Verifying Baklava

Report

Revision: 13

Created: March 10, 2006

Last modified: April 26, 2006

Prepared by: Saeed Jahed and Paul Vrbik

Supervised by: Christopher Anand and Wolfram Kahl

Abstract

This report discusses a method of pseudo-formal verification of concurrency models generated by the Coconut[Ana05] compiler. These concurrency models, specified by a language called Baklava[1] as described in [ACHZ05], are translated to terms of the π-calculus. Correctness theorems are formed and proven using logical inference in an automated proving environment built on an implementation of Rewriting Logic called Maude.

While the method can only verify finite Baklava code-sets, it lays the ground work for verification of infinitely long code-sets by creating theorems and proving methods that can be used on both finite and infinite Baklava code-sets. Verification of infinitely long Baklava code-sets is equivalent to formal verification of the Geometer (the component of the Coconut compiler which generates Baklava code-sets.)

[1] Previously known as Onion.

Contents

1 Introduction

Formal verification has become an important step in software development and has grabbed the much needed attention from the developer community as they can detect errors early in the development process and thereby reduce the chances of or even avoid the release of faulty software. It also provides assurance of correctness to the end-user.

Mathematics forms the basis of formal verification [Pan04]. It yields methods to model, in a precise and non-ambiguous way, specification of systems. Furthermore, it provides methods for analyzing such systems and thereby observing properties. Formal verification is the process of observing desired properties.

There are two main methods for formal verification [Wik06]. *Model checking* is the method where the system is modelled using a finite state machine and all the different states of the system are explored. Model checking on its own can't be generalized to work with systems that work with infinite states. An alternative method is *logical inference*. This is when the system is described on some abstract algebra and then reduced to a property using equations and reduction rules. Such properties are theorems that can be proven using automated proving systems.

There are many tools and software packages available to aid this formal verification process. *Spin* is a popular open-source model checker developed at Bell Laboratories. It comes with a specification language called PROMELA and supports the use of linear temporal logic which is used to reason and describe time-based properties for specification of correctness properties. While PROMELA is the only language supported by Spin, it is definitely not the only *process calculus* available to us. ACP (Algebra of Communicating Processes), CCS (Calculus of Communicating Systems), and the π-calculus are some alternative process calculi that can be used to formally describe a concurrent system.

These languages can also be used to form correctness properties about such systems. Each specification calculus comes with its own set of axioms and inference rules. The axioms

are used as the basis for constructing terms in that calculus. The inference rules along with equivalence relationships can be used to identify *equivalent systems*. Detecting the existence (or the absence of) equivalence between two concurrent systems is at the heart of verification by logical inference. When two systems are equivalent (or when one system can be reduced to the other) then it means that one system, through computational steps, can become the other system. Such equivalence statements are conjectures that can be proven using automated theorem proving systems or alternatively even manually by hand. HOL (stands for Higher Order Logic), Isabelle, and PVS (Prototype Verification System) are among the most popular automated proving systems.

1.1 Baklava

The Geometer is the component of the Coconut compiler that generates Baklava code-sets to handle the runtime concurrency of the compiled Coconut program [ACHZ05].

Informally, verification of Baklava is the process of verifying that Baklava code-sets are parallelized correctly by the Coconut Geometer. That is, when executed on a computer cluster they return the same results, independent of timing issues.

There are two approaches to such verification. The first approach is to verify each generated Baklava code-set as a concurrent system on its own. This approach is independent of the Geometer. That is, there is no need to know how the Geometer works. The second is to verify that the current Geometer always generates correct Baklava code-sets which requires a formal specification for the Geometer.

In this project the first approach was used. However heavy consideration for the second motivated the implementation of easily generalized methods.

2 Objectives and requirements

The objective of this project was to build a system to serve in the formal verification of Baklava code-sets. This was both to aid detect errors in the development of the Coconut Geometer and also provide the Coconut end-user with assurance of correctness.

In an overview, this is to build a fast (section 2.2) and reliable tool to formally verify the correct parallelization of Baklava code-sets (section 2.1) which can easily adapt to newly added or modified components of the Geometer (section 2.3) and whose modules can be reused for formal verification of other critical Coconut components.

2.1 Verified Baklava

A "correct" Baklava code-set is one that returns the same results independent of timing issues and without running into a deadlock.

Deadlock A system is in a state of *deadlock* if one or more of its processes are waiting on something that will not come in a finite period of time.

Result variations A system has *result variations* if it can compute two or more different results depending on the timing of its concurrent processes.

Note that these properties on a system will need to be precisely defined.

Formal definition requirement We will *formally* and *precisely* define the meaning of deadlock and result variations on a system. From that we will formally define what is considered a "correct" Baklava parallelization.

2.2 Performance

Coconut programs usually run long periods of time over large sets of data. This results in large sets of Baklava code that will need to be verified. Slow and resource consuming algorithms for verification of these Baklava code-sets can cause a bottle neck, especially if the Baklava code-sets are to be verified on-the-fly as they are being sent to the cluster.

Performance requirement. The Baklava verifier needs to verify Baklava code-sets faster than they can be executed on a potential computer cluster.

Of course this heavily depends on the size of the cluster and the platform the verifier is ran on. Although we could not meet such a requirement, the formal verification of infinitely long Baklava code-sets (for which we laid the groundwork) would satisfy such a requirement.

2.3 Modular and Reusable

The Coconut project is in a development stage where new components are added often and old components are replaced, rewritten, and modified frequently. We need to take this into account if we expect our project to be of any use to Coconut in the near future.

Modularity requirement. Both the tools and the methods we develop should be constructed modularly so that changes made to the Geometer and the definition of Baklava can only effect a small component of our tools and methods, rather than effecting the entire project.

Reusability requirement. Both the tools and the methods we develop should be constructed in a manner to allow portions of our project to be reused in formal verification of other (even non-concurrent) components of the coconut.

2.4 Two end-users

This project was aimed to serve two distinct end-user groups.

