equivalent GAIL Programs - ACM Example
(A-D are GAIL programs. A and B are listed in the text.)

To add this more complex optimization, a new pattern also needs to be added to the language definition because the comparison may occur anywhere in the guard (see Figure 3.8). In addition, a new metafunction needs to be written to replace the comparison in the guard with `true` (see Figure 3.9). This optimization demonstrates that even complex optimizations can be written in a concise way in GAIL. Figure 3.10 shows the ACM example after applying the rewrite rules including the new one, called "shake-n-wake". Listed below is the original program (Program A), and one with the optimization applied (Program B).

```
A: ((MICAZ_SENSORS
     (define-sq sq_samples 31 () DISPLACE)
     (define-sv sv_light 0)
     (define-sv sv_state 0)
     (define-sv sv_count 0))
    ((guarded-action "poll"
                     (continuous-constraint)
                     (and (= sv_state 0) (> (read (adc ai_light)) 200))
                     ((set sv_state 1)
                      (set sv_count 0)
                      (push-back sq_samples (read (adc ai_light)))
                      (send RADIO ALL 1 ())))
     (guarded-action "sample"
                     (time-constraint 0 100)
                     (= sv_state 1)
                     ((push-back sq_samples (read (adc ai_light)))
                      (set sv_count (+ sv_count 1))))
     (guarded-action "stop_sampling"
                     (guard-constraint "sample")
                     (> sv_count 30)
                     ((set sv_state 0)
                      (send RADIO 0 2 (sq_samples))
                      (set-queue sq_samples ())))
     (guarded-action "remote_trigger"
                     (message-received)
                     (= (get-msg-tag) 1)
                     ((set sv_state 1)
                      (delete-message)))))

B: ((MICAZ_SENSORS
     (define-sq sq_samples 31 () DISPLACE)
     (define-sv sv_light 0)
     (define-sv sv_state 0)
     (define-sv sv_count 0))
    ((guarded-action "poll"
                     (interrupt-constraint)
                     (and (= sv_state 0) true)
                     ((set sv_state 1)
                      (set sv_count 0)
                      (push-back sq_samples (read (adc ai_light)))
                      (send RADIO ALL 1 ())))
     (guarded-action "sample"
                     (time-constraint 0 100)
                     (= sv_state 1)
                     ((push-back sq_samples (read (adc ai_light)))
                      (set sv_count (+ sv_count 1))))
     (guarded-action "stop_sampling"
                     (guard-constraint "sample")
                     (> sv_count 30)
```

```
                    ((set sv_state 0)
                     (send RADIO 0 2 (sq_samples))
                     (set-queue sq_samples ())))
    (guarded-action "remote_trigger"
                    (message-received)
                    (= (get-msg-tag) 1)
                    ((set sv_state 1)
                     (delete-message)))))
```

3.4 Objective Functions

During the compilation process, rewrite rules are applied to GAIL programs to produce equivalent programs. Then, an objective function is used to choose the best program. Due to the constrained nature of GAIL, and because programs can be modeled as finite state machines, it is possible to write objective functions that analyze many different aspects of the various equivalent programs. Also, since GAIL includes hardware descriptions, the actual hardware must be chosen from a library to determine the properties of the hardware so that they may be included in the analyses. The current implementation does not provide interface instructions, such as instructions for connecting wires from different pieces of hardware, but a future version will likely determine this information automatically when choosing hardware from the library.

Since all variables are statically allocated, memory usage in GAIL programs is easy to analyze. Also, by examining the output code, the maximum stack depth can be determined. By analyzing the assembly output of the compiled GAIL programs, it is also possible to estimate the power or energy consumption and the runtime of programs given enough information about the platform on which they are run. Also, since GAIL includes hardware descriptions, when the hardware pieces are chosen from a library, additional information such as the total cost, total volume, and total weight can be determined. The objective function used in the current implementation simply chooses the original program, but future versions will likely use a combination of many different types of program analyses to determine which program is best suited for the application.

Chapter 4

Code Generator

In order to compile GAIL programs for various platforms, the programs are first compiled into C using a code generator. The code generator is built using a tool called LISA. The C code can then be compiled by the MANTIS OS compiler to an object file that can be loaded directly onto the hardware.

