Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 141 (2005) 145–162

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Towards Verification of Well-Formed Transactions in Java Card Bytecode

René Rydhof Hansen[1]

Informatics and Mathematical Modelling
Technical University Denmark
Kgs. Lyngby, Denmark

Igor A. Siveroni[2]

Department of Computing
Imperial College
London, UK

Abstract

Using transactions in Java Card bytecode programs can be rather tricky and requires special attention from the programmer in order to work around some of the limitations imposed and to avoid introducing serious run-time errors due to inappropriate use of transactions.
In this paper we present a novel analysis that combines control and data flow analysis with an analysis that tracks active transactions in a Java Card bytecode program. We formally prove the correctness of the analysis and show how it can be used to solve the above problem of guaranteeing that transactions in a Java Card bytecode program are *well-formed* and thus do not give rise to run-time errors.

Keywords: Java Card bytecode, well-formed transactions, static analysis, Flow Logic

1 Introduction

Smart cards are increasingly being used in applications that require a certain amount of safety and security, such as electronic Metro tickets and electronic wallets. Smart cards are also at the heart of every GSM mobile phone where

[1] Email: rrh@imm.dtu.dk
[2] Email: siveroni@doc.ic.ac.uk

doi:10.1016/j.entcs.2005.02.032

the so-called SIM (Subscriber Identity Module) cards are actually smart cards. The widespread deployment and often sensitive nature of smart card applications makes it imperative that the programs running on the smart cards perform correctly.

Java Cards are a special type of smart card that has an on-card Java Card Virtual Machine making it possible to develop programs for such smart cards in a dialect of Java, called Java Card. Such programs are then compiled to a dialect of Java bytecode called Java Card Virtual Machine Language (JCVML) or Java Card bytecode. One particular problem for JCVML programs is the use of transactions (a mechanism for ensuring atomic updates of variables). JCVML does not allow transactions to be nested, i.e., there can be only one open transaction at any given moment. Attempts at opening another transaction will result in a run-time error; this is also the result of attempting to close a transaction when there is no open transaction. Due to the nature of JCVML programs transactions may even be opened and closed in different methods and depend on particular data flows in the program. This makes it quite hard to actually ensure that transactions are indeed well-formed.

In this paper we show how previously developed control and data flow analyses for a JCVML-like language called Carmel, a rational reconstruction of JCVML with the same expressive power as JCVML but better suited for formal methods, can be extended to also track transactions and ultimately show that the transactions of a given program are indeed well-formed and thus that the program will never give rise to a transaction related run-time error.

Related work. In [7] it is shown how high-level safety and security properties can be enforced in Java Card programs by automatically generating JML (Java Modeling Language), cf. [1], annotations for the program. Well-formedness of transactions is given as an example of a high-level property that can be verified using this technique. We are not aware of any related work that targets Java Card bytecode and the use of static analysis to solve this problem seems novel.

2 Well-Formed Transactions

The JCVML API provides access to a well-known and essential tool for ensuring data and program integrity under the circumstances outlined namely *transactions*. A transaction guarantees the atomicity of updates taking place in the transaction, i.e., either all the updates are successful or none of them are successful. However, resources on a smart card are rather limited, and as a consequence JCVML transactions are not allowed to be nested, i.e., there can be only one active transaction at a time: any attempt at beginning a new

`public void atomicWrapper(byte [], short)`	// $\hat{TD}_0$	$\hat{TD}_1$
`{`	//	
`  0: API.getTransactionDepth`	// $\{0\}$	$\{1\}$
`  1: if le 0 goto 4`	// $\{0\}$	$\{1\}$
`  2: push 1`	// $\emptyset$	$\{1\}$
`  3: goto 5`	// $\emptyset$	$\{1\}$
`  4: push 0`	// $\{0\}$	$\emptyset$
`  5: store 3`	// $\{0\}$	$\{1\}$
`  6: load 3`	// $\{0\}$	$\{1\}$
`  7: if ne 0 goto 9`	// $\{0\}$	$\{1\}$
`  8: API.beginTransaction`	// $\{0\}$	$\emptyset$
`  9: load 0`	// $\{1\}$	$\{1\}$
` 10: load 1`	// $\{1\}$	$\{1\}$
` 11: load 2`	// $\{1\}$	$\{1\}$
` 12: invokevirtual dosomething(byte[], short)`	// $\{1\}$	$\{1\}$
` 13: load 3`	// $\{1\}$	$\{1\}$
` 14: if ne 0 goto 16`	// $\{1\}$	$\{1\}$
` 15: API.commitTransaction`	// $\{1\}$	$\emptyset$
` 16: return`	// $\{0\}$	$\{1\}$
`}`		

Fig. 1. Carmel implementation of a wrapper method

transaction while another is already active will result in an exception being thrown; similarly an exception will be thrown if an attempt at closing a transaction (either by committing or aborting it) is made when no transaction is active. Finally, no transaction must be active when the program terminates. If no transaction on any possible execution path in a given program violates the above rules the transactions of the program are said to be *well-formed*. The limitation imposed by the above transaction semantics leads to a defensive programming style where it is explicitly checked if a transaction is in progress before starting a new one and similarly checking that a transaction is active before trying to close it. Furthermore, programmers often have to "work around" the limitation of not being able to nest transactions and this leads to a non-trivial control flow for transactions where a transaction may be started in one method and committed (or aborted) in another, possibly dependent upon whether or not the first method was invoked in a context where a transaction was already in progress or not. These problems conspire to make it very hard for a programmer to ensure that transactions are indeed well-formed since all possible program executions have to be taken into account. Use of exceptions that may also open or close transactions further complicates the situation. In Section 4 we develop a program analysis that can be used to statically guarantee that all the transactions in a program are well-formed.

In Figure 1 a Carmel method, called `atomicWrapper`, is shown to illustrate both the concrete Carmel syntax and the problems regarding transactions discussed above. The example is taken from a demonstration applet, an electronic purse called DeMoney, modified to fit our language. It was created by Trusted Logic as part of the SecSafe project[8] and is specified in [5]. Note that we use '`//`' to indicate program comments and that the comments in Figure 1 actually show the (partial) result of the above mentioned analysis and will be explained in more detail in Section 5. Finally note that Carmel, like JCVML, is a stack based language. The `atomicWrapper` method is intended to be used as a "wrapper" for a method called `dosomething` (lines 9–12) guaranteeing that `dosomething` is always invoked inside a transaction and thereby ensuring the atomicity of the updates performed by that method. It illustrates the problems discussed above: if the wrapper method, `atomicWrapper`, is invoked inside an active transaction, then it must not start a new transaction; this is handled by first obtaining the current transaction depth (line 0) and then in lines 1–5 recording a "0" or a "1", depending on the current transaction depth, in local variable number "3" (the '`le`' operator used in line 1 is Carmel's less then or equal, '$\leq$', operator). Finally, the value of local variable "3" is checked before beginning a transaction (lines 6–8 where the `ne` operator used in line 7 is Carmel's not equal, '$\neq$', operator) and if a transaction was started (in line 8) in this method then it should be committed in this method as well (lines 13–15).