Geometer developer: First, it was designed to aid in the development process of the Coconut Geometer. The Geometer is still under development and it needs to be tested as new components are developed for it. The Geometer can also generate run-time code for different types of computer clusters. As an example, it currently generates code for a cluster of Apple Macintosh computers with dual G5 processors. In the near future, it will be generating code for the IBM Cell processor as well. Newer components can always be added to the Geometer and they also need to be tested and formally verified.

Coconut Compiler user: Second, Coconut being a tool used for performance-requiring scientific applications, it will be tackling problems that will most likely have heavy resource requirements (time, computation, etc.) If a compiled Coconut program returns no results or incorrect results due to bad parallelization it would be a waste of resources. A Coconut end-user will need to be assured of the correct parallelization of their compiled program.

Developer documentation requirement. As mentioned in the section Modular and Reusable (2.3) portions of our methods and implementations will need to be modified as the Geometer is modified. So we shall document our tools and methods so that they can be easily modified by the Coconut developers.

Coconut developer interface requirement. We shall provide the Coconut developer with an interface to the Baklava verifier, which with only a minimum understanding of the verification process provides the developer with logical reasoning as to why a given Baklava code-set is parallelized correctly or incorrectly. “Logical reasoning” here means a human-readable set of sentences that try to justify our results (as opposed to the formal logical reasoning expressed mathematically.)

Coconut user interface requirement. We shall provide the Coconut end-user with an interface to the Baklava verifier, which without requiring an understanding of the verification process provides the end-user with the certification of correct or incorrect parallelization of Baklava.

3 Design

The goal of this project is to formally verify certain properties of a given Baklava code-set. It is very easy to write programs that can informally check for deadlocks and result variations. It is immensely more difficult to form methods of *formally* verifying a system against its properties. More difficult yet is to develop such formal methods that verify an entire class of such systems (i.e. those Baklava code-sets generated by the Geometer).

To start we needed to be able to formally and thereby precisely express properties as given in the Verified Baklava subsection of the Requirements section. From here on we will use the axioms and the common notations of the Zermelo-Fraenkel set theory for our mathematical formalizations.

What is a *property*? Let $x \in Baklava^*$ represent a Baklava code-set, where $Baklava^*$ is the set of all Baklava code-sets. Let $P(x)$ be a *predicate* (as in the first-order predicate calculus or as a function relationship $P \subseteq Baklava^* \times \{\texttt{true}, \texttt{false}\}$) so that it is either *true* or *false*. P is a *property* on Baklava.

Before we can form properties on the set of all Baklava code-sets, $Baklava^*$, we need to be able to form the members of such a set. That is, we need a language to express a Baklava code-set. Considering how Baklava was intended to express concurrent systems, we turned to *process calculi* (also referred to as *process algebra*). Process calculi are languages for modeling concurrent systems.

Using a mathetical notation of the Baklava language itself to describe Baklava code-sets was also an option, but this makes it more difficult to express properties about Baklava without somehow formally specifying the semantics of Baklava. Using a process calculus also has given us the advantage of being able to formally specify the semantics of Baklava via a translation scheme between Baklava and that process calculus.

We looked at two process calculi, CCS (Calculus of Communicating Systems) and the π-

calculus, both developed by Robbin Milner, and formally defined in [Mil82] and [Mil99] respectively. It was decided that the π-calculus would be used since the π-calculus was development as an *improvement* over the CCS and that the improvements did not seem to convolute the intended meaning. One important notation in the π-calculus for us was the notion of *mobility*; that is, the ability of the π-calculus to communicate *names* via *channels* (which are names themselves,) and to evolve only through such communications. This fit the model of a eynchronized computer cluster well for our purposes; we can not only use it to describe the concurrency semantics of the model, but also, the concurrency semantics of each machine on the cluster, and each action on each machine (such as reading from memory, changing status, etc.).

Our time-frame did not allow for us to identify whether the improvements of the π-calculus over CCS (such as mobility) could have been unneeded. Ideally we would have also liked the chance to consider more calculi, although we did briefly look at PROMELA [Hol91] of the Spin verification tool.

We turned to [Mil93] to gain a basic understanding of the π-calculus.

3.1 Introduction The π-calculus

The π-calculus is a convenient method for describing concurrent systems like parallel processes. As in any parallel process the most fundamental aspect of the system is the ability to transmit information between nodes. Representation of this in the π-calculus is achieved by using what we call a *name*. Names typically denoted by lower case letters $x, y, \ldots \in \mathcal{X}$, play dual roles as communication channels and variable names. More specifically we have the atomic actions

Transmit — $x(y)$ means input some variable along the channel x and call it y.

Receive — $\bar{x}y$ means output the variable y along the channel x.

A *process*, which we denote by upper case letters $P, Q, \ldots \in \mathcal{P}$, is the only other entity we have in the π-calculus. A process is built by combining processes and atomic actions by the following syntax

$$P ::= \sum_{i \in I} \pi_i.P_i \big| P|Q \big| !P \big| (vx)P$$

where I is a finite indexing set (in the case $I = \emptyset$ we write the sum as 0).

Here $\pi.P$ means do an atomic action π and then do P in sequence, where $P + Q$ means do either P or Q (but not both) and $\sum_{i \in I} \pi_i.P_i$ represents selecting one out of many processes. $P|Q$, pronounced "P bar Q" denotes two processes being run concurrently and $!P$, pronounced "bang P" is short hand for $P|P|P....$ Finally, $(vx)P$, read "new x in P" restricts the usage of x to the process P. If we limit $\sum \pi_i.P_i$ to only type of communication we have what we call a *normal process* denoted by capital letters $M, N, \ldots \in \mathcal{N}$ and given by

$$N := \pi.P|0|M + N$$

3.1.1 Structural Congruence

Now that we have a defined processes it is natural ask when two processes are equivalent. This is what we call *structural congruence* and it is given by first defining *free names* $\texttt{fn}(P)$ and *bound names* $\texttt{bn}(P)$ of a process P in the usual way. For example

$$\texttt{bn}(x(y)) = \{y\}\ ,\ \texttt{fn}(x(y)) = \{x\}$$

$$\texttt{bn}(\bar{x}y) = \emptyset\ ,\ \texttt{fn}(\bar{x}y) = \{x, y\}$$

We define the structural congruence to be the $\subseteq$-least congruence relation over $\mathcal{P}$ such that