4.1 LISA

The code generator is built using LISA, a tool that automatically generates a Java-based compiler from a specification file. The specification file contains regular expressions for the lexical analysis, attributes for the nonterminals in the language, and a formal grammar in BNF along with Java assignment statements for the attributes.

From the specification file, LISA generates a scanner file, a parser file, an evaluator file, and a compiler file. These four Java files can then be compiled to produce the GAIL code generator. The scanner splits the input into tokens and then the parser arranges the tokens into a tree. Then, the evaluator takes the tree and determines the values of the attributes for each node in the tree.

The attributes for the nonterminals are Java types, such as `int` or `String`. These attributes are determined using the Java assignment statements provided with the formal grammar. LISA capable of automatically determining whether each attribute is inherited or synthesized. The values of inherited attributes are dependent on the parent and siblings of the node in the tree, while the values of synthesized attributes are dependent on the node's children. Finally, the compiler file combines the scanner, parser, and evaluator together.

4.1.1 LISA Example

Below is an example that uses LISA to compile the simple-lang example from Section 3.1 into C code. The first part of the file lists the regular expressions that define the terminals in the language. Next, the attributes are defined. Programs have a `code` attribute, and expressions have a `val` attribute and a `code` attribute. The `code` attributes will hold the generated code while the `val` attribute will contain the name of the variable where the value of the expression stored. All of these attributes are synthesized attributes. After the attributes are defined, the grammar is specified along with the attribute assignment statements. Finally, an additional function is defined to generate unique names for temporary values in the generated code.

```
language SimpleLang {
    lexicon {
        NUMBER \-?[0-9]+(.[0-9]+)?
        PLUS \+
        MINUS \-
        LP \(
        RP \)
        //space, tab, line feed, carriage return
        WHITESPACE [\ \0x09\0x0A\0x0D]
        ignore #WHITESPACE
    }

    attributes String PROG.code, EXP.val, EXP.code;

    rule Program {
        PROG ::= EXP compute {
            PROG.code = "#include <stdio.h>\n\nint main() {\n" +
                EXP.code + "\nprintf(\"%f\\n\", " + EXP.val +
                ");\n\nreturn 0;\n}";
        };
    }

    rule Expression {
        EXP ::= #NUMBER compute {
            EXP.val = getTemp();
            EXP.code = "float " + EXP.val + " = " +
                #NUMBER.value() + ";\n";
        };

        EXP ::= ( #PLUS EXP EXP ) compute {
            EXP[0].val = getTemp();
            EXP[0].code = EXP[1].code + EXP[2].code + "float " +
                EXP[0].val + " = " + EXP[1].val + " + " +
                EXP[2].val + ";\n";
        };

        EXP ::= ( #MINUS EXP EXP ) compute {
            EXP[0].val = getTemp();
            EXP[0].code = EXP[1].code + EXP[2].code + "float " +
                EXP[0].val + " = " + EXP[1].val + " - " +
                EXP[2].val + ";\n";
        };
    }

    method Temps {
        int tempCount = 1;

        String getTemp() {
            return "temp" + tempCount++;
        }
    }
}
```

When the code generator is run on the the following simple-lang program, the following steps are taken by the evaluator.

Input Program: `(+ (- 7 2) (- 4 3))`

Evaluator Steps:

1. `(+ (- 7 2) (- 4 3))` is a program.
   `(+ (- 7 2) (- 4 3)).code = ?`

2. `(+ (- 7 2) (- 4 3))` is an expression.
   `(+ (- 7 2) (- 4 3)).val = "temp1"`
   `(+ (- 7 2) (- 4 3)).code = ?`

3. `(- 7 2)` is an expression.
   `(- 7 2).val = "temp2"`
   `(- 7 2).code = ?`

4. `7` is an expression.
   `7.val = "temp3"`
   `7.code = "float temp3 = 7;\n"`

5. `2` is an expression.
   `2.val = "temp4"`
   `2.code = "float temp4 = 2;\n"`

6. `(- 7 2).code = 7.code + 2.code + "float temp2 = temp3 - temp4;\n"`

7. `(- 4 3)` is an expression.
   `(- 4 3).val = "temp5"`
   `(- 4 3).code = ?`

8. `4` is an expression.
   `4.val = "temp6"`
   `4.code = "float temp3 = 4;\n"`

9. `3` is an expression.
   `3.val = "temp7"`
   `3.code = "float temp4 = 3;\n"`

10. `(- 4 3).code = 4.code + 3.code + "float temp5 = temp6 - temp7;\n"`

11. `(+ (- 7 2) (- 4 3)).code = (- 7 2).code + (- 4 3).code + "float temp1 = temp2 + temp5;\n"`

