Advanced Control Flow in Java Card Programming

Peng Li Steve Zdancewic
University of Pennsylvania
{lipeng,stevez}@cis.upenn.edu

ABSTRACT

Java Card technology simplifies the development of smart card applications by providing a high-level programming language similar to Java. However, the master-slave programming model used in current Java Card platform creates control flow difficulties when writing complex card programs, making it inconvenient, tedious, and error-prone to implement Java Card applications. This paper examines these drawbacks of the master-slave model and proposes a concurrent thread model for developing future Java Card programs, which is much closer to conventional Java network programming. This paper also presents a code translation algorithm and a corresponding tool that makes it possible to write card programs in the concurrent thread model without losing compatibility with the existing Java Card API.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Languages Constructs and Features—*Control Structures*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Structured Programming*

General Terms

Languages

Keywords

Java Card, Continuation, CPS, Control Flow, Smart Card, Trampolined Style

1. INTRODUCTION

Java Card technology provides a subset of the Java programming language as well as the runtime environment for smart card programming, making it possible for developers to program the smart card using a modern high-level language.

The design of Java Card is restricted by the time and space constraints of the smart card, but it also lacks support for some basic software engineering principles. As others have pointed out, the current Java Card implementation is unsatisfactory in a sense that it encourages a low level programming style [6].

In this paper, we focus on the control flow issues in Java Card programming. To avoid confusion, it is necessary to make a distinction between *communication models* and *programming models*. In the existing design of Java Card, the command-response communication model between the card and the host naively led to the master-slave programming model, in which the host application always actively calls the services on the card. This programming model suffices for simple applications such as maintaining a counter or storing a secret piece of data, but the complexity of control flow increases when the card program requires more interaction between the host and the card. The control flow on the card is usually modeled as a state machine and it has to be hand-coded in ad hoc ways by the programmer, which is usually tedious and error prone.

We argue that the design of the high-level programming interface should not be restricted by the underlying communication model. We propose the concurrent thread model as a better design choice for the Java Card programming interface. In this model, the host program and the card program run in separate threads and communicate with each other via symmetrical interfaces. The concurrent thread model enables the programmer to use language-based control-flow primitives over the whole session, relieving the programmer's burden of coding up state machines for managing control flow.

It is challenging to use the concurrent thread model in existing Java Card environments due to the lack of thread support in the Java Card virtual machine. Instead, we propose code translation as a way to support this programming model without losing compatibility with the existing Java Card environment. Card applications can be written in the concurrent thread model and automatically translated to master-slave style Java Card programs, while behaving the same with regard to the host application.

This code translation approach can be used as a temporary solution in transition to the concurrent thread model. We call for the support of thread management in the Java Card environment. As the smart card technology advances, it will be practical to have multi-threading on the smart card so that the concurrent programming model will prevail.

LCTES'04, June 11–13, 2004, Washington, DC, USA.

2. THE MASTER-SLAVE PROGRAMMING MODEL

In existing industry standards, smart cards communicate with host computers by using special data packets called APDUs (application protocol data units). The APDU-level communication between the card and the host computer is half-duplex, which means the APDU packets can be sent in both directions but not at the same time. The command-response communication model is used: the host sends a command APDU to the card and the card sends a response APDU back to the host. The command APDUs and the request APDUs alternate with each other on the communication channel.

Similar to the communication model, the existing Java Card platform uses the master-slave programming model between the host and the card, as illustrated in Figure 1. The host computer plays the active (master) role, repetitively performing the dialog of sending a command to the card and waiting for a response from the card, as if calling a method on the card. Applications running on Java smart cards are called applets. Once an applet is *installed* and *selected* on the card, it waits for commands from the host computer. The applet plays a passive (slave) role in the programming model. Upon arrival of a command APDU, the Java Card virtual machine delivers the APDU to the applet by invoking the `process` method of the applet. The applet then starts execution as if being called by the host computer, processes the command and creates a response APDU. When the applet returns the control to the VM, the response APDU is sent to the host computer.

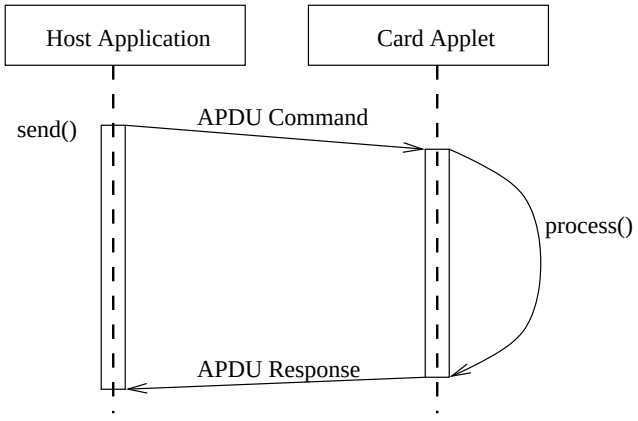


Figure 1: The master-slave model

The master-slave programming model naturally led to the implementation of RPC or RMI. In Java Card 2.2, unidirectional RMI is implemented so that the host can call the methods on the card. This is convenient for the programmers because it relieves the burden of marshaling and unmarshaling the data to and from the APDU.

3. STATE MACHINES FOR CONTROL FLOW

In this section we present the control flow problems in the master-slave programming model. We start by showing some situations where the control flow of a Java Card programs must be modeled as state machines and hand-coded in ad hoc ways by the programmer.

3.1 Card calling host methods

As the card programs become more complex, it becomes desirable to invoke method calls from the card to the terminal. Without bidirectional RMI, the programmers usually have to use a trick to invert the master-slave relationship as in Figure 2.

For example, the card wants to store a piece of temporary data on the host computer. The card calls the host by sending a special response APDU. The host API recognizes the special response. Instead of returning the response to the host application, it treats the APDU as a command from the card, process it by calling an event listener in the application, then send the response from the host in the format of a command APDU to the card. The card wakes up, examines its own state and resumes execution.