3 The Carmel Language

The full JCVML comprises more than 100 instructions and a comprehensive API implementing all sorts of helpful routines, including access to the virtual machine's transaction layer. Rather than specifying our analysis for the JCVML we instead base our analysis and developments on the Carmel language. The Carmel language is an abstraction of the JCVML: it abstracts away most of the implementation oriented details, e.g., the constant pool, while retaining the full expressive power of the JCVML, indeed there is a simple syntactic conversion from JCVML to Carmel. Syntax and semantics of the Carmel language is defined and discussed in detail in [9,10]. For presentation purposes we only include the instructions and features that are needed to present the main insights.

The instruction set for Carmel includes instructions for stack manipulation, local variables, object generation, field access, a simple conditional, throwing an exception, and method invocation and return:

$$\begin{array}{rl}\mathsf{Instr} ::= & \texttt{push}\ c \mid \texttt{pop}\ n \mid \texttt{numop}\ op \mid \texttt{new}\ \sigma \mid \texttt{invokevirtual m} \mid \\ \mid & \texttt{return} \mid \texttt{getfield}\ f \mid \texttt{putfield}\ f \mid \texttt{if}\ cmpOp\ \texttt{goto}\ pc_0 \\ \mid & \texttt{load}\ x \mid \texttt{store}\ x \mid \texttt{throw}\end{array}$$

In addition we include, from the API, instructions for accessing the transaction functionality:

$$\begin{aligned} \mathsf{Instr} ::= \ldots \mid & \ \texttt{API.beginTransaction} \mid \texttt{API.commitTransaction} \\ \mid & \ \texttt{API.abortTransaction} \mid \texttt{API.getTransactionDepth} \end{aligned}$$

A Carmel program, $P \in \mathsf{Program}$, is defined to be the set of classes it declares: $P.classes$. Each class, $\sigma \in \mathsf{Class}$, contains a set of methods, $\sigma.methods \subseteq \mathsf{Method}$, and fields $\sigma.fields \subseteq \mathsf{Field}$. Each is comprised by an instruction for each program counter, $pc \in \mathbb{N}_0$ in the method, $m.instructionAt(pc) \in \mathsf{Instr}$. For legibility we shall use a JCVML-like concrete syntax as shown in Figure 1.

3.1 Semantics

The semantics for Carmel is a straightforward, albeit involved, structural operational semantics (SOS) and is described in detail in [9]. In this section we introduce a simplified version of the Carmel semantics that is better suited for investigating and presenting the problem at hand. The main simplification is, maybe somewhat surprisingly, the removal of the actual underlying transaction mechanics: instead of defining a full-blown transactional semantics we simply abstract the underlying transaction mechanism away and track only the transaction depth in a special semantic component. It should be noted that while this abstraction is very convenient for proof and presentation purposes it does not make the problem any easier to solve and, consequently, tracking the transaction depth is sufficient for our purposes.

A value is either a number or an object reference (a location): $\mathsf{Val} = \mathsf{Num}+\mathsf{ObjRef}$. The global heap is then a map from object references to objects: $\mathsf{Heap} = \mathsf{ObjRef} \to \mathsf{Object}$, where objects are simply maps from field names to values: $\mathsf{Object} = \mathsf{Field} \to \mathsf{Val}$. Objects are instantiated from classes and we use $o.class$ to denote the class of an object $o \in \mathsf{Object}$. In JCVML only objects of class `Throwable` can be used as exceptions, for simplicity we allow any class to be used as an exception: $\mathsf{Exception} = \mathsf{Class}$. The operand stack is modelled as a sequence of values: $\mathsf{Stack} = \mathsf{Val}^*$ and the local heap (for variables local to a method) as a map from (local) variables to values: $\mathsf{LocHeap} = \mathsf{Var} \to \mathsf{Val}$. Stack frames record the current transaction depth (can only be 0 or 1) and the current method and program counter along with a local heap and an operand stack. To facilitate our proofs we also annotate the current method with the context, i.e., transaction depth in which it was originally invoked (either 0 or 1). Thus an annotated method, (m, τ_m), belongs to the domain $\mathsf{Method} \times \{0, 1\}$ and is written m^{τ_m}. The domain for stack frames is then given by: $\mathsf{Frame} = \{0, 1\} \times (\mathsf{Method} \times \{0, 1\}) \times \mathbb{N}_0 \times \mathsf{LocHeap} \times \mathsf{Stack}$. For exceptions a special frame is defined: $\mathsf{ExcFrame} = \{0, 1\} \times \mathsf{ObjRef} \times \mathsf{Method} \times \mathbb{N}_0$.

With the semantic domains in place we can now specify the semantic configurations and the reduction rules. Configurations are either *running configurations* or *final configurations*: $\mathsf{Conf} = \mathsf{RunConf} + \mathsf{FinConf}$. The running configurations are of the form: $\langle H, F :: SF\rangle$ where $H \in \mathsf{Heap}$, $SF \in \mathsf{Frame}^*$, and $F = \langle \tau, m^{\tau_m}, pc, L, S\rangle \in \mathsf{Frame}$, meaning that the program is currently executing the instruction at program counter pc in method m with local heap L and operand stack S and the method was invoked in transaction depth τ_m, or $F = \langle \tau, loc_X, m_X, pc_X\rangle \in \mathsf{ExcFrame}$ meaning that an exception has been thrown, in method m_X at program counter pc_X, with loc_X pointing to the exception object. If $\tau = 1$ then the instruction is executed within an active transaction and if $\tau = 0$ no transaction is in progress at the current instruction (note that this may of course be different from the transition depth in which the method was invoked: τ_m). Final configurations are of the form $\langle H, \tau, \langle \mathsf{Ret}\ v\rangle\rangle$ indicating that a program has terminated in transaction depth τ and with return value v (and $v = \perp$ if no return value is present). This leads to reduction rules of the following form for a given program, $P \in \mathsf{Program}$: $P \vdash C \Longrightarrow C'$ for $C, C' \in \mathsf{Conf}$. Figure 2 shows the semantic rules for a few interesting instructions, the full semantics can be found in [9]. The semantics for the (full) transaction API is shown in Figure 3.

In the reduction rule for `invokevirtual` (method invocation) we must take care to handle dynamic dispatch correctly. This is done as in JCVML by using a function, *methodLookup*, to represent the class hierarchy. It takes a method identifier, m', and a class, *o.class*, as parameters and returns the method, m_v, that implements the body of m', i.e., the latest definition of m' in the class hierarchy. In the same rule, note that a reference to the object is passed to the object itself in local variable number 0. A final thing to note is that run-time errors caused by transactions that are not well-formed are modelled as stuck configurations. For lack of space only `return`-instructions that actually return a value are shown. Note that two cases must be handled: one for "normal" method return and one for program termination, i.e., executing a `return`-instruction on an empty call stack.