12. The final program code, shown below, is generated.

```
#include <stdio.h>

int main() {
float temp3 = 7;
```

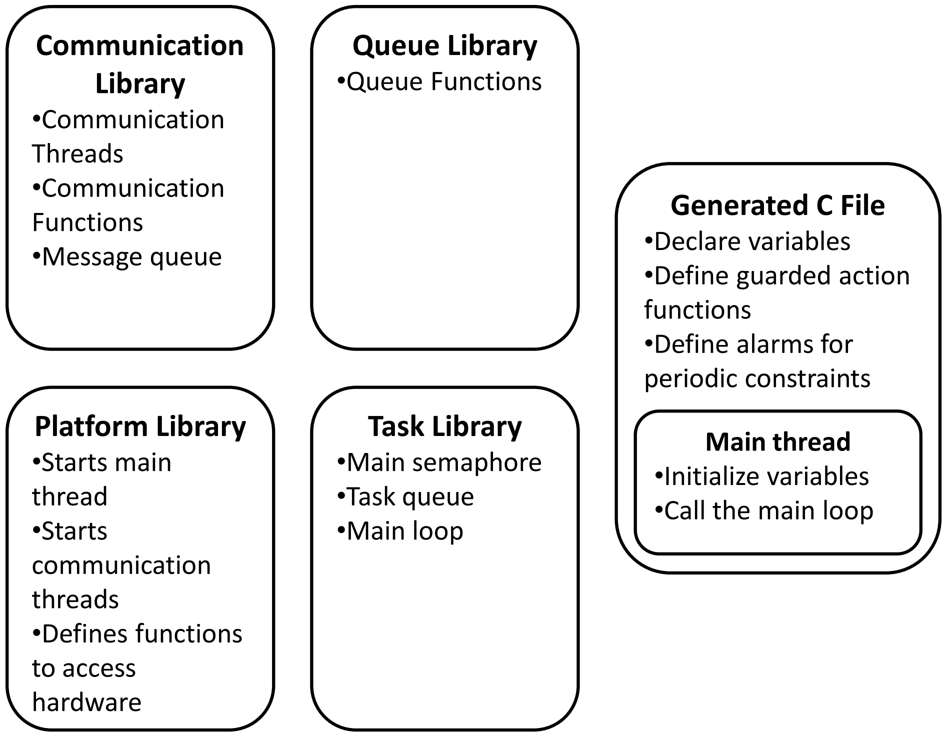


Figure 4.1: GAIL Libraries

```
float temp4 = 2;
float temp2 = temp3 - temp4;
float temp6 = 4;
float temp7 = 3;
float temp5 = temp6 - temp7;
float temp1 = temp2 + temp5;

printf("%f\n", temp1);

return 0;
}
```

4.2 GAIL C Code

Before GAIL code is compiled into C code, it is processed using a macro designed for the specific platform on which the program is to run. These macros allow for keywords to be defined for

	Code Size (in bytes)		**Size in Memory (in bytes)**	
	GAIL	**C**	**Code**	**Data**
GAIL Library	N/A	26,969	5,872	177
Task 1 Example	281	2,682	567	14
Task 3 Example	502	4,683	1,927	45
Lights Example	382	3,157	759	22
ACM Example	1,109	7,564	2,151	157

Figure 4.2: Sizes
(All sizes are in bytes.)

built-in sensors so that the hardware available on the platform is stored in one central place.

The C code produced by the GAIL code generator is designed to be used with MANTIS OS, an operating system designed for wireless sensor networking platforms. The final code follows a simple template that uses the GAIL libraries. Figure 4.1 shows the contents of the GAIL libraries.