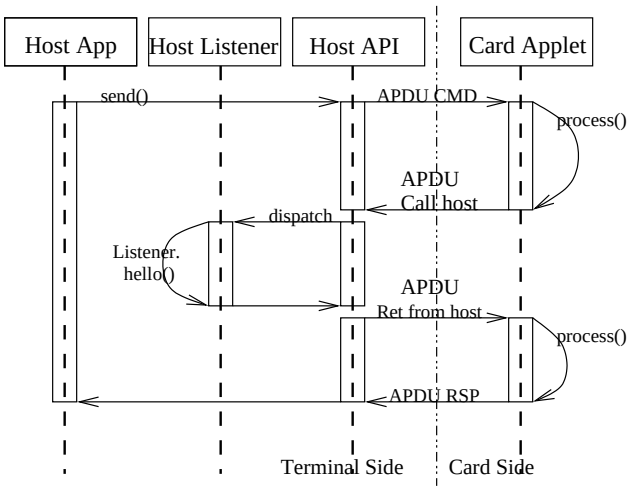


Figure 2: Card calling the host

From the host point of view, there is no technical difficulty of implementing bidirectional RMI. However, there is a challenge for the card programmer: in the current card programming model, a server side method "`hello()`" cannot be called by a single statement "`host.hello();`" on the card, because each time the card receives an APDU packet, it starts executing from a single entry point, the `process` method! The card program has to examine which state it was in, and jump to the continuation of its previous state. The card programmer must carefully manipulate the states to thread together different fragments of the code in order to complete a call to the host. In our experience, such a job is usually tedious and error-prone.

In such a program, the control flow can be modeled as a finite state machine. When the card processes an incoming APDU, it performs some computation and makes a transition to the next state.

3.2 Interaction with host in loops and method calls

As another example of problematic control flow, consider the following scenario. Suppose that during an authentication process, the card applet wants to verify the PIN number of the user. The user can make at most N attempts. As a standard Java programmer, one might naturally want to encapsulate the authentication process in a

method `authenticate()`, and use the following code in the program:

```
if (authenticate()) .... else ....
```

And, inside the implementation of `authenticate`, it is natural to use a loop to control the number of attempts:

```
public boolean authenticate() {
    for ( int i = 0; i < N; i++ )
       if (verify(host.promptPIN()))
          return true;
    return false;
}
```

Unfortunately, since the card cannot call the host method directly, the Java Card applet cannot be written like the above code. The control flow must be implemented by a state machine, as mentioned in the previous subsection. In doing so, we lose the following control flow features provided by the Java language:

- Method abstraction: All the interaction with the host must be controlled at the top-most level, because it is impossible to interact with the host inside a method call. The state machine has a flat structure: all the states are parallel to each other, without ordering and nested structures. This severely limits the application of some fundamental software engineering techniques like encapsulation and abstraction.
- Loop control: the loop control variable i in the above code is a local variable invisible to the outside of the loop. As we implement the loop in the state machine, it becomes part of the global state. The state of the machine will be a vector consisting of all the loop variables, which is difficult to maintain by the programmer.

3.3 Error handling

Let us consider error handling in the state machine. For each state q, the card is usually expecting a small number of inputs $i_1, \cdots, i_n$ as defined by the protocol. However, the programmer must be responsible for handling all the unexpected inputs. One should also note that the state machine is nondeterministic: for each combination of state q and input i, the program may have branches and exceptions — it may jump to several different states and produce different outputs. Take the following branch as an example:

```
if (state==S_MSG_REQ && input==MSG_REQ) {
   if (bad_apdu_format()) {
       state = S_APDU_ERR;
       output = MSG_APDU_ERR;
       .....
   } else {
       // some computation
       .....
       if (success) {
            state = S_MSG_REP;
            output = MSG_REP;
            ....
       } else {
            state = S_MSG_FAIL;
            output = MSG_FAIL;
            ....
```

The above code handles a request from the host computer and it succeeds or fails depending on the result of some computation. Besides these possibilities, it has to check for invalid data formats and handle such errors gracefully. The problem is that such error handling must be done everywhere in the state machine and there is a lot of duplication in the code.

In the Java programming language, there already exists a good solution for error handling: exceptions. Instead of handling the errors everywhere, the Java programmer can put an exception handler for a block of code. In the state machine, in order to share an exception handler for different states, the programmer can group some states together in the scope of exception handlers for them. This approach avoids rewriting some of the error handling code, but it also complicates the analysis of the state machine, making the control flow of the state machine hard to reason about.

3.4 Complexity of the state machine

The size of the state machine compounds the difficulty of programming the smart cards by hand using the master-slave model. Because of poor support for encapsulation and abstraction, developing and maintaining the card code is difficult. Suppose the state machine has t states, i input messages, each state is expecting j input messages on average, and there are e common errors that the programmer has to handle. The code size will be on the order of tje. The maintenance of the code has a high cost: the analysis of the control flow in the state machine has a complexity of tie because for each state, the programmer has to consider the behavior of the system under all possible inputs. When the card program scales up, the code will be difficult to maintain and the programmer may have to rely on the aid of automated tools.

4. THE CONCURRENT THREAD MODEL

4.1 Motivation

We have seen the various control flow problems in the master-slave programming model. All these problems are caused by the inherent limitations of the master-slave programming model, which is designed according to the underlying command-response communication model. We argue that, the programming interface of the smart card should not necessarily be modeled in the same way as its underlying communication, for several reasons:

- The programming interface should provide a better abstraction over low-level operations, for the same reasons we need high level programming languages. RMI can solve marshaling/typing issues, but it doesn't solve the control flow issues. Take the control flow in most high level programming languages as an example; the underlying machine language usually provides control flow primitives like `jmp`, `push` and `pop`, but higher level programming languages hide these details and provide advanced control flow mechanisms as loops, method calls and exception handling, discouraging the use of `goto` [2] which simulates the execution model of the underlying machine. These high-level control structures relieve the programmer's burden of managing control flow and better support software engineering techniques such as encapsulation, abstraction and code reuse.
- The communication model will evolve over time. The technical trend is that the smart cards will be more like

general-purpose computers. In the future, the smart card operating systems will probably have IP networking and multi-threading support built-in. Instead of using a specific programming interface for each communication model, it will be nice to use a general programming model that can adapt the changes of the underlying communication model.