For the `throw` instruction we use the *findHandler* function (cf. [9] for a definition and explanation) to check if a local handler exists for the exception being thrown; if not a special exception frame is put on top of the call stack. This is a slight deviation from the official JCVM to facilitate the SOS definition of the semantics and the proof of correctness of the analysis.

The main thing to note in the semantics for the API is that beginning a transaction (`API.beginTransaction`) will only succeed if the current transaction depth is 0, i.e., no other transaction is in progress. Similarly, committing or aborting a transaction, using instruction `API.commitTransaction`

$$\frac{m.instructionAt(pc) = \texttt{push}\ c}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H, \langle \tau, m^{\tau_m}, pc+1, L, c :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{pop}}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, c :: S\rangle :: SF\rangle \Longrightarrow \langle H, \langle \tau, m^{\tau_m}, pc+1, L, S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invokevirtual}\ m' \wedge \\ S = v_1 :: \cdots :: v_{|m'|} :: loc :: S_0 \wedge m_v = methodLookup(m', o.class) \wedge \\ o = H(loc) \wedge L' = [0 \mapsto loc, 1 \mapsto v_1, \ldots, |m'| \mapsto v_{|m'|}] \end{array}}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H, \langle \tau, m_v^{\tau}, 0, L', \epsilon\rangle :: \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{return}}{\begin{array}{r} P \vdash \langle H, \langle \tau', m'^{\tau_{m'}}, pc', L', v :: S'\rangle :: \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow \\ \langle H, \langle \tau', m^{\tau_m}, pc+1, L, v :: S\rangle :: SF\rangle \end{array}}$$

$$\frac{m.instructionAt(pc) = \texttt{return}}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, v :: S\rangle :: \epsilon\rangle \Longrightarrow \langle H, \tau, \langle \mathsf{Ret}\ v\rangle\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{throw} \\ F = \begin{cases} \langle \tau, m^{\tau_m}, pc_0, L, S\rangle & \text{if } findHandler(m, pc, H(loc_X).class) = pc_0 \\ \langle \tau, loc_X, m, pc\rangle & \text{if } findHandler(m, pc, H(loc_X).class) = \perp \end{cases} \end{array}}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, loc_X :: S\rangle :: SF\rangle \Longrightarrow \langle H, F :: SF\rangle}$$

Fig. 2. Semantic Rules (excerpt)

$$\frac{m.instructionAt(pc) = \texttt{API.getTransactionDepth}}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H, \langle \tau, m^{\tau_m}, pc+1, L, \tau :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{API.beginTransaction}}{P \vdash \langle H, \langle 0, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H, \langle 1, m^{\tau_m}, pc+1, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{API.commitTransaction}}{P \vdash \langle H, \langle 1, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H, \langle 0, m^{\tau_m}, pc+1, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{API.abortTransaction}}{P \vdash \langle H, \langle 1, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H, \langle 0, m^{\tau_m}, pc+1, L, S\rangle :: SF\rangle}$$

Fig. 3. Semantic Rules for the transaction API

or `API.abortTransaction` respectively, only succeeds if a transaction is currently active. This implies that `API.beginTransaction` should only be executed in a context where the current transaction depth is zero and the instructions `API.commitTransaction` and `API.abortTransaction` should only be executed if the current transaction depth is one. This is formalised in Definition 3.2 below. It should be clear from the above and the discussion in the previous section that it can be very hard, if not impossible, for a programmer to manually inspect a program and guarantee that for all possible execution paths all transactions will always be well-formed.

To conclude the semantics, we need to define initial configurations. Carmel programs, like JCVML applets, can have multiple entry-points. For simplicity, and without loss of generality, we assume that each class, σ, has exactly one entry point, denoted m_σ, taking exactly one parameter, a (self-)reference to the calling object (denoted loc_σ):

Definition 3.1 (Initial Configurations) *If* $P \in \mathsf{Program}$ *then* C *is an* initial configuration *if and only if* $C = \langle H, \langle 0, m_\sigma^0, 0, [0 \mapsto loc_\sigma], \epsilon\rangle :: \epsilon\rangle$ *and* $H(loc_\sigma).class = \sigma$*, for* $\sigma \in P.classes$ *and* m_σ *the entry point of* σ*.*

Note that initially no transactions are open. For ease of reference, we define *P.entry* to be the set of entry points for the program P. We can now formally define the notion of well-formedness:

Definition 3.2 (Well-Formedness) *Let* C_0 *be an initial configuration of* $P \in \mathsf{Program}$ *and let* $C_1 \in \mathsf{Conf}$ *such that* $P \vdash C_0 \Longrightarrow^* C_1$ *then the transactions of* P *are said to be* well-formed *if and only if for all configurations* $C_1 = \langle H, \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle$ *the following holds:*

(i) $m.instructionAt(pc) = \texttt{API.beginTransaction} \Rightarrow \tau = 0$

(ii) $m.instructionAt(pc) = \texttt{API.commitTransaction} \Rightarrow \tau = 1$

(iii) $m.instructionAt(pc) = \texttt{API.abortTransaction} \Rightarrow \tau = 1$

and for all configurations $C_1 = \langle H, \tau, \langle \mathsf{Ret}\ v\rangle\rangle$ *it is the case that* $\tau = 0$*.*

4 Control, Data, and Transaction Flow Analysis

In this section we present a combined control, data, and transaction flow analysis that can be used to verify that a program uses transactions in a well-formed manner. The analysis is based on previously developed control and data flow analyses (see [4,3] for a full discussion); these analyses are then extended with an analysis to track active transactions, we call this a *transaction flow* analysis. The transaction flow analysis (TFA) tracks (a conservative approximation of) the possible transaction depth for every instruction in the program. We have found that in order to obtain the precision needed for our purposes it is necessary for the TFA to be a context dependent analysis (see [6]) where the contexts are taken to be the transaction depth in which a given method was invoked; thus there are only two possible contexts, namely zero and one, which greatly limits the increase in complexity that usually follows from adding contexts to an analysis.

4.1 Abstract Domains

The abstract domain for numbers reflects the fact that we need to track constants but not computations involving constants since any use of a manipulated or computed value for transaction depths would be very hard to verify (both manually and automatically) and should be regarded as highly suspect: $\overline{\mathsf{Num}} = \mathbb{Z}^\top$. We shall write INT for the top-element of $\mathbb{Z}^\top$.