1. Processes are congruent if they only differ by a change of bound names
2. The semi-group $(\mathcal{N}, \equiv, +, 0)$ is a commutative monoid.
3. The semi-group $(\mathcal{P}, \equiv, |, 0)$ is a commutative monoid.
4. $!P \equiv P|!P$
5. $(vx)0 \equiv 0$, $(vx)(vy)P \equiv (vy)(vx)P$
6. If $x \notin \texttt{fn}(P)$ then $(vx)(P|Q) \equiv P|(vx)Q$

Consequence $(vx)P \equiv P$ when $x \notin \texttt{fn}(P)$

$$\begin{aligned}
(vx)P &\equiv (vx)P|0 && \text{(by 3)}\\
&\equiv p|(vx)0 && \text{(by 6)}\\
&\equiv p|0 && \text{(by 5)}\\
&\equiv p && \text{(by 3)}
\end{aligned}$$

3.1.2 Reduction rules

We now finally discuss the most important part of π-calculus; simplifying a process by reducing it. The *reduction relation* $\rightarrow$ over processes; $P \rightarrow P'$ means that P can be transfomed into P' by a single computational step. The main reduction rule which captures the ability of processes to communicate through channels is the following:

$$\texttt{COMM} : (\ldots + x(y).P)\big|(\ldots + \bar{x}z.Q) \rightarrow P(z/y)\big|Q$$

`COMM` is the only axiom for $\rightarrow$; otherwise we only have the inference rules:

- If $P \equiv P'$ then also $P\big|Q \equiv P'\big|Q$. Which means that parallel composition does not inhibit computation. It is formally given as

$$\texttt{PAR} : \frac{P \rightarrow P'}{P\big|Q \rightarrow P'\big|Q}$$

- If $P \equiv P'$, then also $(vx)P \equiv (vx)P'$ which ensures that computation can proceed underneath a restriction. It is formally given as

$$\texttt{RES} : \frac{P \rightarrow P'}{(vx)P \rightarrow (vx)P'}$$

- If $Q \equiv P$ and $P \rightarrow P'$ and $P' \equiv Q'$, then also $Q \rightarrow Q'$. In this rule, called the *structural* rule, $\equiv$ denotes the *structural congruence*, which says that concurrency is commutative and associative. It is the leas congruence such that:

  - $Q\big|Q' \equiv Q'\big|Q$, $Q\big|\left(Q'\big|R\right) \equiv \left(Q\big|Q'\right)\big|R$ and $Q\big|0 \equiv Q$
  - $(vx)(vy)Q = (vy)(vx)Q$
  - $(vx)\left(Q\big|Q'\right) \equiv (vx)\left(Q\big|Q'\right)$, provided x is not a free name in Q'

Which is formally given as

$$\texttt{STRUCT} : \frac{Q \equiv P \;\; P \to P' \;\; P' \equiv Q'}{Q \to Q'}$$

3.1.3 Examples

Example 1 By the monoid property $0 \vert P \equiv P \vert 0 \equiv p$ we have trivially that

$$0 \vert 0 \equiv 0$$

Example 2 Using `COMM` we demonstrate a simple input and output channel in a concurrent system.

$$\bar{c}.P \vert c.Q \to P \vert Q$$

Example 3 An example of sending names through a channel.

$$\begin{aligned} \bar{c}(x).P \vert c(y).\bar{y}.Q \vert x.R &\to P \vert \bar{x}.Q \vert x.R \\ &\to P \vert Q \vert R \end{aligned}$$

Example 4 Using the ! (bang) operation we demonstrate how a system may reduce in multiple ways.

1.

$$\begin{aligned} \bar{x}.P \vert (!x + z).Q \vert \bar{z}.R &\to P \vert Q \vert !(x + z).Q \vert \bar{z}.R \\ &\to P \vert Q \vert Q \vert !(x + z).Q \vert R \end{aligned}$$

2.

$$\bar{x}.P\big|(!x+z).Q\big|\bar{z}.R \to \bar{x}.P\big|Q\big|!(x+z).Q\big|$$
$$\to P\big|Q\big|Q\big|!(x+z).Q\big|R$$

Example 5 An example of a non-reducible (deadlocked) system

$$x.\bar{y}\big|y.\bar{x}$$

3.2 Correctness properties

Now that we can model concurrent systems using the π-calculus, we can define correctness properties on such systems.

In the Requirements section of this report, we have already described what deadlocks and result variations are. We now formally state the property of being deadlock-free and the property of being result variation-free.

Before we can define a deadlock-free system, let us define what a deadlocked process is.

Finite process definition A process P is said to be *finite* if it is not defined recursively and does not use the replication operator ("!"). P is not recursively defined if it can be driven from the π-calculus grammar in a finite number of steps.

Let $\Pi(\mathcal{N})$ be the set of all terms of the π-calculus with names from $\mathcal{N}$.

Nonreducible definition A process $P \in \Pi(\mathcal{N})$ is *nonreducible* if

$$\forall X \in \Pi(\mathcal{N}), P \not\longrightarrow X.$$

Note that $P \not\longrightarrow X$ is short for $\neg(P \longrightarrow X)$.

Deadlock definition A finite process P is in a *deadlocked* state if P is *nonreducible*.

By the above definition 0 and all processes congruent to it are clearly in a *deadlocked* state. For our purposes, 0 represents "completion" of a task; so, 0 represents an *acceptable* deadlock.

Eventual reduction ($\longrightarrow^*$) definition Let $\longrightarrow^* \subseteq \Pi(\mathcal{N}) \times \Pi(\mathcal{N})$ such that $P \longrightarrow^* R$ if process P eventually reduces to R via a finite number of reductions; that is, for some $n \in \mathbb{N}$,

$P \longrightarrow P_1 \longrightarrow \cdots \longrightarrow P_n$ and $P_n \equiv R$.