4.2 The symmetric programming interface

We propose a symmetric programming model for both the host application and the card applet: the *concurrent thread model*. When the host interacts with the card applet, both sides have a running thread. The programming model is symmetric, in the sense that the card program is written in the same style as the host program just as if it were a stand alone application running on a general purpose computer. When the host starts a session with a card applet, the card launches a thread for the applet. In comparison to the master-slave model where the card applet is reactive and is always called by the host program, the applet thread is active. The programmer can assume that the applet thread keeps running during the whole session without yielding control to the host program. The card thread and the host program are two concurrent threads, and they communicate with each other through some communication mechanism provided by the platform API.

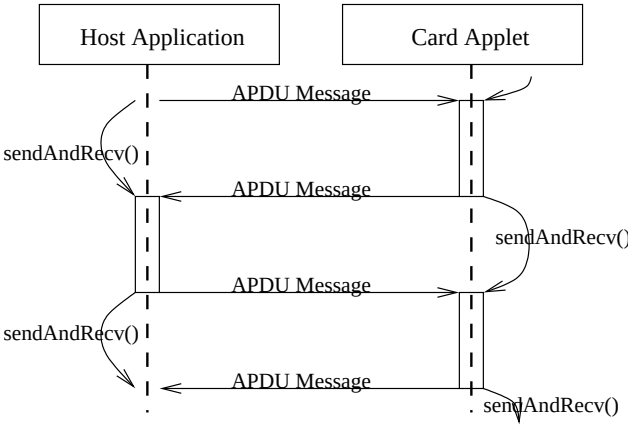


Figure 3: The Concurrent Thread Model

This programming model is independent of the underlying communication model. In the concurrent thread model, the programmer can write card applications in the same way as for conventional Java network programming, without learning much special knowledge about the card. To make it work with the existing half-duplex APDU protocol, the API only needs to provide a small number of communication primitives, such as `sendAndReceive()`, a method that sends an APDU to the other thread, waits for the reply, and returns the received APDU. Based on the APDU level communication functionality, we can easily build higher level communication interfaces in the API. As the smart cards become more powerful it will be possible to implement advanced networking protocols such as IPV6 on the card OS.

The benefit of using such a programming model on the Java Card platform is obvious. The scope of language-based control flow can now span across communication operations, eliminating the need of manually coding up control flow state machines. Although the Java Card language has various limitations inherent to smart card programming, the language-based control-flow features make little difference with the full Java language. Therefore, the symmetry of the concurrent thread model reduces the gap between general Java programming and Java Card programming. The developer can use the same programming interfaces and design patterns for both the host program and the card program.

4.3 Implementation challenges

To apply the concurrent thread programming model on Java Card, there are several challenges:

- The current Java Card framework is designed with the master-slave model. There is no threading support in the current Java Card. An overhaul is needed for the whole Java Card architecture.

- A number of existing API libraries need to be modified or rewritten, but backward compatibility is also needed for old applications.

We expect to have multi-threading functionalities in future Java Card operating system in order to perfectly support the concurrent thread model. But it may take a long time for the OS to evolve. The other possibility is to seek for solutions based on existing hardware and software environments. In the next section we will present code translation as a solution to apply the concurrent thread programming model without modifying the Java Card OS.

5. CODE TRANSLATION

The concurrent thread model greatly simplifies the control flow in smart card programming, but the existing Java Card environment does not support threads, limiting the application of this programming model.

In this section we present code translation as a way of supporting programming in the concurrent thread model on existing Java Card environments. Code translation works in the following way. First, the programmer writes a card applet in the concurrent thread model, using a source language similar to Java Card. Then, a code translator takes the source code as input, runs a translation algorithm and produces a state-machine-like program compatible with existing Java Card. Finally, the program generated by the code translator is compiled by existing Java Card tools and runs on the smart cards.

In other words, the code translator compiles card programs written in the concurrent thread model to programs in the master-slave model. It takes the source code as a specification of the control flow and simulates the process of hand coding the control flow state machines as mentioned in section 3.

5.1 The source program

Syntactically, the source language is the same as the Java Card language. The only difference is the programming model. From the programmer's point of view, a card applet is simply a runnable thread. The thread is launched by the card API when the communication session begins (i.e. the card applet is selected) and keeps running during the APDU communications.

The card applet thread communicates with the thread on the host computer by sending and receiving APDU packets. The communication is performed by calling a special method: `sendAndReceive(apdu)`, which sends the APDU to the other thread, waits for the responding APDU, and returns the new APDU object. When this method returns, the card thread continues running.

Other than the programming model and APDU communications, the source language has few differences from the Java Card language. The programmer can directly use the Java Card API in the same way as writing Java Card programs.

5.2 The target program

The code translator reads the source programs and produces standard Java Card programs that behave the same with regard to the host application. As we have discussed, the control flow of existing Java Card programs is reactive: every time the card receives an APDU packet, the card applet starts execution from a single entry point, i.e. `process()`, and yields control before sending the response APDU. Therefore, we have to break up the execution of the source program written in the concurrent thread model at the point it calls the `sendAndReceive()` communication primitive, and resume the execution when the card applet gains control again.

Without support of thread suspension and recovery, it would still be easy to implement the code translation if we were working with a language that supports continuations [1], such as SML/NJ [9] and Scheme [8]. We could simply save the current continuation at the place where communication is needed, and call the saved continuation when the card applet gains control again. Although Java does not support continuations, an ad hoc control mechanism can be used to manage the continuations of the card program:

- The source program is divided into basic blocks. The entry of each basic block in the source program corresponds to a control flow state. Each basic block is translated into a method definition, which includes all the statements in that basic block. The method returns the name of the next control state, i.e. the next basic block to be executed.

- The control stack and the local variables are managed by a user level mechanism. We need a helper class in the target program that simulates the control stack operations of the source program. Method calls, returns and exceptions are translated into explicit operations on the control stack.

  There should also be a mechanism for storing the local variables. All the statements should be translated so that the references to the local variables are redirected to custom-built mechanism. A simple solution is to translate local variables to fields in the class, forcing the resource allocation to be static. The trade-off is that we cannot translate recursive methods in this way. For recursive methods, local variables must be translated to operations on the control stack, which has an additional performance overhead.