For object references we follow the "usual" approach, called *class object* *graphs* in [11], and abstract object references into their corresponding class: $\overline{\mathsf{ObjRef}} = \mathsf{Class}$. If more precision is needed an abstraction based on the *textual object graphs* of [11] could be used instead. However, for typical Java Card programs the former is quite acceptable since very few classes are instantiated more than once. In the interest of legibility we shall write (Ref σ) for $\sigma \in \overline{\mathsf{ObjRef}}$. The domain for abstract exceptions is the same as for classes: $\overline{\mathsf{Exception}} = \overline{\mathsf{ObjRef}}$. A value is either a number or an object reference; this is easily modelled directly as: $\overline{\mathsf{Num}} + \overline{\mathsf{ObjRef}}$ and we shall then use sets of these values as our abstract values: $\overline{\mathsf{Val}} = \mathcal{P}(\overline{\mathsf{Num}} + \overline{\mathsf{ObjRef}})$. As discussed above, the analysis is context dependent with transaction depths (at method invocation) as context: $\overline{\mathsf{Context}} = \{0, 1\}$.

We follow the approach of Freund and Mitchell, cf. [2], and keep track of the local heap and operand stack (and transaction depth) for *each* instruction and every context; this makes the analysis of local heaps, operand stacks, and transaction depths flow sensitive (intra-procedurally) which is necessary for tracking changes in the transaction depth and analysing the cases where transaction depths are checked before starting or ending a transaction. Since an instruction is uniquely determined by a method and a program counter we introduce the notion of addresses for convenience: $\mathsf{Addr} = \mathsf{Method} \times \mathbb{N}_0$. The local heap is then modelled as a map, for every context and address, from (local) variables to abstract values: $\overline{\mathsf{LocHeap}} = \overline{\mathsf{Context}} \to \mathsf{Addr} \to \mathsf{Var} \to \overline{\mathsf{Val}}$. Stacks are modelled as sequences of abstract values (again one for each context and instruction): $\overline{\mathsf{Stack}} = \overline{\mathsf{Context}} \to \mathsf{Addr} \to (\overline{\mathsf{Val}}^*)^\top$. Abstract heaps are also modelled in a straightforward way as maps from object references to abstract objects: $\overline{\mathsf{Heap}} = \overline{\mathsf{Context}} \to \overline{\mathsf{ObjRef}} \to \overline{\mathsf{Object}}$. Abstract objects are maps from fields to abstract values: $\overline{\mathsf{Object}} = \mathsf{Field} \to \overline{\mathsf{Val}}$. To track exceptions that are not handled locally an exception cache is needed: $\overline{\mathsf{ExcCache}} = \overline{\mathsf{Context}} \to \mathsf{Method} \to \mathcal{P}(\overline{\mathsf{Exception}})$. Finally, to track the transaction depth for each instruction, we use the following: $\overline{\mathsf{Trans}} = \overline{\mathsf{Context}} \to \mathsf{Addr} \to \mathcal{P}(\{0, 1\})$.

4.2 Flow Logic Specification

The Flow Logic framework is a specification oriented (rather than implementation oriented) approach to program analysis. Instead of detailing how an analysis is to be carried out the framework is used to specify what it means for an analysis result to be an acceptable, or correct, analysis of a program. This separation makes it very convenient to specify analyses in the framework and often gives rise to (relatively) clear and succinct specifications.

The judgements for our control, data, and transaction analysis take the following form $(\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD}) \models (m_0, pc_0) : \texttt{instr}$ meaning that the (proposed) analysis, $(\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD})$, is a correct approximation of the instruc-

$$
\begin{array}{l}
(\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD}) \models (m_0, pc_0) : \texttt{API.getTransactionDepth} \\
\quad \text{iff} \quad \forall \delta \in \hat{TD}_{\{0,1\}}(m_0, 0) \colon \hat{TD}_\delta(m_0, pc_0) \neq \emptyset \Rightarrow \\
\qquad\qquad \hat{TD}_\delta(m_0, pc_0) \sqsubseteq \hat{TD}_\delta(m_0, pc_0 + 1) \\
\qquad\qquad \forall \delta' \in \hat{TD}_\delta(m_0, pc_0) : \{\delta'\} :: \hat{S}_\delta(m_0, pc_0) \sqsubseteq \hat{S}_\delta(m_0, pc_0 + 1) \\
\qquad\qquad \hat{L}_\delta(m_0, pc_0) \sqsubseteq \hat{L}_\delta(m_0, pc_0 + 1) \\
\\
(\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD}) \models (m_0, pc_0) : \texttt{API.beginTransaction} \\
\quad \text{iff} \quad \forall \delta \in \hat{TD}_{\{0,1\}}(m_0, 0) \colon \hat{TD}_\delta(m_0, pc_0) \neq \emptyset \Rightarrow \\
\qquad\qquad \{1\} \subseteq \hat{TD}_\delta(m_0, pc_0 + 1) \\
\qquad\qquad \hat{S}_\delta(m_0, pc_0) \sqsubseteq \hat{S}_\delta(m_0, pc_0 + 1) \\
\qquad\qquad \hat{L}_\delta(m_0, pc_0) \sqsubseteq \hat{L}_\delta(m_0, pc_0 + 1) \\
\\
(\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD}) \models (m_0, pc_0) : \texttt{API.commitTransaction} \\
\quad \text{iff} \quad \forall \delta \in \hat{TD}_{\{0,1\}}(m_0, 0) \colon \hat{TD}_\delta(m_0, pc_0) \neq \emptyset \Rightarrow \\
\qquad\qquad \{0\} \subseteq \hat{TD}_\delta(m_0, pc_0 + 1) \\
\qquad\qquad \hat{S}_\delta(m_0, pc_0) \sqsubseteq \hat{S}_\delta(m_0, pc_0 + 1) \\
\qquad\qquad \hat{L}_\delta(m_0, pc_0) \sqsubseteq \hat{L}_\delta(m_0, pc_0 + 1)
\end{array}
$$

Fig. 4. Excerpt of Flow Logic for transaction API

tion `instr` located in method m at program counter pc, where $\hat{H} \in \overline{\mathsf{Heap}}$, $\hat{L} \in \overline{\mathsf{LocHeap}}$, $\hat{S} \in \overline{\mathsf{Stack}}$, $\hat{E} \in \overline{\mathsf{ExcCache}}$, and $\hat{TD} \in \overline{\mathsf{Trans}}$. Later in this section we show how to lift this to cover entire programs and take initial configurations into account. In Figures 4, 5, and 6 an excerpt of the Flow Logic specification is shown, the full specification can be found in [4,3]. Note that to enhance the legibility of the Flow Logic specification we use linebreaks and indentation rather than explicitly writing out all the conjunction operators. Due to space considerations we only discuss a few instructions in detail. For a full discussion of the control flow and data flow aspects of the analysis see [4,3]. In the interest of both legibility and succinctness we introduce the shorthand notation: $\hat{TD}_{\{0,1\}}(m, pc)$ to mean $\hat{TD}_0(m, pc) \cup \hat{TD}_1(m, pc)$.