The generated code contains a main thread as well as threads for each interface over which messages are to be received. The receive threads simply initialize their interface, and then wait for messages to be received. Once a message is received, the receive threads place the message into a special message queue for the main thread to handle.

The main thread starts by initializing all of the variables in the program, as well as any timers that will be used by guards with periodic constraints. Next, it enters the main loop, where it waits on the main semaphore. When it is time for a guard to be evaluated, a pointer to the guard's function is added to a task queue, and then the thread posts to the main semaphore, causing the main thread to wake up. The main thread then calls the functions in the task queue until the queue is empty, then loops back to wait on the main semaphore again.

4.3 Examples

The examples from Section 2.5 have been compiled using the code generator to demonstrate the output of the code generator. Figure 4.2 shows the size the size of the GAIL library as well as the size of each of the examples. The total memory usage of an application can be determined at compile time by calculating the sum of the code, the data, and the stack for MANTIS OS, the GAIL library, and the program. Since there is no dynamic allocation, no heap space is required.

4.3.1 Task 1 Example C Code

Below is the generated C code for Task 1 (see Section 2.5.1 for the Task 1 example).

```
//(((define-di (adc ai_light) "getLight") (define-di (adc ai_temp) "getTemp")
//(define-do do_leds "setLEDs") ) ((guarded-action "sense_and_send"
//(time-constraint 0 2000) true ((send 2 0 0 (ID (read (adc ai_light))
//(read (adc ai_temp))))))))

/* Begin Generated Code */

#include "linux.h"

/* Begin variable definitions */

/* End Variable Definitions */



/* Begin Defining Guarded Actions */

//(guarded-action "sense_and_send" (time-constraint 0 2000) true
//((send 2 0 0 (ID (read (adc ai_light)) (read (adc ai_temp))))))
/*************************************************************************/
//true
inline boolean_t sense_and_send_guard() {
    return TRUE;
}

//((send 2 0 0 (ID (read (adc ai_light)) (read (adc ai_temp)))))
inline void sense_and_send_actions() {
    //Side-effect-free code
    //Side-effect code
    /* Begin (send 2 0 0 (ID (read (adc ai_light)) (read (adc ai_temp)))) */
    send_pkt.size = sizeof(scalar_t) + sizeof(scalar_t) + sizeof(scalar_t) +
        0 + 6 + 2*sizeof(uint16_t) + sizeof(scalar_t);
    uint8_t* temp1 = send_pkt.data;
    *((uint16_t*)temp1) = node_id;
    temp1 += sizeof(uint16_t);
    *((uint16_t*)temp1) = 0;
    temp1 += sizeof(uint16_t);
    *((scalar_t*)temp1) = 0;
    temp1 += sizeof(scalar_t);
    *temp1 = sizeof(scalar_t);
    temp1 += sizeof(uint8_t);
    *temp1 = SCALAR_TYPE;
    temp1 += sizeof(uint8_t);
    *((scalar_t*)temp1) = node_id;
    temp1 += sizeof(scalar_t);
    *temp1 = sizeof(scalar_t);
    temp1 += sizeof(uint8_t);
    *temp1 = SCALAR_TYPE;
    temp1 += sizeof(uint8_t);
    *((scalar_t*)temp1) = getLight();
    temp1 += sizeof(scalar_t);
    *temp1 = sizeof(scalar_t);
    temp1 += sizeof(uint8_t);
    *temp1 = SCALAR_TYPE;
    temp1 += sizeof(uint8_t);
    *((scalar_t*)temp1) = getTemp();
    temp1 += sizeof(scalar_t);
    send((uint8_t)2);
    /* End (send 2 0 0 (ID (read (adc ai_light)) (read (adc ai_temp)))) */
}

//(time-constraint 0 2000)
mos_alarm_t sense_and_send_alarm;
void sense_and_send_ga() {
    if (sense_and_send_guard()) {
        sense_and_send_actions();
    }
}
/******************************************************************************/

/* End Defining Guarded Actions */



void mainThread() {
    /* Begin Init Code */
    initScheduler();

    //(time-constraint 0 2000)
    sense_and_send_alarm.func = wakeUp;
    sense_and_send_alarm.data = (void*)sense_and_send_ga;
    sense_and_send_alarm.msecs = 0;
    sense_and_send_alarm.reset_to = 2000;
    mos_alarm(&sense_and_send_alarm);

    /* End Init Code */



    /* Begin Guarded Actions */

    runScheduler();

    /* End Guarded Actions */
}

/* End Generated Code */
```