Deadlocked result set The *deadlocked result set* of a process P written as $\Delta(P)$, is the set of all nonreducible processes that P will eventually reduces to. More formally,

$$\Delta(P) = \{X | P \longrightarrow^* X \text{ and } X \text{ is nonreducible}\}$$

For our purposes, a system defined using a process P is result variation and deadlock-free if

$$A \in \Delta(P) \text{ and } B \in \Delta(P) \implies A \equiv B, \text{and } R \in \Delta(P),$$

where R is a predefined *expected* resulting system. We talk more about such resulting systems in our implementation section.

We say, a system is correctly parallelized if it is free of result variations and deadlocks.

We can now theoretically detect correct or incorrect parallelization in a concurrent system modelled by the π-calculus. The next step is to translate the Baklava code-sets to π-calculus processes.

3.3 Specification of semantics of Baklava

The translation scheme of Baklava to π-calculus processes is a formal specification of the semantics of Baklava. This formal specification is our assumption of how Baklava is going to be implemented on a cluster.

For the sake of modularity, we've divided this translation scheme into two separate schemes. First, we translate a given Baklava code-set to an *abstract* Baklava code-set. The abstract Baklava, is a simplified version of the Baklava language designed to be somewhat more independent of the type of the cluster Baklava is going to be ran on. Second, we represent the generated abstract Baklava code-set as a π-calculus process. See that the second translation is independent of the type of Baklava (or the type of the cluster Baklava is going to be ran on.)

We define abstract Baklava as a data structure in a module of its own, along with a definition for a class of all *abstractable* Baklava types (and clusters.)

In separate modules that will be importing the above module, we define an instance of abstractable Baklava for each type of Baklava. This would mean that we'd come up with a translation scheme from that type of Baklava to abstract Baklava.

3.4 Rewriting Logic

Now that we have established correctness properties, we need a logical framework to reason such correctness properties on. Rewriting logic is a natural choice for such a framework.

Rewriting Logic is a reflective logical and semantic framework [MOM93].

A rewrite theory is a tuple $\mathcal{R} = (\Sigma, E, L_R, R)$ where the pair (Σ, E) is an Σ-theory (i.e. Σ is an equational signature and E is a set of Σ-equations,) L_R is a set of labels for such rewrite rules, and R is a set of rewrite rules (axioms) of the form $l : t \longrightarrow t'$ where $l \in L_R$ and $t, t' \in T_{\Sigma,E}(X)$.

$T_{\Sigma,E}(X)$ is the set of all E-equivalence classes of terms on the set of variables X.

Provable sentence in such a rewrite theory $\mathcal{R}$, written $\mathcal{R} \vdash [t]_E \longrightarrow [t']_E$, where t and t' are Σ-terms, and can only be obtained using the following inference rules [MJeM01]:

Reflexivity For each $[t] \in T_{\Sigma,E}(X)$ we have $\dfrac{}{[t] \longrightarrow [t]}$.

Congruence For each $f \in \Sigma_n$ where $n \in \mathbb{N}$, we have

$$\frac{[t_1] \longrightarrow [t'_1] \cdots [t_n] \longrightarrow [t'_n]}{[f(t_1, \cdots, t_n)] \longrightarrow [f(t'_1, \cdots, t'_n)]}.$$

Replacement For each rule $l : [t(x_1, \cdots, x_n)] \longrightarrow [t'(x_1, \cdots, x_n)]$ in R, we have

$$\frac{[w_1] \longrightarrow [w'_1] \cdots [w_n] \longrightarrow [w'_n]}{[t(\bar{w}/\bar{x})] \longrightarrow [t'(\bar{w'}/\bar{x})]}.$$

Transitivity

$$\frac{[t_1] \longrightarrow [t_2] \quad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}.$$

In Rewriting Logic, a rewrite rule $t \longrightarrow t'$ has a computational as well as logical meaning.

Logically, it is an inference rule

$$\frac{t}{t'};$$

that is, if we have the premise t, we can draw the conclusion t'. Computationally, it roughly means that in a concurrent system can go from a state t to the state t'.

Concurrent languages (among many other types of languages) can be naturually expressed as a rewrite theory in the semantical framework of rewriting logic.

Using the π-calculus "COMM" axiom as a rewrite rule, we developed a model for the π-calculus in rewriting logic. We eventually found a better model [TSMO02] for the π-calculus in rewriting logic as well as the implementation of the model in Maude (a language based on rewriting logic.) There are also other model checkers available for the π-calculus, such as MMC [YRS04], most of which can probably be easily made to work with our π-calculus output.

4 Implementation

There are two main components to our implementation. The first component is the translation scheme from Baklava code-sets to the π-calculus and it consists of a module **`AbstractBaklava`** that defines the structure of abstract Baklava, as well as classes that that define what *abstractability* means. This module is imported by each abstraction implementation for a type of Baklava. For example, we defined the abstraction for G5 Baklavas in the *AbstractableG5Baklava* module. These were implemented using Haskell.

The second component of our implementation are the reduction algorithms. While the reduction axiom of the π-calculus was easy to implement in Haskell, the inference rules where more difficult to deal with than expected. As a result, we eventually decided to try implementing the inference rules as a rewrite theory in Maude.

Due to a design error, our reduction system in Maude was insufficient. That is, our reduction rules implementation did not guarantee to capture every possible reduction. Fortunatly, there was a much better model and thereby a much better implementation of the π-calculus semantics in Maude. We eventually found more tools for verification of systems modelled in the π-calculus;

4.1 Abstraction

We implement two modules to define the more abstract data structures, and the functions to work with such data structures.

4.1.1 Baklava Abstraction

The **`AbstractBaklava`** module defines data structures for *abstract* Baklava and thereby the abstract cluster that it will work on.

```
module AbstractBaklava where
import qualified Data.Map as Map
```

In our first attempt, we represented the abstract cluster as an instance of the general cluster classes (see **`Node`**, **`CPU`**, **`Status`**, **`Reference`**, and **`Computation`** classes of the Geometer **`Cluster`** module). At first, we also tried making **`AbstractBaklava`** a subtype of the more general **`Onion`** data type. At the end, we decided to represent both the abstract cluster and Baklava as their own data types, without relying on the types already defined as part of the Geometer.