Since our analysis is a context dependent analysis, where context is the transaction depth in which the current method (denoted m_0) was invoked, the specification for all instructions are prefixed with a $\forall \delta \in \hat{TD}_{\{0,1\}}(m_0, 0)$ to ensure that all possible contexts in which the current method was invoked are taken into account. This abstract context is then used as an index for the abstract domains operated over and is written as a sub-script, e.g., the δ in $\hat{S}_\delta$.

With the above in mind the `API.getTransactionDepth` instruction can now be analysed (as shown in Figure 4). First we must ensure that the instruction is analysed in all the possible contexts that the current method has been called in, but we only continue with the analysis if the current instruction is actually executed in such a context. This is expressed as the following condition imposed by all instructions: $\forall \delta \in \hat{TD}_{\{0,1\}}(m_0, 0)\colon \hat{TD}_\delta(m_0, pc_0) \neq \emptyset$. Note that all instructions that are reachable in a given context, δ, will have a

$\hat{TD}_\delta(m_0, pc_0) \neq \emptyset$; thus the premise of the above implication can be seen as a simple reachability test that enhances the precision of the analysis by ignoring instructions and contexts that are unreachable. Now analysing the effect of the instruction is straightforward: since this instruction does not alter the transaction depth, the current (abstract) transaction depth is simply copied forward. In symbols: $\hat{TD}_\delta(m_0, pc_0) \subseteq \hat{TD}_\delta(m_0, pc_0 + 1)$. Next the value of the current abstract transaction depth is put on top of the stack for the next instruction: $\forall \delta' \in \hat{TD}_\delta(m_0, pc_0) : \{\delta'\} :: \hat{S}_\delta(m_0, pc_0) \sqsubseteq \hat{S}_\delta(m_0, pc_0 + 1)$ where $\sqsubseteq$ is the point-wise extension of $\subseteq$ to finite sequences of abstract values. Finally, since no local variables were modified they are simply copied onto the next instruction: $\hat{L}_\delta(m_0, pc_0) \sqsubseteq \hat{L}_\delta(m_0, pc_0 + 1)$. Here $\sqsubseteq$ is the point-wise extension of $\subseteq$ to maps in the domain $\mathsf{Var} \to \overline{\mathsf{Val}}$. The specification for the `API.beginTransaction` is quite similar except that here the transaction depth is changed; after the instruction has been executed the transaction depth is certain to be one: $\{1\} \subseteq \hat{TD}_\delta(m_0, pc_0 + 1)$. The analysis of the remaining transaction instructions is similar, and we shall not go into further details here.

The Flow Logic specification for the `invokevirtual` instruction (see Figure 5) consists mainly of formulae for moving actual parameters to the invoked method and making sure to copy the return value back closely mimicking the way the concrete semantics operates. We shall not go into further detail with that aspect of the analysis here, merely refer to [4]. For our purposes in this paper the first interesting thing to note about the analysis of `invokevirtual` is how the current abstract context is copied to the invoked method as a base context: $\forall \delta' \in \hat{TD}_\delta(m_0, pc_0) : \{\delta'\} \subseteq \hat{TD}_{\delta'}(m_v, 0)$ where m_v is the invoked method found by searching the class hierarchy taking overloading into account. Note that δ' is used both as a value and as an index. The second thing to note is how the context, i.e. the value of the transaction depth, of the invoked method is written back to the invoking method upon return: $\hat{TD}_{\delta'}(m_v, \mathsf{END}_{m_v}) \subseteq \hat{TD}_\delta(m_0, pc_0 + 1)$. Here $(m_v, \mathsf{END}_{m_v})$ is a special "end-address" for the method "m_v" used in a manner similar to "exit" nodes in control flow graphs. Note that since the invoked method may change the transaction depth we simply use the context returned from the invoked method and ignore the context at the point of invocation. In addition to the normal flow of control associated with method invocation, we must also ensure that any exceptions thrown but not handled in the invoked method are re-thrown in the invoking method. To this end the HANDLE predicate is

$$
\begin{array}{l}
(\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD}) \models (m_0, pc_0) : \texttt{invokevirtual}\ m \\
\quad \text{iff} \quad \forall \delta \in \hat{TD}_{\{0,1\}}(m_0, 0) \colon \hat{TD}_\delta(m_0, pc_0) \neq \emptyset \Rightarrow \\
\qquad A_1 :: \cdots :: A_{|m|} :: B :: X \lhd \hat{S}_\delta(m_0, pc_0) : \\
\qquad \forall (\text{Ref } \sigma) \in B : \\
\qquad\quad m_v = \mathit{methodLookup}(m, \sigma) \\
\qquad\quad \forall \delta' \in \hat{TD}_\delta(m_0, pc_0) : \\
\qquad\qquad \{\delta'\} \subseteq \hat{TD}_{\delta'}(m_v, 0) \\
\qquad\qquad \{(\text{Ref } \sigma)\} :: A_1 :: \cdots :: A_{|m|} \sqsubseteq \hat{L}_{\delta'}(m_v, 0)[0..|m|] \\
\qquad\qquad \hat{TD}_{\delta'}(m_v, \mathsf{END}_{m_v}) \subseteq \hat{TD}_\delta(m_0, pc_0 + 1) \\
\qquad\qquad \forall ((\text{Ref } \sigma_X), T_X) \in \hat{E}_\delta(m_v) : \\
\qquad\qquad\quad \mathsf{HANDLE}_{(\hat{L}_\delta, \hat{S}_\delta \hat{E}_\delta, \hat{TD}_\delta)}((\text{Ref } \sigma_X), T_X, (m_0, pc_0)) \\
\qquad\qquad A :: \_ \lhd \hat{S}_{\delta'}(m_v, \mathsf{END}_{m_v}) : \\
\qquad\qquad\quad A :: X \sqsubseteq \hat{S}_\delta(m_0, pc_0 + 1) \\
\qquad \hat{L}_\delta(m_0, pc_0) \sqsubseteq \hat{L}_\delta(m_0, pc_0 + 1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD}) \models (m_0, pc_0) : \texttt{throw} \\
\quad \text{iff} \quad \forall \delta \in \hat{TD}_{\{0,1\}}(m_0, 0) \colon \hat{TD}_\delta(m_0, pc_0) \neq \emptyset \Rightarrow \\
\qquad B :: X \lhd \hat{S}_\delta(m_0, pc_0) : \\
\qquad \forall (\text{Ref } \sigma_X) \in B : \\
\qquad\quad \mathsf{HANDLE}_{(\hat{L}_\delta, \hat{S}_\delta \hat{E}_\delta, \hat{TD}_\delta)}((\text{Ref } \sigma_X), \hat{TD}_\delta(m_0, pc_0), (m_0, pc_0))
\end{array}
$$