- The engine of the control state machine is simply a loop. In each round of the loop, the engine examines its current state and calls the method corresponding to that state (a basic block). The method will execute all the statements in the corresponding basic block and return the name of the next state, which becomes the current state in the next round of the loop. Therefore, control is transferred from one block to another at the top-level loop by passing around the name of the control flow states. This kind of program organization is called the trampolined style [3].

- With all the mechanisms above, it is easy to save and restore the continuations of the source program: the continuation is simply the control state plus all the contents in the control stack. We only need to modify one branch of the state machine so that the card yields control when the communication primitive is called. When the card applet receives the next command APDU and regains control, it jumps into the main loop of the control state machine and the thread execution is resumed.

T	::=	$c \mid \mathsf{boolean}$	Type
C	::=	$\mathsf{class}\ c\ \mathsf{extends}\ c\ \{\overline{F}\ \overline{M}\}$	Class decl.
F	::=	$T\ f;$	Field decl.
M	::=	$T\ m(\overline{T\ x})\ s$	Method decl.
e	::=		Expressions
		x	*Variable*
		$\mid e.f$	*Field access*
		$\mid \mathsf{this}$	*This*
		$\mid \mathsf{true} \mid \mathsf{false}$	*Basic values*
		$\mid e\ \mathsf{op}\ e$	*Binary operation*
s	::=		Statements
		$e.f = e;$	*Field update*
		$\mid T\ x;$	*Local var. decl.*
		$\mid x = e;$	*Local var. update*
		$\mid x = \mathsf{new}\ c();$	*Object creation*
		$\mid x = e.m(\overline{e});$	*Method invocation*
		$\mid \mathsf{if}\ (e)\ s\ \mathsf{else}\ s$	*Conditional*
		$\mid \mathsf{while}\ (e)\ s$	*Loop*
		$\mid \mathsf{return}\ e;$	*Return*
		$\mid \{\overline{s}\}$	*Block*
		$\mid \mathsf{throw}\ e;$	*Raise an exception*
		$\mid \mathsf{try}\ s\ \mathsf{catch}(T\ x)\ s$	*Exception handling*

Figure 4: The source language

5.3 The code translation algorithm

This section presents a simplified version of our code translation algorithm. We restrict the source language to be similar to Feather Weight Java [7], with most of the control flow primitives: loops, method calls, and exceptions. The syntax of the source language is shown in Figure 4.

The translation algorithm is presented in Figure 5. $\mathbb{S}_{\mathsf{exp}}$, $\mathbb{S}_{\mathsf{stmt}}$ and $\mathbb{S}_{\mathsf{method}}$ refer to the translation function for expressions, statements and methods, respectively. For statements and methods, the output of the translation is a list of declarations in the target language. $\mathbb{F}(T\ x)$ means a field declaration and $\mathbb{B}$ means a method declaration which contains a basic block. Different kinds of basic blocks are defined in Figure 6.

For simplicity, we assume that the method calls are not recursive, which is usually practical for smart card programming. Therefore, the local variables can be directly translated to fields in the target class and the control stack only

Rules for translating expressions:

$$
\begin{array}{rcl}
\mathbb{S}_{\mathsf{exp}}[\![x]\!] & = & \mathbb{V}(\mathrm{x}) \\
\mathbb{S}_{\mathsf{exp}}[\![e.f]\!] & = & \mathbb{S}_{\mathsf{exp}}[\![e]\!].f \\
\mathbb{S}_{\mathsf{exp}}[\![\mathsf{this} \mid \mathsf{true} \mid \mathsf{false}]\!] & = & \mathsf{this} \mid \mathsf{true} \mid \mathsf{false} \\
\mathbb{S}_{\mathsf{exp}}[\![e_1 \ \mathsf{op}\ e_2]\!] & = & \mathbb{S}_{\mathsf{exp}}[\![e_1]\!] \ \ \mathsf{op} \ \ \mathbb{S}_{\mathsf{exp}}[\![e_2]\!]
\end{array}
$$

Rules for translating statements:

$$
\begin{array}{l}
\mathbb{S}_{\mathsf{stmt}}[\![e_1.f = e_2]\!]_\eta = \mathbb{B}_{\mathsf{seq}}(\mathbb{S}_{\mathsf{exp}}[\![e_1]\!].f = \mathbb{S}_{\mathsf{exp}}[\![e_2]\!])_\eta \\
\mathbb{S}_{\mathsf{stmt}}[\![T\ x]\!]_\eta = \mathbb{F}(T\ x) \\
\mathbb{S}_{\mathsf{stmt}}[\![x = e]\!]_\eta = \mathbb{B}_{\mathsf{seq}}(\mathbb{V}(x) = \mathbb{S}_{\mathsf{exp}}[\![e]\!])_\eta \\
\mathbb{S}_{\mathsf{stmt}}[\![x = \mathsf{new}\ C()]\!]_\eta = \mathbb{B}_{\mathsf{seq}}(\mathbb{V}(x) = \mathsf{new}\ C())_\eta \\
\mathbb{S}_{\mathsf{stmt}}[\![x = \mathsf{return}\ e]\!]_\eta = \mathbb{B}_{\mathsf{ret}}(\mathbb{S}_{\mathsf{exp}}[\![e]\!])_\eta \\
\mathbb{S}_{\mathsf{stmt}}[\![x = \mathsf{throw}\ e]\!]_\eta = \mathbb{B}_{\mathsf{throw}}(\mathbb{S}_{\mathsf{exp}}[\![e]\!])_\eta
\end{array}
$$

$$
\begin{array}{lll}
\mathbb{S}_{\mathsf{stmt}}[\![\{s_1, \ldots, s_m\}]\!]_\eta & & \eta = (b, n, x) \\
= \quad \mathbb{S}_{\mathsf{stmt}}[\![s_1]\!]_{\eta_1} & & \eta_1 = (b, t_1, x) \\
\quad \mathbb{S}_{\mathsf{stmt}}[\![s_2]\!]_{\eta_2} & & \eta_2 = (t_1, t_2, x) \\
\quad \ldots\ldots & \text{where} & \ldots \\
\quad \mathbb{S}_{\mathsf{stmt}}[\![s_m]\!]_{\eta_m} & & \eta_m = (t_{m-1}, n, x)
\end{array}
$$