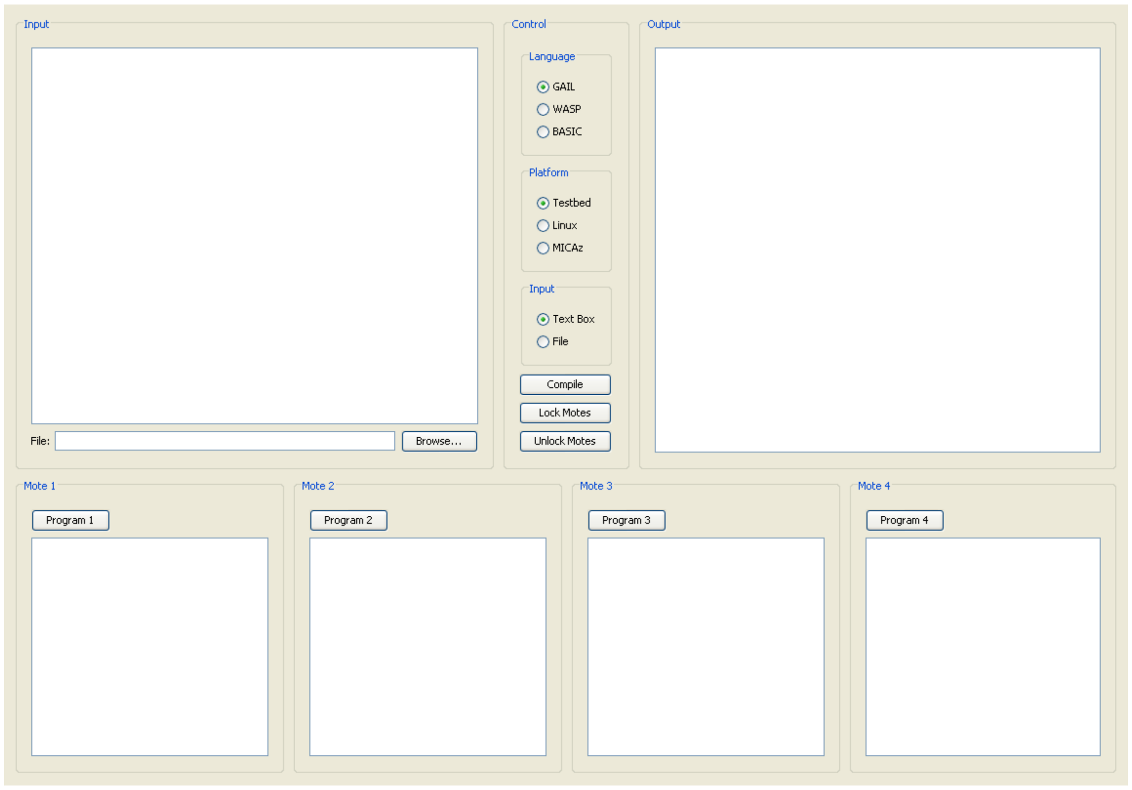
Chapter 5

User Interface

Setting up the environment to compile GAIL programs with MANTIS OS is somewhat complex and time consuming. Because of this, setting up a web-based system is a logical choice to allow multiple people to test GAIL. The web-based system uses a Java applet to allow users to write code, compile the code on the server, remotely program a testbed of MICAz Modules via ethernet-based programmers, and remotely view the serial output of the MICAz Modules. There is also a link to a streaming webcam so that users can view the hardware and see LEDs blinking or hear buzzers sounding.

5.1 Server

The server, written in Java, creates a new thread for each user that connects to it. These threads then wait for requests to be received from clients, perform the necessary tasks on the server, and send back the responses. In addition to these threads, if a user requests to program a module from the testbed, a new thread is created to forward the output from the module's serial port to the client. The current implementation of the server only allows one user at a time to be actively compiling a



Language: [✓ GAIL] WASP BASIC

Platform: [✓ Testbed] Linux MICAz

Input: [✓ Text Box] File

Figure 5.1: Web Interface

program. In addition, only one user at a time may use the testbed.