Fig. 5. Flow Logic for method invocation and exception throwing

defined:

$$
\begin{array}{l}
\mathsf{HANDLE}_{(\hat{L}_\delta, \hat{S}_\delta \hat{E}_\delta, \hat{TD}_\delta)}((\text{Ref } \sigma_X), T_X, (m, pc)) \equiv \\
\quad \mathit{findHandler}(m, pc, \sigma_X) = \bot \Rightarrow \\
\qquad ((\text{Ref } \sigma_X), T_X) \in \hat{E}_\delta(m) \\
\quad \mathit{findHandler}(m, pc, \sigma_X) = pc_X \Rightarrow \\
\qquad \{(\text{Ref } \sigma_X)\} \sqsubseteq \hat{S}_\delta(m, pc_X) \\
\qquad \hat{L}_\delta(m, pc) \sqsubseteq \hat{L}_\delta(m, pc_X) \\
\qquad T_X \subseteq \hat{TD}_\delta(m, pc_X)
\end{array}
$$

The predicate uses the *findHandler* function to check if a given exception has a local handler. If no local handler exists the exception and the transaction depths, T_X, are saved in the cache; if a handler is found the stack, local heap, and transaction depths are copied forward to the handler.

The `throw` instruction (see Figure 5) simply throws an exception based on the object reference(s) found on top of the stack using the HANDLE predicate defined above.

Since transaction depths should only ever be stored and loaded and never used in computations (to facilitate verification of the correct use of transactions depths) a simple constant tracking analysis is sufficient although it can trivially be replaced with a more precise data flow analysis should the need arise. The simplicity of the present data flow analysis is most evident in the analysis of arithmetic operations (see Figure 6) since all operations result in INT, the top element of $\overline{\mathsf{Num}}$, being pushed onto the stack: $\{\mathsf{INT}\} :: X \sqsubseteq \hat{S}_\delta(m_0, pc_0 + 1)$.

$$
\begin{array}{l}
(\hat{H},\hat{L},\hat{S},\hat{E},\hat{TD}) \models (m_0,pc_0) : \texttt{numop}\ op \\
\quad \text{iff}\quad \forall\delta \in \hat{TD}_{\{0,1\}}(m_0,0) : \hat{TD}_\delta(m_0,pc_0) \neq \emptyset \Rightarrow \\
\qquad\quad A_1 :: A_2 :: X \triangleleft \hat{S}_\delta(m_0,pc_0) : \\
\qquad\quad \{\mathsf{INT}\} :: X \sqsubseteq \hat{S}_\delta(m_0,pc_0+1) \\
\qquad\quad \hat{L}_\delta(m_0,pc_0) \sqsubseteq \hat{L}_\delta(m_0,pc_0+1) \\
\qquad\quad \hat{TD}_\delta(m_0,pc_0) \subseteq \hat{TD}_\delta(m_0,pc_0+1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{H},\hat{L},\hat{S},\hat{E},\hat{TD}) \models (m_0,pc_0) : \texttt{if}\ cmpOp\ \texttt{goto}\ pc \\
\quad \text{iff}\quad \forall\delta \in \hat{TD}_{\{0,1\}}(m_0,0) : \hat{TD}_\delta(m_0,pc_0) \neq \emptyset \Rightarrow \\
\qquad\quad A_1 :: A_2 :: X \triangleleft \hat{S}_\delta(m_0,pc_0) : \\
\qquad\quad \widehat{cond}(cmpOp, A_1, A_2) \Rightarrow \\
\qquad\qquad \hat{S}_\delta(m_0,pc_0) \sqsubseteq \hat{S}_\delta(m_0,pc) \\
\qquad\qquad \hat{L}_\delta(m_0,pc_0) \sqsubseteq \hat{L}_\delta(m_0,pc) \\
\qquad\qquad \hat{TD}_\delta(m_0,pc_0) \subseteq \hat{TD}_\delta(m_0,pc) \\
\qquad\quad \widehat{cond}(\neg cmpOp, A_1, A_2) \Rightarrow \\
\qquad\qquad \hat{S}_\delta(m_0,pc_0) \sqsubseteq \hat{S}_\delta(m_0,pc_0+1) \\
\qquad\qquad \hat{L}_\delta(m_0,pc_0) \sqsubseteq \hat{L}_\delta(m_0,pc_0+1) \\
\qquad\qquad \hat{TD}_\delta(m_0,pc_0) \subseteq \hat{TD}_\delta(m_0,pc_0+1)
\end{array}
$$

Fig. 6. Excerpt of Flow Logic for arithmetic operations and conditionals

Conditionals are more interesting, here we define two abstract comparison predicates indicating whether the true and/or false branch may be taken:

$$
\begin{array}{l}
\widehat{cond}(cmpOp, A_1, A_2) \equiv \\
\quad (\mathsf{INT} \in A_1 \cup A_2) \vee (\exists n_1 \in A_1 \exists n_2 \in A_2 : cmpOp(n_1, n_2))
\end{array}
$$

This is another example of exploiting a reachability property to enhance the precision of the analysis.

We can now lift the analysis to cover whole programs. Here we must make sure that initial configurations are taken into account:

$$
\begin{array}{l}
(\hat{H},\hat{L},\hat{S},\hat{E},\hat{TD}) \models P \qquad \text{iff} \\
\quad \forall (m,pc) \in P.addresses : \\
\qquad m.instructionAt(pc) = \texttt{instr} \Rightarrow (\hat{H},\hat{L},\hat{S},\hat{E},\hat{TD}) \models (m,pc) : \texttt{instr} \\
\quad \forall \sigma \in P.classes\text{: } (\text{Ref } \sigma) \in \hat{L}_0(m_\sigma, 0) \ \wedge\ \{0\} \subseteq \hat{TD}_0(m_\sigma, 0)
\end{array}
$$

4.3 Semantic Correctness

In order to prove the semantic correctness of the analysis we take an approach similar to that of [6] and define *representation functions* for all semantic objects and a *correctness relation* between program configurations. We then show that the analysis preserves these under evaluation by establishing a *subject reduction* property for the analysis.

Since we do simple constant tracking, numbers are simply represented as themselves: $\beta_{\text{Num}}(n) = \{n\}$. In the analysis specification any arithmetic operation on numbers result in INT. Locations only make sense relative to a given heap, so the representation function for locations requires the relevant heap as an additional parameter: if $H(loc).class = \sigma$ then $\beta^H_{\text{ObjRef}}(loc) = \{(\text{Ref } \sigma)\}$. A value in Carmel is either a number or a location, thus the representation function for values: $\beta_{\text{Num}}(v)$ for $v \in \mathsf{Num}$ and $\beta^H_{\text{ObjRef}}(v)$ for $v \in \mathsf{ObjRef}$. Abstract objects are simply considered as maps from fields to abstract values: $\beta^H_{\text{Object}}(o)(f) = \beta^H_{\text{Val}}(o.f)$. Heaps map locations to objects. Since all objects of the same class are represented as the same abstract object, the representation function for heaps has to find all objects of a given class and then use the (least upper bound of the) representation of these objects:

$$\beta_{\text{Heap}}(H)(\text{Ref } \sigma) = \bigsqcup_{\substack{loc \in \text{dom}(H) \\ \beta^H_{\text{Val}}(loc) = (\text{Ref } \sigma)}} \beta^H_{\text{Object}}(H(loc))$$

Abstract stacks are sequences of abstract values leading to a straightforward representation function for stacks: $\beta^H_{\text{Stack}}(v_1 :: \cdots :: v_n) = \beta^H_{\text{Val}}(v_1) :: \cdots :: \beta^H_{\text{Val}}(v_n)$. Local heaps have a similarly simple representation function: $\beta^H_{\text{LocHeap}}(L)(x) = \beta^H_{\text{Val}}(L(x))$.