$$
\begin{array}{lll}
\mathbb{S}_{\mathsf{stmt}}[\![\mathsf{if}(e)\ s_1\ \mathsf{else}\ s_2]\!]_\eta & & \\
= \quad \mathbb{B}_{\mathsf{branch}}(\mathbb{S}_{\mathsf{exp}}[\![e]\!], t_1, t_2)_\eta & & \eta = (b, n, x) \\
\quad \mathbb{S}_{\mathsf{stmt}}[\![s_1]\!]_{\eta_1} & \text{where} & \eta_1 = (t_1, n, x) \\
\quad \mathbb{S}_{\mathsf{stmt}}[\![s_2]\!]_{\eta_2} & & \eta_2 = (t_2, n, x)
\end{array}
$$

$$
\begin{array}{lll}
\mathbb{S}_{\mathsf{stmt}}[\![\mathsf{while}(e)\ s]\!]_\eta & & \\
= \quad \mathbb{B}_{\mathsf{branch}}(\mathbb{S}_{\mathsf{exp}}[\![e]\!], t_1, n)_\eta & \text{where} & \eta = (b, n, x) \\
\quad \mathbb{S}_{\mathsf{stmt}}[\![s]\!]_{\eta_1} & & \eta_1 = (t_1, n, x)
\end{array}
$$

$$
\begin{array}{lll}
\mathbb{S}_{\mathsf{stmt}}[\![x = e.m(\overline{e})]\!]_\eta & & \eta = (b, n, x) \\
= \quad \mathbb{B}_{\mathsf{call}}(\mathbb{S}_{\mathsf{exp}}[\![e]\!], \mathbb{T}(m), \mathbb{S}_{\mathsf{exp}}[\![\overline{e}]\!], t_1)_\eta & & \\
\quad \mathbb{B}_{\mathsf{callfinish}}(\mathbb{V}(x))_{\eta_1} & & \eta_1 = (t_1, n, x) \\
\quad \mathbb{T}(m) \textit{ is the tag for method } m & &
\end{array}
$$

$$
\begin{array}{lll}
\mathbb{S}_{\mathsf{stmt}}[\![\mathsf{try}\ s_1\ \mathsf{catch}(T\ x)\ s_2]\!]_\eta & & \eta = (b, n, x) \\
= \quad \mathbb{F}(T\ x) & & \\
\quad \mathbb{S}_{\mathsf{stmt}}[\![s_1]\!]_{\eta_1} & & \eta_1 = (b, n, t_2) \\
\quad \mathbb{B}_{\mathsf{catch}}(T, \mathbb{V}(x), t_3)_{\eta_2} & \text{where} & \eta_2 = (t_2, n, x) \\
\quad \mathbb{S}_{\mathsf{stmt}}[\![s_2]\!]_{\eta_3} & & \eta_3 = (t_3, n, x)
\end{array}
$$

Rule for translating a method declaration:

$$
\begin{array}{lll}
\mathbb{S}_{\mathsf{method}}[\![T_m\ m(\overline{T\ x})s]\!] & & \\
= \quad \mathbb{F}(T_m\ m_{\mathsf{ret}}) & & \\
\quad \mathbb{F}(\overline{T\ x}) & & \\
\quad \mathbb{S}_{\mathsf{stmt}}[\![s]\!]_{\eta 1} & \text{where} & \eta_1 = (t_1, t_2, t_2) \\
\quad \mathbb{B}_{\mathsf{ret}}(\mathbb{S}_{\mathsf{exp}}[\![e]\!])_{\eta_2} & & \eta_2 = (t_2, t', t')
\end{array}
$$

Figure 5: The Translation Algorithm

$\mathbb{B}_{\mathsf{seq}}(e)_{(b,n,x)} =$

```
public void block_b() {
   try {  e;
          Scheduler.goto(n);
   } catch (Exception ex) {
          Scheduler.setException(ex);
          Scheduler.goto(this, x);
   }
}
```

$\mathbb{B}_{\mathsf{branch}}(e, t_1, t_2)_{(b,n,x)} =$

```
public void block_b() {
   try {  if (e) Scheduler.goto(t1);
          else Scheduler.goto(this, t2);
   } catch (Exception ex) {
   ............
```

$\mathbb{B}_{\mathsf{ret}}(e)_{(b,n,x)} =$

```
public void block_b() {
   try { m_ret = e;
         Scheduler.leaveFrame();
   } catch (Exception ex) {
   ............
```

$\mathbb{B}_{\mathsf{call}}(e_m, t_m, \overline{e}, t_{\mathsf{ret}})_{(b,n,x)} =$

```
public void block_b() {
   try { V(x) = e;
      Scheduler.enterFrame(this, t_ret);
      Scheduler.goto(e_m , t_m);
   } catch (Exception ex) {
   ............
```

$\mathbb{B}_{\mathsf{callfinish}}(f)_{(b,n,x)} =$

```
public void block_b() {
   if (Scheduler.hasException())
        Scheduler.goto(this, x);
   else { f = m_ret;
          Scheduler.goto(this, n); }
}
```

$\mathbb{B}_{\mathsf{throw}}(e)_{(b,n,x)} =$

```
public void block_b() {
   Scheduler.setException(ex);
   Scheduler.goto(x);
}
```

$\mathbb{B}_{\mathsf{catch}}(T, f, t_{\mathsf{catch}})_{(b,n,x)} =$

```
public void block_b() {
   if (Scheduler.getException() instanceof T) {
        x = (T) Scheduler.getException();
        Scheduler.clearException();
        Scheduler.goto(this, t_catch);
   } else {
        Scheduler.goto(this, x);
   }
}
```

Figure 6: Code generation for each basic block

needs to handle method calls and returns. Method arguments and return values are also handled in this way. In the definition of the source language we do not have method calls at the expression level, so the translation for expressions is straightforward. Access to a local variable x is renamed to its corresponding field $\mathbb{V}(x)$, as in the $\mathbb{S}_{exp}$ rules.