It was decided that this module should remain completely independent of the Geometer modules. As discussed in the design section, the definition of abstract Baklava and its translation to the π-calculusshould not be effected by changes to the Geometer. However, the functions that abstract a given Baklava (as defined in the **`AbstractableG5Baklava`** module for the **`G5Baklava`**) do rely on data structures provided by the Geometer.

We now define the structure of an abstract Baklava. The provided descriptions here are non-normative; the formal specification *is* the code that translates this to the π-calculus.

```
data AbstractBaklava =
```

- Globally record **`Status`** as the status of the node identified by **`NodeID`**

```
SendStatus NodeID Status
```

- Wait for the globally recorded status of the node identified by **`NodeID`** to match **`Status`** before executing the next abstract Baklava in the queue.

```
| WaitStatus NodeID Status
```

- Wait for the globally recorded status of the node identified by **`NodeID`** to match **`Status`** before executing the abstract Baklavas in **`BaklavaQueue`**.

```
| GuardStatus NodeID Status BaklavaQueue
```

- Send data from the global reference **`Reference`** to the node identified by **`NodeID`**.

```
| SendData Reference NodeID
```

- Receive data into the global reference **`Reference`** and execute the second argument on success as an abstract Baklava, or the third argument otherwise.

```
| ReceiveData Reference AbstractBaklava AbstractBaklava
```

- Copy data from the global reference first argument to the global reference second argument.

```
| CopyData Reference Reference
```

- Send **`BaklavaQueue`** to the node identified by **`NodeID`** to be placed on its queue.

```
| SendCommand NodeID BaklavaQueue
```

- Execute **`Computation`** and set the appropriate **`Status`**.

```
| Execute Computation [Status]
```

Note some of the changes in the above data structure against the one defined by the Geometer. For instance, we define statuses and references completely globally rather than locally. All reference calls have to be made globally.

```
type BaklavaQueue = [AbstractBaklava]
```

A baklava queue is executed as a sequence. That is, given a **BaklavaQueue** $[x_0, \ldots, x_n]$, for all $i \in \{1, \ldots, n\}$, x_i can only be executed if execution of x_{i-1} has been completed.

We define each component of an abstract cluster, **Node**s, **Status**es, etc; and then we define a cluster as a collection of such components.

See that some of these components don't just define the cluster's structure, they also define its *state*.

```
data Node = Node {nodeBaklavaQueue :: BaklavaQueue}
```

A node is identified by a **NodeID**. That is, a cluster will consist of a mapping of such **NodeID**s to **Node**s (which represent the current state of a node.)

```
data NodeID = NodeID Integer deriving (Eq, Ord)
```

A **Status** is used to define the current status state of **Node**s for example.

```
data Status = Status {statusStatus :: Integer}
```

A **Reference** globally identifies an *storage object*. A **LocalReference** locally identifies an *storage object*, usually on a **Node**.

```
data Reference = Reference {referenceNode :: NodeID, referenceReference :: LocalReference}
data LocalReference = LocalReference Integer
```

A **Computation** represents a *signature* for a function mapping **Reference**s in **computationInput** to **Reference**s in **computationOutput**. That is, it is a data structure that describes which references will be modified after a given computation, and also describes the references it depends on.

```
data Computation = Computation {computationInput :: [Reference], computationOutput :: [Reference]}
```

A **`Cluster`** is the representation of an abstract cluster's state. It consists of a collection of **`Node`**s which are mapped from **`NodeID`**s, which uniquely identify them.

```
data Cluster = Cluster {
        clusterNodes :: ClusterNodes
        }

type ClusterNodes = Map.Map NodeID Node
```

We now make our cluster, its components, and abstract Baklava showable. While this is used for debugging, its more important purpose is to construct names to be used in the π-calculustranslations.

```
instance Show Cluster where
        show Cluster {clusterNodes = a} = "Abstract Cluster = {Nodes " ++ (show a) ++ "}"

instance Show Node where
        show node = "Abstract Node {Queue=" ++ show (nodeBaklavaQueue node) ++ "}"

instance Show Status where
        show (Status a) = "STATUS#" ++ (show a)

instance Show NodeID where
        show (NodeID a) = "NODE#" ++ (show a)

instance Show Reference where
        show (Reference nodeid local) = "REF{" ++ show nodeid ++ "/" ++ show local ++ "}"

instance Show LocalReference where
        show (LocalReference a) = show a

instance Show Computation where
        show Computation{computationInput = i, computationOutput = o}
                = "COMP{" ++ show i ++ " → " ++ show o ++ "}"
```

Showing **AbstractBakava**s is only for development purposes. It is not meant to be used for π-calculusnamings.

```
instance Show AbstractBaklava where

        show (SendStatus nodeID status)
                = "Send␣Status␣{to␣" ++ (show nodeID)
                ++ "␣," ++ show status ++ "␣}."

        show (GuardStatus nodeid status baklavas)
                = "Gaurd␣{for␣" ++ (show status)
                ++ "␣on␣" ++ show nodeid ++ "␣,␣then␣do␣"
                ++ show baklavas ++ "␣}."

        show (WaitStatus nodeid status)
                = "Wait␣{for␣" ++ (show status) ++ "␣on␣" ++ show nodeid ++ "␣}."

        show (SendData from to)
                = "Send␣Data␣{from␣" ++ show from ++ "␣,␣to␣"
                ++ show to ++ "␣}."

        show (ReceiveData to _ _)
                = "Receive␣Data␣{into␣" ++ show to ++ "␣}."

        show (CopyData from to)
                = "Copy␣Data␣{from␣" ++ show from ++ "␣,␣to␣" ++ show to ++ "␣}."

        show (SendCommand to baklavas)
                = "Send␣Command␣{tell␣" ++ show to ++ "␣,␣to␣do␣"
                ++ show baklavas ++ "␣}."

        show (Execute comp status)
                = "Execute␣{function:" ++ show comp
                ++ "␣,␣then␣set␣status␣" ++ show status ++ "␣}."
```

AbstracableCluster is the class of clusters that are translated to **AbstractCluster**s.