With the representation functions in place for the basic semantic domains, we can now lift these definitions to stack frames and semantic configurations. For this we use *correctness relations* that specify when a stack frame or semantics configuration is correctly represented in an analysis. First stack frames:

$$\begin{array}{lll} \langle \tau, m^{\tau_m}, pc, L, S\rangle \, \hat{\mathcal{R}}^H_{\text{Frame}} \, (\hat{L}, \hat{S}, \hat{TD}) & \text{iff} & \tau \in \hat{TD}_{\tau_m}(m, pc) \wedge \\ & & \beta^H_{\text{LocHeap}}(L) \sqsubseteq \hat{L}_{\tau_m}(m, pc) \wedge \\ & & \beta^H_{\text{Stack}}(S) \sqsubseteq \hat{S}_{\tau_m}(m, pc) \end{array}$$

In essence the above correctness relations simply states that the abstract representation of the semantic objects in the state frame should be contained in the analysis. The only thing to note is the use of the transaction depth in which the current method was called (the τ_m annotation on the method m) as a context for the analysis, e.g., $\hat{S}_{\tau_m}(m, pc)$. This is extended to cover the entire call stack (a sequence of stack frames) taking care to handle exception frames correctly by propagating exceptions that are not handled locally:

$$\begin{array}{l} F_1 :: \cdots :: F_n \, \hat{\mathcal{R}}^H_{\text{CallStack}} \, (\hat{L}, \hat{S}, \hat{TD}) \quad \text{iff} \\ \quad \forall i \in \{2, \ldots, n\} : F_i \, \hat{\mathcal{R}}^H_{\text{Frame}} \, (\hat{L}, \hat{S}, \hat{TD}) \wedge \\ \quad F_1 = \langle \tau_1, m_1^{\tau_{m_1}}, pc_1, L_1, S_1\rangle \Rightarrow F_1 \, \hat{\mathcal{R}}^H_{\text{Frame}} \, (\hat{L}, \hat{S}, \hat{TD}) \wedge \\ \quad F_1 = \langle \tau_X, loc_X, m_X, pc_X\rangle \Rightarrow (\beta^H_{\text{ObjRef}}(H(loc_X).class), \{\tau_X\}) \in \hat{E}_\delta(m_X) \end{array}$$

The correctness relation for an a running configuration is defined as follows:

$$\begin{array}{lll} \langle H, SF\rangle \, \hat{\mathcal{R}}_{\text{RunConf}} \, (\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD}) & \text{iff} & \beta_{\text{Heap}}(H) \sqsubseteq \hat{H} \wedge \\ & & SF \, \hat{\mathcal{R}}^H_{\text{Frames}} \, (\hat{L}, \hat{S}, \hat{TD}) \end{array}$$

Final configurations are particularly simple to handle:

$$\langle H, \tau, \langle \mathsf{Ret}\ v\rangle\rangle\ \hat{\mathcal{R}}_{\text{FinConf}}\ (\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD}) \quad \text{iff} \quad \beta_{\text{Heap}}(H) \sqsubseteq \hat{H}$$

The above is combined to the following definition for full configurations:

$$\begin{aligned} &C\ \hat{\mathcal{R}}_{\text{Conf}}\ (\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD}) \quad \text{iff} \\ &\quad C \in \mathsf{RunConf} \Rightarrow C\ \hat{\mathcal{R}}_{\text{RunConf}}\ (\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD}) \\ &\quad C \in \mathsf{FinConf} \Rightarrow C\ \hat{\mathcal{R}}_{\text{FinConf}}\ (\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD}) \end{aligned}$$

We are now in a position to state and prove that the analysis specification is semantically correct. This is done by establishing a subject reduction property for the analysis, i.e., by proving that the analysis result is invariant under execution:

Theorem 4.1 (Soundness) *If $P \in \mathsf{Program}$ with C_0 as an initial configuration, and $\mathcal{A} \models P$ such that $P \vdash C_0 \Longrightarrow^* C_1$ and $P \vdash C_1 \Longrightarrow C_2$ then $C_1\ \hat{\mathcal{R}}_{\text{Conf}}\ \mathcal{A}$ implies $C_2\ \hat{\mathcal{R}}_{\text{Conf}}\ \mathcal{A}$.*

Proof. By induction on the length of the evaluation sequence combined with case analysis and using a technical lemma establishing that the call-stack is well-formed. □

5 Verifying Well-Formedness

In this section we show how the analysis discussed in the previous section can be used to statically verify that the transactions in a Carmel program are indeed well-formed. Below we define a property of a program and an analysis of that program describing how the analysis can be used to guarantee that a program has well-formed transactions:

Definition 5.1 (Static Well-Formedness) *A program, P, is said to have* statically well-formed *transactions with respect to $\mathcal{A} = (\hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{TD})$ if and only if $\mathcal{A} \models P$ and for* `instr` $= m.instructionAt(pc)$ *the following holds:*

(i) `instr` $=$ `API.beginTransaction` $\Rightarrow \hat{TD}_{\{0,1\}}(m, pc) \subseteq \{0\}$

(ii) `instr` $=$ `API.commitTransaction`$\hat{TD}_{\{0,1\}}(m, pc) \subseteq \{1\}$

(iii) `instr` $=$ `API.abortTransaction` $\Rightarrow \hat{TD}_{\{0,1\}}(m, pc) \subseteq \{1\}$

(iv) *for all $m_\sigma \in P.entry$ and $pc \in \mathsf{PC}$ then $m_\sigma.instructionAt(pc) =$* `return` $\Rightarrow \hat{TD}_{\{0,1\}}(m_\sigma, pc) \subseteq \{0\}$

Note that since a program can only terminate by throwing an (uncaught) exception or by executing a return instruction in the initial entry point. Therefore it is sufficient to check that all `return`-instructions of all entry points can only be executed when no transaction is active. A more precise alternative

```
public void m_Alice(Alice)                          // $\hat{TD}_0$ | $\hat{TD}_1$
{                                                   //
  0: load r 0                                       // $\{0\}$ | $\emptyset$
  1: push s 41                                      // $\{0\}$ | $\emptyset$
  2: push s 1                                       // $\{0\}$ | $\emptyset$
  3: invokevirtual atomicWrapper(byte[], short)     // $\{0\}$ | $\emptyset$
  4: API.beginTransaction                           // $\{0\}$ | $\emptyset$
  5: load r 0                                       // $\{1\}$ | $\emptyset$
  6: push s 41                                      // $\{1\}$ | $\emptyset$
  7: push s 1                                       // $\{1\}$ | $\emptyset$
  8: invokevirtual atomicWrapper(byte,[], short)    // $\{1\}$ | $\emptyset$
  9: API.commitTransaction                          // $\{1\}$ | $\emptyset$
 10: return                                         // $\{0\}$ | $\emptyset$
}
```

Fig. 7. Invoking `atomicWrapper` in different contexts

would be to extend the analysis context for entry points to show whether or not an entry point was invoked from the system or from inside the program. In the latter case no special requirements are made on the `return`-instructions.