Statements are translated by the $\mathbb{S}_{stmt}$ rules. For simplicity, we treat each statement as a basic block in the demonstrated algorithm. Each translation step has a parameter $\eta = (b, n, x)$, a tuple of three control states. b is the state assigned to the current basic block, n is the state of the next basic block if the code is executed sequentially, and x is the state assigned to the current exception handler. The translation algorithm recursively translates the statements in nested structures and new states are created on the fly.

5.4 Optimization

The translation algorithm presented in this section can be optimized in a number of ways. In particular, the following optimizations can be used to reduce the code size of the output.

First, note that we treated every single statement as a basic block in the above algorithm, we can improve it by grouping several statements together in the same block, thus reducing the number of basic blocks.

Second, we do not have to translate every control structure in the source code. A method declaration or a statement needs to be translated into basic blocks only if it calls the communication primitive, or if it calls other methods that needs to be translated. The number of basic blocks can be greatly reduced if we leave most method declarations and statements untouched but only translate those where communication may happen.

Other optimizations can also be used to reduce the size of the generated code. For example, in the code generation in Figure 6, all the statements are guarded by exception handlers. In practice, most of these statements will not generate exceptions, so we can perform an analysis to eliminated unnecessary exception handlers. Dead code elimination can also be used to eliminate useless blocks.

6. EVALUATION

6.1 A small example

We have implemented a prototype of the code translator that translates a subset of the Java Card language using a similar algorithm. Some optimizations mentioned in the last section are also implemented.

The following is an example program written in the concurrent thread model. The method `verifyPIN()` sends a request to the host to read the PIN number, and waits for the reply from the host computer. Once the PIN number is read, it performs some checking on the PIN number, and returns the result of the verification. To focus on the control-flow translation, we are hiding the code of marshaling and unmarshaling the APDUs.

```
public boolean verifyPIN() throws Exception {
    ........          // marshall APDU to be sent
    sendAndReceive(); // send and wait for the reply
    ........          // unmarshall APDU from the host
    ........          // verify the PIN
    return true;
}
```

As an example we used in section 3, the `authenticate(N)` method prompts the user to input the PIN number at most N times. It uses a loop to call the `verifyPIN()` method defined above. This method also has an exception handler that deals with various exceptions that might happen during the communication.

```
1   public boolean authenticate(byte N) {
2       byte i = 0;
3       boolean success=false;
4       try {
5          while (i<N && !success) {
6              success = verifyPIN();
7              i = i+1;
8          }
9          return success;
10      } catch (Exception e) {
11         return false;
12      }
13  }
```

We are interested in the translation of the `authenticate` method because it contains a lot of control-flow primitives: method abstractions, loops, and exception handling. The following code is the expected output when we translate the `authenticate` method.

- ```
  public boolean authenticate_ret;
  public byte N_1;
  byte i_2;
  boolean success_3;
  Exception e_10;
  ```

  These are the fields generated by the translation algorithm. All of the fields are automatically renamed by the translator. The field "`authenticate_ret`" is used to store the returned value of the method, and "`N_10`" is used to pass the argument to this method. Others fields are translated from local variables.

- ```
  public void authenticate_entry() {
      i_2 = 0;
      success_3 = false;
      Scheduler.goto(this, STATE_5);
  }
  ```

  This block corresponds to line 2 and line 3 in the `authenticate` method. It uses $\mathbb{B}_{seq}$ case in Figure 6. Compared to the translation algorithm presented above, two optimization steps are used: the two adjacent statements on line 2 and 3 are merged in the same block, and exception handlers are removed because these statements will not generate exceptions.

- ```
  public void line_5() {
      if (i_2 < N_1 && !success_3)
         Scheduler.goto(this, STATE_6A);
      else
         Scheduler.goto(this, STATE_9);
  }
  ```

  This block corresponds to the branch on line 5. It uses the $\mathbb{B}_{branch}$ case in Figure 6, with an optimization step that removes the exception handler.

- ```
  public void line_6A() {
      Scheduler.enterFrame(this, STATE_6B);
      Scheduler.goto(this, STATE_verifyPIN_entry);
  }
  public void line_6B() {
      if (Scheduler.hasException()) {
         Scheduler.goto(this, STATE_10);
      } else {
          success_3 = verifyPIN_ret;
          Scheduler.goto(this, STATE_7);
  ```

```
    }
}
```

These two blocks correspond to the method call on line 6. They are similar to the $\mathbb{B}_{\mathsf{call}}$ and $\mathbb{B}_{\mathsf{callfinish}}$ cases in Figure 6.

- ```
  public void line_7() {
      i_2 = i_2 + 1;
      Scheduler.goto(this, STATE_5);
  }
  ```

  This block corresponds to line 7 and the $\mathbb{B}_{\mathsf{seq}}$ case.

- ```
  public void line_9() {
      authenticate_ret = success_3;
      Scheduler.leaveFrame();
  }
  ```

  This block corresponds to line 9 and the $\mathbb{B}_{\mathsf{ret}}$ case.

- ```
  public void line_10() {
      if (Scheduler.getException() instanceof Exception) {
          e_10 = (Exception) Scheduler.getException();
          Scheduler.clearException();
          Scheduler.goto(this, STATE_11);
      } else {
          Scheduler.goto(this, STATE_authenticate_exp);
      }
  }
  ```

  This block corresponds to line 10 and the $\mathbb{B}_{\mathsf{catch}}$ case.

- ```
  public void line_11() {
      authenticate_ret = false;
      Scheduler.leaveFrame();
  }
  ```

  This block corresponds to line 9 and the $\mathbb{B}_{\mathsf{ret}}$ case.

- ```
  public void authenticate_exp() {
      Scheduler.leaveFrame();
  }
  ```

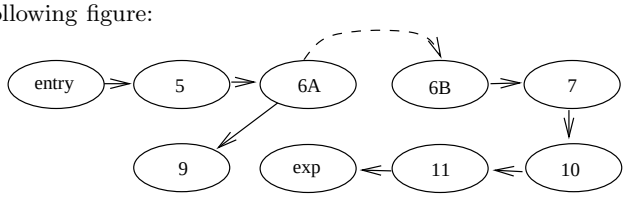
  This block serves as the exception handler of the whole method. It does nothing but simply returns so that the caller can handle the exception.