5.2 Client

The client is also written in Java, but takes the form of an applet embedded in an HTML page. When a user clicks a button, the client sends the request to the server, then creates a new thread to wait for the response since the thread that handles the button events is also the thread responsible for the graphical part of the applet. Like the server, if the client requests to program a module from the testbed, a new thread is created to receive the data from the module and display it. Figure 5.1 shows a screenshot of the web interface.

5.3 Webcam

The current webcam setup is very simple. The video from a simple USB webcam is captured by VLC, a media server and media player. Then, VLC transcodes the video and distributes the streaming video. The user receives the video stream via a Flash-based media player embedded in an HTML page.

Chapter 6

Future Work

Since the current implementation of GAIL is a proof of concept, some parts of GAIL were either simplified or left out. One part that was left out is ensuring that hardware is only sampled once per set of actions (Rule 3 from Section 2.3), however since the time delay between samples in the same set of actions is likely to be small, the difference in samples is also likely to be small except in applications with rapidly changing environments. Also, some of the constraints have not yet been implemented. Other features, like random numbers and time of day would further improve the expressiveness of the language.

6.1 Implementation of Data Types

One possible improvement is to change the implementation of data types in GAIL. For example, scalars are currently implemented as 32-bit floating point numbers, but the implementation might be changed to allow larger numbers, larger precision, or a more compact representation. Also, queues are currently implemented as arrays, but these could possibly be implemented as sparse arrays or some other type in order to better accommodate the requirements of the application. They could also possibly use flash memory as storage for larger queues that are accessed infrequently.

6.2 Communication

The current implementation of the communication is a single-hop unicast model. Obviously, this severely limits the types of applications possible. A more complex implementation could provide multi-hop communication or the ability to self heal in the case of node failure.

6.3 Interrupts

Interrupt constraints are not yet implemented, and will need to be changed in future versions to include more information about the interrupts. One possibility is to add a `define-interrupt` expression to define the hardware that is connected to a particular interrupt pin. In addition, interrupts could behave similarly to incoming messages by having an interrupt queue and a `(get-interrupt)` expression. This would allow guarded actions to specify which queue they are monitoring.

6.4 Optimizations

The current set of optimizations that are performed on GAIL programs is quite limited, making this an area that needs quite a bit of improvement. However, because GAIL uses rewrite-based optimization, adding new optimizations can be done in very few lines of code. Also, adding the ability to create simple functions in GAIL, similar to macros, might allow some programs with large amounts of redundant code to become more compact. To allow some energy-saving optimizations to be made, the implementation or design of some of the constraints might have to be modified to properly allow the device to enter a low power mode when appropriate.

As more optimizations are added, the number of equivalent programs generated will increase. Because of this, it may no longer be reasonable to use the objective function on all possible equivalent programs. Instead, it may be more beneficial to use a directed search algorithm so that only a small subset of the set of all possible equivalent programs needs to be searched. Also, to facilitate analyzing the equivalent programs during the rewriting process, the equivalent programs can be compiled to object files which can then be analyzed together with their GAIL programs.

Also, the optimizations in GAIL need to be verified, and perhaps the language needs to be expanded in order to ensure that accuracy is maintained and that rewritten programs are actually equivalent. Many possible optimizations might involve rewriting software into specialized hardware, or vice versa. In either case, special care has to be taken to ensure the accuracy of the resulting expression.

6.5 Hardware and Software Models

Due to time limitations, the optimizations currently present in GAIL have not been verified. Verifying that these optimizations are in fact beneficial, as well as testing the power and energy consumption of the code produced would further validate the language. Developing more complex hardware and software models will allow better estimates of node-level costs such as price and energy consumption. In addition, these models could be used to estimate the impact that each optimization will have on the node-level costs.

6.6 Hardware Library

Currently, the ability to describe hardware in GAIL is somewhat limited. Expanding this would result in the ability to incorporate a larger variety of types of hardware. It would also allow more accurate descriptions of hardware, which would result in a larger number of possible optimizations and it would help to ensure that optimizations do not alter the meaning of programs.