The following theorem shows that if the above property holds of a program and an analysis of that program then the program does indeed have well-formed transactions:

Theorem 5.2 *Let $P \in \mathsf{Program}$ and $\mathcal{A} \models P$ and let P have statically well-formed transactions with respect to $\mathcal{A}$, then P has well-formed transactions.*

Proof. Follows immediately from Theorem 4.1 and Definition 3.2 and 5.1. □

We illustrate the use of static well-formedness by applying it to the wrapper method shown in Figure 1 and discussed in Section 2. Figure 7 shows an entry point method (cf. Def. 3.1), called `m_Alice`, that invokes the wrapper method in different contexts, i.e., both inside an active transaction (lines 4–9) and outside any transaction (lines 0–3). The right-hand columns (in the comments) of the example programs in Figures 1 and 7 show the result of using a prototype implementation of the analysis on the program consisting of the methods `atomicWrapper`, `m_Alice`, and `dosomething` (not shown); it is assumed that `dosomething` does not use any of the transaction API instructions, and thus that it does not affect the result of the transaction flow analysis. In order for the program to have well-formed transactions we need to check four instructions, two in `atomicWrapper`: line 8 and 15, and two in `m_Alice`: line 4 and 9.

From the analysis results it is easily seen that

$$\begin{aligned}
\hat{TD}_0(\texttt{atomicWrapper}, 8) \cup \hat{TD}_1(\texttt{atomicWrapper}, 8) &= \{0\} \cup \emptyset \subseteq \{0\}\\
\hat{TD}_0(\texttt{m\_Alice}, 4) \cup \hat{TD}_1(\texttt{m\_Alice}, 4) &= \{0\} \cup \emptyset \subseteq \{0\}\\
\hat{TD}_0(\texttt{atomicWrapper}, 15) \cup \hat{TD}_1(\texttt{atomicWrapper}, 15) &= \{1\} \cup \emptyset \subseteq \{1\}\\
\hat{TD}_0(\texttt{m\_Alice}, 9) \cup \hat{TD}_1(\texttt{m\_Alice}, 9) &= \{1\} \cup \emptyset \subseteq \{1\}\\
\hat{TD}_0(\texttt{m\_Alice}, 10) \cup \hat{TD}_1(\texttt{m\_Alice}, 10) &= \{0\} \cup \emptyset \subseteq \{0\}
\end{aligned}$$

and thus that the transactions are indeed well-formed. It is also instructive to look at line 12 of `atomicWrapper`, which is the instruction that invokes the wrapped method `dosomething`: $\hat{TD}_{\{0,1\}}(\texttt{atomicWrapper}, 12) = \{1\}$, indicating that no matter what context `atomicWrapper` itself is invoked in it will ensure that `dosomething` is always invoked inside an active transaction and thus it is guaranteed that all updates in `dosomething` are performed atomically.

6 Conclusion

In this paper we have shown, and formally proved correct, a novel program analysis for guaranteeing that transactions in a Java Card bytecode program are well-formed. We believe that this will be very valuable to programmers since the analysis can be completely automated and gives good feedback as to where possible violations may occur.

By increasing the precision of the data flow component it is relatively straightforward to extend the analysis to cover situations where deeper nesting levels are allowed. A number of other interesting features relating to transactions, e.g., no exceptions in a transaction and no pin verification inside a transaction (see [7]), can easily be checked by simple extensions of the analysis described in this paper. This is left for future work.

A small experimental prototype has been implemented. Running time, for small examples similar in size to the example in this paper, is on the order of a few seconds. Future work includes implementing a proper prototype for benchmarking purposes.

Acknowledgement

The authors would like to thank Mikael Buchholtz and Flemming Nielson for reading and commenting on early drafts of the paper.

References

[1] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technoogy Transfer (STTT)*. To appear. Preprint available at ftp://ftp.cs.iastate.edu/pub/leavens/JML/sttt04.pdf.

[2] Stephen N. Freund and John C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'98*, pages 310–328, Vancouver, British Columbia, Canada, 1998. ACM Press.

[3] René Rydhof Hansen. Extending the Flow Logic for Carmel. SECSAFE-IMM-003-1.0. Available from http://www.doc.ic.ac.uk/~siveroni/secsafe/docs/, 2002.

[4] René Rydhof Hansen. Flow Logic for Carmel. SECSAFE-IMM-001-1.5. Available from http://www.doc.ic.ac.uk/~siveroni/secsafe/docs/, 2002.

[5] Renaud Marlet and Cédric Mesnil. Demoney: A Demonstrative Electronic Purse — Card Specification. SECSAFE-TL-007 (version 0.8). Available from http://www.doc.ic.ac.uk/~siveroni/secsafe/docs/, November 2002.

[6] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[7] Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman, and Jean-Louis Lanet. Enforcing High-Level Security Properties for Applets. In J.-J. Quisquater, P. Paradinas, Y. Deswarte, and A.A. El Kalam, editors, *Proc. of Smart Card Research and Advanced Application Conference, Cardis'04*, pages 1–16. Kluwer, 2004.

[8] Igor Siveroni. SecSafe. Web page: http://www.doc.ic.ac.uk/~siveroni/secsafe/, 2003.

[9] Igor Siveroni. Operational Semantics of the Java Card Virtual Machine. *Journal of Logic and Algebraic Programming*, 58(1–2):3–25, January–March 2004. Special issue on Formal Methods for Smart Cards.

[10] Igor Siveroni and Chris Hankin. A Proposal for the JCVMLe Operational Semantics. SECSAFE-ICSTM-001-2.2. Available from http://www.doc.ic.ac.uk/~siveroni/secsafe/docs/, October 2001.

[11] Jan Vitek, R. Nigel Horspool, and James S. Uhl. Compile-Time Analysis of Object-Oriented Programs. In *Proc. International Conference on Compiler Construction, CC'92*, volume 641 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.