The above trampolined style code is executed by the main loop of the scheduler. Each time a basic block calls the `goto` method of the scheduler, the scheduler updates its current state. The main loop has a dispatch table so that the method corresponding to the current state can be located in each step.

```
class Scheduler {

  Object obj;
  short  s;
  ... ...
  public static void goto(Object o, short s ) {
     obj = o; state = s;
  }
  ... ...
  public static void mainloop() {
    ... ...
    while (true) {
      switch (state) {
      STATE_sendAndRecv:        return;
      ........
      STATE_authenticate_entry: (MyApp)obj.authenticate_entry();
      STATE_5:                  (MyApp)obj.line_5();
      STATE_6A:                 (MyApp)obj.line_6A()
      STATE_6B:                 (MyApp)obj.line_6B();
      STATE_7:                  (MyApp)obj.line_7();
      STATE_9:                  (MyApp)obj.line_9();
      STATE_10:                 (MyApp)obj.line_10();
      STATE_11:                 (MyApp)obj.line_11();
      STATE_authenticate_exp:   (MyApp)obj.authenticate_exp();
      ........
```

The underlying control-flow state machine is shown in the following figure:



6.2 Code quality and performance

We are interested in comparing machine-translated code with state-machine like Java Card programs written by hand, because they are supposed to behave identically on the card. The following is a quantitative estimation of the machine-generated code in comparison to the hand-written code. First, we define some variables that we are going to use in the comparison.

- Let d be the maximum depth, in terms of control-flow structures, of the communication primitives in the source code (in the concurrent-thread model). For example, the communication primitive `sendAndReceive()` in the `verifyPIN()` method has a depth of 5, because it is called in the following stack trace:

  1. The method verifyPIN()
  2. The while loop in authenticate()
  3. The exception handler in authenticate()
  4. The method authenticate()
  5. The top-level method that calls authenticate()

- Suppose the communication protocol has t different messages and the communication primitive appears t times in the source code (in the concurrent thread model). We can compare the number of control-flow states:

  – Hand-written code: as we have discussed in section 2, each time the card communicates with the host, the card program must yield control. Since there are t different messages in the communication protocol, the corresponding control-flow state machine in the hand-written code is likely to have at least t states.

  – Machine-generated code: the number of control states in the target code is the approximately the number of all the control structures in the source code that get translated to trampolined-style basic blocks. To be fair in the comparison, we assume that methods are used only once, because in the hand-written code, it is inconvenient to encapsulated the states into method calls and reuse them, as we have seen in section 3. Since we only translate those structures that have communications inside them, each communication primitive corresponds to at most d control-flow states, and there are t of them. Therefore, the total number of states should be no more than dt.