This class isn't general enough to include all 'abstractable' clusters. It only includes those that can be translated through independent *node* and *status* translations. For example, a cluster whose *status* abstraction requires a knowledge about the *nodes* can't be in this class.

Abstraction of a cluster is the process of mapping its nodes to the abstract **ClusterNodes**.

```
class AbstractableCluster clusternodes where
        abstractNodes :: clusternodes → ClusterNodes
```

abstractNodes can usually be defined with one or two functors, one of which can be defined using the functor provided over **Data.Map.Map k** for any k (namely for **k** being clusternodes).

```
class AbstractableBaklava baklava where
        abstractBaklava :: NodeID → baklava → AbstractBaklava
```

4.1.2 G5 Abstraction as an Example

Now that we have defined what it means to be abstract via the **AbstractBaklava** module, we can define our first abstraction, the G5 abstraction.

The **AbstractableG5Baklava** module defines an abstraction of the G5 Baklava and the G5 Cluster.

```
module AbstractableG5Baklava where

import qualified Data.Map as Map
import qualified Geometer
import qualified G5Cluster
import qualified Onion
import AbstractBaklava
```

First, we make the G5 cluster abstractable by defining its abstraction.

```
instance AbstractableCluster (Map.Map G5Cluster.G5Node [Geometer.G5Onion]) where
```

Since the G5 cluster consists of nodes with two CPUs which can execute computations concurrently, in our abstraction, we let each CPU be represented by a node of its own.

We do this by mapping each node of the G5 cluster to two abstract nodes. That is, $node_i^{G5} \mapsto node_{2i}^{abstract}, node_{2i+1}^{abstract}$, where each abstract node represents a single computation thread on a single CPU of the G5 cluster. We put all the work of executing the initial Baklava queue assigned to the given G5 machine on the first CPU.

Of course, this assumption (our specification of Baklava), along with all the other assumptions can be changed to fit an implemented machine.

This abstraction is tricky, since both $node_{2i}^{abstract}$ and $node_{2i+1}^{abstraction}$ can share resources such as statuses and memory.

```
abstractNodes = Map.foldWithKey
                (λnode bakQ curmap
                        →       Map.insert (translateID node 0)
                                (Node {nodeBaklavaQueue = fmap
                                (abstractBaklava $ translateID node 0) bakQ}) $
                                Map.insert (translateID node 1)
                                (Node {nodeBaklavaQueue = []})
                                curmap
                )
                Map.empty
```

We now make the **`G5Onion`**, the G5 Baklava, abstractable. That is, we provide a scheme for translating G5 Baklava to abstract Baklava. This scheme, composed with the π-calculustranslation, specifies G5 Baklava.

The Baklava commands act on the node they are being executed on; so, for some transla-

tions we will need to know which node the Baklava command was going to be executed on (see for example the translation for **`Execute`**.) So we carry the identification of the node that the commands will be executed on through out our recursive translation. We refer to that node as the 'current node' from here on. Mostly, the 'current node' node will stay the same in our recursive calls; but, for commands that transfer the execution to another node (such as **`SendCommand`**,) we need to change our 'current node' to that node.

```
instance AbstractableBaklava Geometer.G5Onion where
```

- Since we are going to be sharing statuses, and since abstract Baklava treats memory and status as global objects, a status being sent to a $node_i^{G5}$ can be treated as being a status sent to $node_{2i}^{abstract}$ and not $node_{2i+1}^{abstract}$. So, two G5 CPUs will be sharing statuses on the first CPU. We also have to make sure **`WaitStatus`** and **`GuardStatus`** are also aware of this.

  This translation is not recursive, and so 'current node' isn't carried forward.

  ```
  abstractBaklava _
          Onion.SendStatus{Onion.oNode = tonode, Onion.oStat = g5status}
          = SendStatus (translateID tonode 0) (translateStatus g5status)
  ```

- Like the previous case, if either $node_{2i}^{abstract}$ $node_{2i+1}^{abstract}$ are guarding or waiting, they will check the status on $node_{2i}^{abstract}$.

  In the case of **`GuardStatus`**, we are also given a Baklava queue to guard. This will also need to be translated.

  Recall that $node_{2i}^{abstract}$ and $node_{2i+1}^{abstract}$ both have their status on the former node. So if we are guarding or waiting for an status on $node_{2i+1}^{abstract}$, we are really waiting for the status from $node_{2i+1}^{abstract}$; the **`roundDown`** function here takes care of this.

  The translation of **`GuardStatus`** is recursive, and so we need to carry forward the 'current node'. It will be carried unchanged, since the execution of the guarded Baklava queue will still be on the same node.

  ```
  abstractBaklava nodeid
  ```

```
        Onion.GuardStatus{Onion.oStat = g5status, Onion.oCmds = g5baklavas}
        = GuardStatus (roundDown nodeid) (translateStatus g5status)
        (fmap (abstractBaklava nodeid) g5baklavas)

abstractBaklava nodeid
        Onion.WaitStatus{Onion.oStat = g5status}
        = WaitStatus (roundDown nodeid) (translateStatus g5status)
```

- Sending and receiving data aren't recursive translations, so 'current node' isn't carried forward; however, 'current node' is used since two CPUs represented by seperate nodes actually share memory.

```
abstractBaklava nodeid
        Onion.SendData{
                Onion.oTo = G5Cluster.Ref toNode _,
                Onion.oFrom = G5Cluster.Ref _ fromRef}
        = SendData (Reference (roundDown nodeid)
        $ LocalReference
        $ translateLocalReference fromRef) (translateID toNode 0)

abstractBaklava nodeid
        Onion.ReceiveData{
                Onion.oTo = G5Cluster.Ref _ localRef,
                Onion.oOk = statok, Onion.oErr = staterr}
        = ReceiveData
                (Reference (roundDown nodeid)
                $ LocalReference $ translateLocalReference localRef)
                (SendStatus (roundDown nodeid) $ translateStatus statok)
                (SendStatus (roundDown nodeid) $ translateStatus staterr)
```

- Copy data is a completely global command. That means, unlike its counterpart command in the Geometer it is not restricted to make copies of only local references.

```
abstractBaklava nodeid
```

```
Onion.CopyData {Onion.oTo = G5Cluster.Ref _ toRef,
Onion.oFrom = G5Cluster.Ref _ fromRef}
= CopyData
        (Reference nodeid $ LocalReference $ translateLocalReference fromRef)
        (Reference nodeid $ LocalReference $ translateLocalReference toRef)
```

- Translation of commands that contain Baklava queues is recursive, so it needs to pass a **`NodeID`** to the recursive translation procedure.

