Using formal methods to cultivate trust in smart card operating systems

Marjan I. Alberda\textsuperscript{a,b}, Pieter H. Hartel\textsuperscript{b,a,*}, Eduard K. de Jong Frz\textsuperscript{c}

\textsuperscript{a} Department of Computer Science, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, Netherlands
\textsuperscript{b} Department of Electronics and Computer Science, University of Southampton, S017 1BJ Southampton, UK
\textsuperscript{c} Integrity Arts Inc., San Mateo, USA

Abstract

To be widely accepted, smart cards must contain completely trustworthy software. Because smart cards contain relatively simple computers, and are used only for a specific class of applications, it is feasible to make the language used to program the software components focused and tiny. Formal methods can be used to precisely specify this language and to reason about properties of the language, which results in more trustworthy software. We explore this process by specifying the core of a proprietary systems programming language for smart card operating systems. We show how the specification obtained is used in proofs, and in the development of tool support.

Keywords: Smart card operating systems; Structural operational semantics; Software components; Proofs; Tool support

1. Introduction

The ITSEC requirements \cite{97} state that formal methods must be used in the construction of IT systems that are to be endowed with maximal trust. The term formal methods describes a mode of operation in the design, specification and implementation of a system which is characterised by the application of formalised methods of reasoning. Formal methods:

- help to clarify the exact requirements of a system;
- help early detection of mistakes and errors;
- make it possible to prove properties about a system;
- make it possible to prove the correctness of an implementation.

We present a formal specification of the core of one of the components of the Tosca\textsuperscript{3} (transaction oriented smart card architecture) smart card system as it is currently being developed by Integrity Arts Inc. It is our aim to demonstrate in this paper the benefits of using a formal approach in a setting that

\textsuperscript{3}Tosca is a trademark of Integrity Arts Inc.
is accessible to an audience that is not specialised in formal methods.

We give a detailed specification of the core of the proprietary systems programming language Clasp (compact language for abstract secure processors) that plays a key role in the Tosca system. The Clasp language is designed for secure execution and to generate a dense object code. The security aspects of the language will be highlighted in this paper; space precludes us from exploring its density properties.

Formal specifications of languages are not novel. An excellent introduction to the subject may be found in [10]. To our knowledge formal specifications as described in this book have not been used to help design and reason about smart card systems. This is the first contribution of this paper. The second contribution is that we indicate how a formal specification may be used to help develop support tools for the development process of a smart card system. We identify tools such as emulators, simulators and animators and give a code fragment from our simulator. The key role of a formal specification in producing these support tools is explained.

In Section 2, we discuss the general design of the Tosca system as a layered software architecture. In Section 3, we present an informal description of the core of Clasp, which is then followed by a formal specification. Section 4 describes how properties of the formal specification can be formulated and proved. It also describes how tools can be built to create the structural parts of simulators, emulators, animators and the actual implementation of components of a smart card system. This section also includes future work. Section 5 describes related work and the final section presents our conclusions.

2. A layered software architecture

Programs in smart cards are usually stored in ROM. This ensures that the programs cannot easily be tampered with. Unfortunately, this also makes smart cards inflexible, because the ROM cannot be changed after fabrication. To overcome this limitation, smart card operating systems need to support the downloading of executable code. To maintain the operating system integrity, the loadable executable code should be contained in recognisable sections of application code [2,11], which are usually referred to as 'applets'. In Tosca, the applet code is executed by a carefully crafted interpreter that makes sure that the applet code behaves properly. Later, we make more precise what 'proper behaviour' means. The provision of a downloadable applet facility makes smart cards flexible, without hampering the security.

The same idea has been used to build flexible Web browsers using Java [6]. Java enables applets to be downloaded from the server and to be executed by the client browser. Java code is compiled by the server into a byte code, which is subsequently interpreted. The Java interpreter can check that the downloaded applets only use the resources they are entitled to [17].

A smart card is not a Web browser, so that although in principle similar approaches can be followed, the practice is different. For one thing, smart cards are small. Web browsers may assume that there are megabytes of memory available. The specially designed Tosca smart card application language is to smart cards what Java is to the Web browser: application code in Tosca is compiled into loadable applets.

To cope with the limitations of smart cards, the Tosca interpreter is itself written in an interpreted code, called Clasp. This increases the code density and improves the level of trust beyond what is possible with a single level of interpretation. The time penalty incurred by using multiple levels of interpretation is not a problem for computers that perform cryptographic operations. A typical smart card transaction lasts for about 1s and the only compute intensive aspect is associated with the cryptography. By coding the compute intensive parts of cryptographic operations at the lowest level (directly in machine code), and by making these operations available as instructions at a higher level of interpretation, smart card systems can be efficient and secure.