We are interested in the following comparisons: code quality and complexity, the space overhead due to the increased code size, and the performance overhead due to maintaining the trampolined-style dispatch table in the scheduler of the machine-generated code.
```

- Code quality and control-flow complexity:

  Quality of code is hard to measure, but subjectively, developing and maintaining high-level code should be easier. We claim that the machine-translated code has much better quality than the hand-written code, by comparing the complexity in the maintenance of control-flow state machine.

  In the hand-written code, the control-flow state machine has t states. The programmer has to be responsible for maintaining these states. As we discussed in section 3.4, the complexity of code maintenance is on the order of t^2e. As the number of states increases, the task of code maintenance becomes extremely difficult.

  On the other hand, the code translation algorithm automatically threads the control-flow states together. Despite the fact that the result of the translation has dt states, the complexity of the state machine is completely hidden under a high-level programming interface. Code maintenance is easy because the programmer only needs to modify the high-level code and the state machines can always be correctly regenerated by code translation.

- Space overhead:

  In smart card programming, code size is important because the space on the card is limited. The machine-generated code is usually longer than the hand-written code, because there are more control states in them. The difference in the number of the states is $dt - t = (d-1)t$. Suppose in the machine-generated code, each control state corresponds to k lines of additional code, as in Figure 6. With optimizations, the machine-generated code is similar to the example in section 6.1, where k is between 3 and 5 on average. Therefore, compared to the hand-written code, the machine-generated code has an estimated increase of:

  LOC Increase $< (d-1)tk$, where $k = 3 \ldots 5$.

- Performance overhead:

  Again, to be fair in comparison, we assume that the methods containing communication operations are not reused in the source code. Because each communication primitive is called in a stack frame of at most d control structures, the amortized cost on the operation of the dispatch table is no more than d. Usually, we have $d < 10$. For existing smart cards, the communication speed between the host computer and card is much slower than the processing speed of the smart card. Therefore, d operations on the dispatch table and method calls are almost trivial compared to the cost of the communication. We thus conjecture that such overhead is not noticeable on the smart card, but we still need future work to quantitatively verify our estimations on real cards.

6.3 BattleShip: a larger example

We were motivated to develop this code translation technique because of our experiences with creating interesting examples of smart card software. One case study was implementing the game of BattleShip in which part of the game state was stored on a Java Card. In BattleShip, two players each control an $N \times N$ board that contains some ship tokens. The initial location of a player's ships is secret to that player. Players take turns guessing a location for their opponent's ships, and after each guess that square of the opponent's board is revealed. A player who guesses correctly gets another chance, and the player whose ships are all hit first loses.

Our goal was to implement the BattleShip game so that one board is controlled by the smart card and the other is controlled by the terminal. The adversarial and confidentiality constraints make the game interesting from a security perspective [11]. To prevent cheating by one of the players it is necessary to use a commitment protocol when setting up the initial board state so that a cheater cannot surreptitiously move the location of one of their hidden ships after the opponent has made his guess.

We implemented the BattleShip game for the Java Card platform by hand. The code was quite complex due to the need for card–terminal communication inside of nested for-loops, for instance when the card and terminal exchange commitments for their initial board configurations. The resulting program contained approximately 550 lines of (quite ugly) Java code. In addition, that code does not implement complete error handling, partly because it requires very tedious and specific code to be written for each of the 16 states of the underlying state machine of the system.

With the help of our translation tool, a much cleaner, well-structured, high-level implementation of the BattleShip game can be rewritten in roughly the same amount of source code. The control-flow complexity is greatly reduced and the code quality is significantly improved. Using the analysis from the previous section, we estimate that $d = 5$, $t = 16$, $k = 4$, and the increase of code size is within $(d-1)tk = 256$ lines. Therefore, we can expect that the translated code could have approximately 800 lines. Out prototype does not yet implemented all the optimizations mentioned in Section 5.4, so the translated code looks quite redundant: it has more than 1000 lines of code and the expansion coefficient k is between 7 and 8.

6.4 Limitations

Our code translator is still under development. In the current stage, there are some limitations that the programmer must be aware of.

First, the local variables are translated to class fields so that recursive methods cannot be correctly translated. However, a recursive method need not to be translated as long as it does not perform any communication operations. So we are only limiting the methods that contain communications to be non-recursive. In comparison, when the programmer write the control-flow state machine by hand, the variables shared among different states have to be stored in the class fields anyway. In some sense, the translation is just simulating how a programmer implements the control-flow state machine. This limitation can be overcome by moving the local variables of recursive methods into the control stack. The translator only needs to know whether a method is recursive before choosing the right resource allocation strategy. The trade-off is that such control stack accesses may severely hurt performance.

Second, we currently require that a call to a translated method must be made in a single assignment statement like "x = a.m();" so as to avoid the cases where method calls are nested inside complicated expressions. This limitation

is merely syntactic and it only applies to method calls that need to be translated. In the future we can add more compilation rules to decompose the expressions and add variables for holding intermediate results in order to allow more method calls to be nested in expressions.

7. RELATED WORK

Interestingly, there is a good analogy between smart cards and web servers. In the early stage, both were designed to store static data: the earliest smart cards were just memory cards, and the earliest web servers were designed to process static hypertext files. As they evolve, both are becoming capable of performing some nontrivial computation: the web servers have server-side scripts and the smart cards have card applets. Both have a command-response communication model: the client (host computer) sends a request to the server (the card) and receives a reply. Similarly, they experience the same control flow problems such as session management. There has been much research about web continuations [4]. The Apache Cocoon project [1] allows web applications to be written in the Flowscript language that supports continuations, with a similar programming model to our concurrent thread model. However, Java Card continuations and web continuations differ in several aspects. The web continuations can multiply and be invoked simultaneously [10], and the resource management for web continuations is much more difficult. The web server simultaneously handles requests from thousands of different sessions, but the smart card only needs to deal with one host application at a time. Our paper shows that saving and restoring continuations on the Java Card is simple and practical.

This code translation tool was developed as part of the PISCES project at the University of Pennsylvania. The PISCES group studies Protocols and Implementations for Smart-Card Enabled Software, with the goal of developing appropriate software engineering techniques for security-critical systems built using smart cards. Previous work has focused on designing open APIs for smart card applications [5]. Our ongoing work is examining the problem of how to partition a system among the smart card, its terminal, and possibly remote hosts, given the constraints on resources and the need to protect confidential information.

8. CONCLUSION

We examined the control-flow difficulties in conventional Java Card programming. The half-duplex APDU communication protocol induces the master-slave programming model, which leads to a state-machine-based design pattern for card applets, because a card program cannot interact with a host inside method calls or loops. We argued that the underlying communication model should not restrict the design of high-level programming interface, and we proposed the concurrent thread model for future Java Card programming. In this model, the card applet has an active thread that communicates with the host computer via communication primitives. Language-based control-flow structures can now be used to control the interaction with the host, relieving the programmers' burden of coding up control-flow state machines.

In existing Java Card environments, the lack of thread support limits the application of the concurrent thread model. We proposed a code translation algorithm based on existing compilation techniques that converts programs written in the concurrent thread model to conventional Java Card programs in the master-slave model. The result is that the developer can program the smart card using the concurrent thread model without losing compatibility with the existing Java Card framework: the existing API can be reused and the translated code can be installed and run on the card. The translation tool improves the code quality of smart card applications, while introducing acceptable space and performance overheads.

[1] http://cocoon.apache.org

Acknowledgments: The authors thank the other members of the PISCES group, Rajeev Alur, Watee Arjsamat, Carl Gunter, Andre Scedrov, Raman Sharykin, and Jason Simas, for their suggestions and input on this work. Watee provided much help with the Java Card development platform, and Raman did much of the programming on the battleship implementation.

9. REFERENCES

[1] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] E. W. Dijkstra. Go to statement considered harmful. *Comm. of the ACM*, 11(3):147–148, Mar. 1968.

[3] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *International Conference on Functional Programming*, pages 18–27, 1999.

[4] P. Graunke, S. Krishnamurthi, S. V. D. Hoeven, and M. Felleisen. Programming the Web with high-level programming languages. In *European Symposium on Programming*, 2001.

[5] C. A. Gunter. Open APIs for embedded security. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[6] P. H. Hartel and E. K. de Jong. A programming and a modelling perspective on the evaluation of Java card implementations. In I. Attali and T. Jensen, editors, *1st Java on Smart Cards: Programming and Security (Java Card Workshop)*, volume LNCS 2041, pages 52–72, Cannes, France, Sep 2000. Springer-Verlag, Berlin.

[7] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java. In *Conference of Object-Oriented Programming, Systems, Languages and Applications*, volume 34 of *ACM SIGPLAN Notices*. ACM Press, Oct. 1999.

[8] R. Kelsey, W. Clinger, and J. R. (Editors). Revised[5] report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[9] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[10] C. Queinnec. The influence of browsers on evaluators or, continuations to program Web servers. *ACM SIGPLAN Notices*, 35(9):23–33, 2000.

[11] L. Zheng, S. Chong, S. Zdancewic, and A. C. Myers. Building secure distributed systems using replication and partitioning. In *IEEE 2003 Symposium on Security and Privacy*. IEEE Computer Society Press, 2003.