  In case of **`SendCmd`**, this **`NodeID`** won't be that of the current node, but the new node that the queue will be passed to.

```
abstractBaklava nodeid
        Onion.SendCmd {Onion.oNode=node, Onion.oCmds=g5baklavas}
        = SendCommand
                (translateID node 0)
                (fmap (abstractBaklava $ translateID node 0) g5baklavas)
```

- The next translation, that of **`Execute`**, is the real reason why we needed to carry through the current node's **`NodeID`**.

  The **`Execute`** defined in the Geometer specifies which CPU the computations should be executed on. For G5, if we are asked to execute on CPU a, and if we are already on it, we just translate as expected, ignoring the CPU. If we are however asked to execute on CPU b, we need to wrap the execution with a *send* command to send the execution request to the next odd node.

```
abstractBaklava nodeid
        Onion.Execute {Onion.oCpu = G5Cluster.G5one,
        Onion.oComps = comp, Onion.oStats = stats}
        = Execute (translateComputation comp nodeid)
        (fmap translateStatus stats)


abstractBaklava nodeid
        Onion.Execute {Onion.oCpu = G5Cluster.G5two,
```

```
                    Onion.oComps = comp, Onion.oStats = stats}
                    = SendCommand (NodeID $ (λ(NodeID x)→x+1) nodeid)
                    [Execute (translateComputation comp nodeid)
                    (fmap translateStatus stats)]

translateComputation :: Geometer.G5Comp → NodeID → Computation
```

Some of the G5 computations defined in the Geometer are abstracted as follows:

- StoreImage $R\ x\ y : \{R\} \mapsto \emptyset$

```
translateComputation
        (Geometer.G5StoreImage (G5Cluster.Ref _ ref) _ _)
        nodeid
        = Computation
        [Reference nodeid $ LocalReference $ translateLocalReference ref]
        []
```

- DisplayImage $R\ x\ y : \emptyset \mapsto \{R\}$

```
translateComputation
        (Geometer.G5DisplayImage (G5Cluster.Ref _ ref) _ _)
        nodeid
        = Computation
        []
        [Reference nodeid $ LocalReference $ translateLocalReference ref]
```

- FT $R_1\ R_2\ x \ldots : \{R_1\} \mapsto \{R_2\}$ *(Fourier transform)*

```
translateComputation
        (Geometer.G5FT2D (G5Cluster.Ref _ ref1) (G5Cluster.Ref _ ref2) _ _)
        nodeid
        = Computation
        [Reference nodeid $ LocalReference $ translateLocalReference ref1]
        [Reference nodeid $ LocalReference $ translateLocalReference ref2]
```

- Not yet translated

  With our test function, we only needed a translation of **`G5FT2d`**.

```
translateComputation a _ = error $ "No translation defined for " ++ show a ++ "."
```

Below, we define the *help* functions used above. Most of these functions translate cluster components.

```
roundDown :: NodeID → NodeID
roundDown (NodeID x) | mod x 2 == 0 = NodeID x
roundDown (NodeID x) | mod x 2 == 1 = NodeID $ x-1

translateLocalReference :: Int → Integer
translateLocalReference = toEnum

translateReference :: G5Cluster.G5Ref → Reference
translateReference  (G5Cluster.Ref nodeid localref)
        = Reference {referenceNode = translateID nodeid 0,
        referenceReference = LocalReference $ translateLocalReference localref}

translateID :: G5Cluster.G5Node → Integer → NodeID
translateID (G5Cluster.G5Node id) a = NodeID ((toEnum id) * 2 + a)
```

The translation of **`Status`** requires that we encode all the components of the status (such as counter and status number) into a single integer. We do this by assuming that there are at most 1024 nodes on the cluster, and that the counter will never go higher than 1024.

```
translateStatus :: G5Cluster.G5Status G5Cluster.G5Node → Status
translateStatus G5Cluster.MRStatus
        {G5Cluster.mrstNode = node, G5Cluster.mrstStatus = status,
        G5Cluster.mrstCount = count} =
        Status {statusStatus = ((λ(NodeID x) → x * 1024 * 1024)
        (translateID node 0))
        + ((toEnum count) * 1024) + (toEnum status) }
```

4.2 The π-calculus translation

Next, we attempt to translate abstract Baklava code-sets to a π-calculus process. This is the final step in our translation scheme. First, we must define a data structure for the π-calculus.

```
module AbstractBaklava2Pi where


import AbstractBaklava
import qualified Data.Map as Map
```

Our definition of a process does not include the replication operator. We do not need the replication operator for most parts. The only components of the system that will need to use the replication operator are those that can be hard-wired into the reduction system (such as the model for retreiving and updating memory cells and statuses.)

```
data Process =
        Bar [Process]
        | Sum [Process]
        | Pi PiType Name [Name] Process
        | Zero


data PiType = Input | Output
```

Names are represented as strings. This makes both the formation of the translation scheme and debugging the outputted system a lot easier. If needs be, we can easily map the set of **`String`** names to **`Integer`**s.

```
data Name = Name String
```

We use the **`Show`** class to express our π-calculus data structure to notation readable by other verifiers as needed.

```
instance Show Process where
        show (Bar []) = ""
```

```
        show (Bar [x]) = show x
        show (Bar (x:xs)) = "(" ++ (show x) ++ ")|" ++ (show $ Bar xs)

        show (Sum []) = ""
        show (Sum [x]) = show x
        show (Sum (x:xs)) = "(" ++ (show x) ++ ")+" ++ (show $ Sum xs)

        show (Pi Input channel vars p) =
        show channel ++ "<" ++ show vars ++ ">." ++ show p
        show (Pi Output channel vars p)
        = show channel ++ "(" ++ show vars ++ ")." ++ show p

        show Zero = "0"
instance Show Name where
        show (Name name) = name

        showList [] _ = ""
        showList [name] _ = show name
        showList (name:names) _ = show name ++ ",␣" ++ show names
```