Also, the entire process of selecting hardware has not been implemented. First, a library of available hardware needs to be created. Then, individual pieces of hardware need to be selected, and the user needs to receive instructions on how to assemble the pieces.

6.7 Runtime Errors

Even though GAIL is a constrained language, it is still possible to encounter runtime errors, such as attempting to pop an item from an empty queue. The current method for handling these errors is to halt all execution. A better way to handle errors might be to let the application decide. For example, perhaps it would be better in some applications to send a message back to the root node when an error occurs. Also, since the errors can be detected before any actions with side effects are taken, the node could simply not execute the actions, and alert other nodes to the fact that an error has been encountered, and the error could be handled accordingly.

6.8 User Interface

Although the current user interface is functional, many of the planned features have not been implemented. These features include uploading and downloading files, rather than typing input into a text box, as well as adding support for additional languages under development by the ABSYNTH Project.

There is also much that can be done to improve the user interface. One such improvement would be to add additional graphics to the applet so that users can clearly see the state of the hardware, and possibly be able to actuate different parts remotely. For example, the user might be able to remotely access a light in order to simulate different values from a light sensor. Also, the interface could be improved by adding additional features commonly found in integrated development environments, such as syntax highlighting, line numbers, and code completion.

Finally, if the server is to be used be more than just the members of the ABSYNTH Project, parts of it will need to be modified to allow it to handle more clients. In particular, the server should allow users to queue up jobs if the someone else is using the testbed. Also, having one thread per user will not scale well if there are too many users connected at one time.

Chapter 7

Conclusion

The use of GAIL as an intermediate language offers many advantages. It is based on a constrained guarded action model and is not Turing-complete so that GAIL programs can be modeled as finite state machines. Because of this, it possible to analyze many aspects of programs at compile time in order to assist in optimizing costs and trying to ensure that system-level constraints are met in wireless sensor network applications. The rewrite-based joint hardware-software optimization makes creating new optimizations simple, and allows the exploration of trade-offs between evaluating complex expressions in hardware and software. Although the current implementation does not yet offer the full set of features, it is already evident that GAIL can be used as part of a larger end-to-end system to design and create wireless sensor networks.

Bibliography

[1] Lan Bai, Robert Dick, and Peter Dinda. Archetype-based design: Sensor network programming for application experts, not just programming experts. In *Proc. Int. Symp. Information Processing in Sensor Networks*, April 2009.

[2] Andrew R. Dalton, William P. McCartney, Kajari Ghosh Dastidar, Jason O. Hallstrom, Nigamanth Sridhar, Ted Herman, William M. Leal, Anish Arora, and Mohamed Gouda. DESAL$^{\alpha}$: An implementation of the dynamic embedded sensor-actuator language. In *The Proceedings of The 17th International Conference on Computer Communications and Networks (ICCCN'08)*, page 7pp, Washington DC, USA, August 2008. IEEE Computer Society.

[3] C. H. Dowding and L. M. McKenna. Crack response to long-term and environmental and blast vibration effects. *Journal of Geotechnical and Geoenvironmental Engineering*, 131(9):1151–1161, September 2005.

[4] C. H. Dowding, H. Ozer, and M. Kotowsky. Wireless crack measurment for control of construction vibrations. In *Proceedings of the Atlanta GeoCongress*, Engineering in the Information Technology Age. Geo-Institute of the American Society of Civil Engineers, 2006.

[5] L. Evers, P.J.M. Havinga, and J. Kuper. Dynamic sensor network reprogramming using sensorscheme. In *Proceedings of the 18th Annual IEEE Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5, Los Alamitos, September 2007. IEEE Computer Society Press.

[6] L. Evers, P.J.M. Havinga, J. Kuper, M.E.M. Lijding, and N. Meratnia. Sensorscheme: Supply chain management automation using wireless sensor networks. In *IEEE Conference on Emerging Technologies & Factory Automation, ETFA*, pages 448–455, 2007.

[7] Sasha Jevtic, Mathew Kotowsky, Robert P. Dick, Peter A. Dinda, and Charles Dowding. Lucid dreaming: Reliable analog event detection for energy-constrained applications. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN)*, April 2007.