First, we need to convert the abstract cluster to a process which consists of all the nodes and the misc. systems running concurrently

$$System := Node_1|\cdots|Node_n|M,$$

where M represents the misc. systems (memory and status update and retrieve.) While M can be modelled in the π-calculus, due to its size (one concurrent process for each cell of memory) it greatly improves performance (without loosing formality since its implementation can be verified manually) and use of resources.

```
abstractCluster2Pi :: Cluster → Process
abstractCluster2Pi Cluster {clusterNodes = nodes}
```

```
        = Bar $ Map.elems $ Map.mapWithKey (abstractNode2Pi) nodes
```

Translation of each node to a process is simply folding its Baklava queue using the function which translates each Baklava appropriately. See that this Baklava translation function also takes as input a process. This process is the computed process so far. Whatever the translation function does, it'll append to this process.

```
abstractNode2Pi :: NodeID → Node → Process
abstractNode2Pi nodeid Node {nodeBaklavaQueue = []}
        = Zero
abstractNode2Pi nodeid Node {nodeBaklavaQueue = b:bs}
        = (abstractBaklava2Pi nodeid b $ abstractNode2Pi nodeid Node{nodeBaklavaQueue=bs})
```

Translation of Baklava to π-calculusprocesses requires knowledge of the 'current node' id. This is because certain translations might be from Baklava commands that act on the 'current node' (such as Execute,) and we will need to model that with the current node id in mind.

```
abstractBaklava2Pi :: NodeID → AbstractBaklava → Process → Process
```

Sending statuses is the process of recording a status for a certain node on a certain node. The three components of the status (the node, status value, and status counter) are all enconded into a single entity *Status* (Integer in this case.) *Showing* such a status guarantees unique names for distinct statuses, and same name for the same statuses.

At the end, our global status table is the mapping of node IDs to statuses. This table and the operations on it are defined in M. To see that modelling such an M is possible, see [Tur96] in which Turner builds a lot of usable machine and cluster structures within the π-calculus.

```
abstractBaklava2Pi _ (SendStatus (NodeID nodeid) status) p
        = Pi Output (Name "Status") [Name $ show nodeid, Name $ show status] p


abstractBaklava2Pi _ (SendData ref (NodeID nodeid)) p
```

```
        = Pi Output (Name "RequestMemory") [Name $ show ref] (
        Pi Input (Name $ "ReadMemory" ++ show ref) [Name "_x"] (
        Pi Output (Name $ "DataTransfer" ++ show nodeid) [Name "_x"] p
        ))
```

Our most challenging translation is probably the translation of the **`Execute`** command. This is because we need to properly model the semantics of a *computation* in order to be able to detect result invariations. That is, we suppose every computation $C : [R_1, \cdots, R_n] \mapsto [R'_1, \cdots, R'_m]$ uses the values from reference $R_1, \cdots, R_n$ to unpredictably write to references $R'_1, \cdots, R'_m$; that is, we don't want to know what a computation actually does to the memory, we just need to know what it modifies and what it depends on.

For *each* input $R_1, \cdots, R_n$ of the given computation, we will *append* the value of that reference to each of the outputs in $R'_1, \cdots, R'_m$. So that's a total of n memory reads and $m \times n$ memory appends.

```
abstractBaklava2Pi _ (Execute Computation{computationInput=i,computationOutput=o} _) p
        = foldr
        (λx p1→Pi Output (Name "RequestMemory") [Name $ show x] (
                Pi Input (Name "ReadMemory") [Name $ "_x" ++ show x] p1 (
                foldr
                (λx p2→Pi Output (Name "AppendMemory")
                [Name $ show x, Name $ "_x" ++ show i] p2)
                Zero
                o)
        ))
        p i
```

Some abstract Baklava commands are still not translated here.

```
abstractBaklava2Pi _ x _ = error $ "Errr... what is " ++ show x
```

5 Evaluation and Future Work

When we started this project, we believed most of the work would be around creating an automated proving system to prove theories formed about our systems. While we ended up spending more time on reading about and implementing model checking systems as we did reading about and doing formal specification of Baklava, it would seem as if at the end, we obtained more visible results from our formal specification of Baklava.

In the future, one could use the same correctness properties formed in this paper as part of invariants to be used to formally verify the correctness of the Geometer. While verifying the invariants against the Geometer's Haskell code might prove difficult, verifying it against a translated version which generates π-calculus models instead might be easier.

If we are to continue verifying Baklavas on-the-fly, rather than verifying the Geometer, some future work in creating an optimized verification system tailored to our needs (as opposed to the generic π-calculusverifiers and model checkers) could be useful.

6 References

[ACHZ05] Rebecca Alsop, Zhi Cao, Mengyu Hu, and Yike Zang. Cluster safe powerdown and runtime system. 2005.

[Ana05] Christohper Anand. 2005.

[Hol91] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil93] Robin Milner. The polyadic π-calculus: A tutorial. 1993.

[Mil99] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[MJeM01] N. Mart, Jos, and e Meseguer. Rewriting logic: Roadmap and bibliography, 2001.

[MOM93] Narciso Marti-Oliet and Jose Meseguer. Rewriting logic as a logical and semantic framework, 1993.

[Pan04] Jun Pang. *Formal Verification of Distributed Systems*. PhD thesis, Vrije Universiteit Amsterdam, August 2004.

[TSMO02] P. Thati, K. Sen, and N. Mart-Oliet. An executable specification of asynchronous π-calculus semantics and may testing in maude, 2002.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, 1996.

[Wik06] Wikipedia. Formal verification — wikipedia, the free encyclopedia, 2006. [Online; accessed 2005-2006].

[YRS04] Ping Yang, C. R. Ramakrishnan, and Scott A. Smolka. A logical encoding of the π-calculus: model checking mobile processes using tabled resolution. *Int. J. Softw. Tools Technol. Transf.*, 6(1):38–66, 2004.