Multiple levels of interpretation are a standard practice in computer architecture [14]. A typical CISC machine would have a micro code program that interprets machine instructions. The operating
Tosca programs compile into Tosca-object code for the Tosca-object interpreter (written in Clasp) Level 2

Clasp programs compile into Clasp-object code for the Clasp-object interpreter (written in Assembler) Level 1

Assembler generate machine code for the smart card CPU (built in hardware) Level 0

<table>
<thead>
<tr>
<th>Tosca programs</th>
<th>compile into</th>
<th>Tosca-object code for the Tosca-object interpreter (written in Clasp)</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clasp programs</td>
<td>compile into</td>
<td>Clasp-object code for the Clasp-object interpreter (written in Assembler)</td>
<td>Level 1</td>
</tr>
<tr>
<td>Assembler</td>
<td>generate</td>
<td>machine code for the smart card CPU (built in hardware)</td>
<td>Level 0</td>
</tr>
</tbody>
</table>

Fig. 1. Layered smart card software architecture.

system would interpret system calls, and a user program (such as the command language interpreter) might interpret a further set of instructions. The point here is that the problems that smart card software architects are facing can be solved using tried and tested ideas such as the multilevel abstract machine and client/server approaches.

Tosca’s multi-layered software architecture is schematically shown in Fig. 1. The Tosca development environment allows for the creation of trustable smart card applications on the basis of the Tosca operating system, standard applets and customer applets. The Tosca language is used to create the applets. Tosca applets are compiled into Tosca-object code, which is then interpreted by the Tosca-object interpreter.

The Tosca-object interpreter is written in Clasp, as are the standard library modules that are supplied with this interpreter. The Clasp language is a threaded code, interpreted language. It has been designed specifically for the provision of trustable code. It can be compiled into a dense and efficient interpreted code called Clasp-object.

Clasp-object interpreters are small assembler programs, written specifically for typical IC card processors, such as the 8051 and the 6805.

In the rest of this paper, we will focus on particular aspects of the Clasp language.

2.1. The compact and secure Clasp language

The clasp programming model is that of a stack machine. It works in the same way as FORTH in that it shares with FORTH the ability to extend the language. However, unlike FORTH, Clasp provides security and dense code, but does not support compiled routines and run-time dictionary lookup.

A Clasp program consists of a number of declarations, of which we shall only discuss the procedure declarations. Each procedure has a name and a body, there are no explicit parameters to the procedures. Arguments are passed implicitly via the stack. Return values are left on the stack. This makes programming in Clasp a bit awkward, but Clasp extensions have been developed to help overcome this limitation. A future extended definition of the language is under construction to provide syntactic support in this aspect. We shall not discuss these extensions here, and we shall not even look at procedure calls. Instead, we will focus on how the body of a procedure is executed. A body consists of a number of control statements and stack operators. Constructs are separated by semicolons as in Pascal. The control statements and the stack operators communicate exclusively via the stack. The stack is kept in the RAM of the smart card, which therefore limits the amount of space that can be used for the stack.

Let us concentrate on the basic mode of operation of the Clasp interpreter: it executes one statement or stack operator at a time, manipulating the stack as appropriate. Some important types of Clasp statements and operators are:

- The statement: \( n \) pushes the value \( n \) onto the stack. Values are bytes so they must be in the range 0...255.

- The statement: \( \text{test true part else false part end} \) tests the value that resides on the top of the stack. If this value is 1 the statements and operators of the true part are executed. The false part is executed if the value is 0.

- The operator: \( \text{pickn} \) duplicates a particular element of the stack.

- The operator: \( \text{dropn} \) removes a certain number of elements from the stack.

- The operator: \( \text{rotn} \) circularly rotates a particular portion of the stack.

- The operators: \( -, < \) replace the top two elements of the stack by a newly computed result.
This summary description shows two weaknesses. First, the Clasp language may seem rather a poor language. We should point out that we have not shown the complete language. For instance, I/O and memory operations and in particular the loop construct and procedures have not been described. We do not need those for the example that we will be discussing in detail. Furthermore, the description of these concepts is not difficult to add. Adding loops and procedures does not increase the number of different language constructs significantly. The complete language is kept small and focused which makes it easier to build language processors and to reason about programs written in the language.

The second point to make is that the summary description above is also vague in many ways. This is typical for informal language descriptions. It may be a source of considerable frustration both to the compiler writer and to the programmer, as neither knows exactly what the language designer intended, and problems created by such vagueness are in general difficult to address. In a trusted environment this vagueness cannot be tolerated, and a formal description of the programming language is called for.

2.2. The syntax of a representative subset of Clasp

The syntax of a programming language is usually described with a formal language, e.g. based on BNF, or the pictorial form of BNF: syntax diagrams. By using appropriately chosen names for the describing rules, the syntax may indicate what particular language constructs mean. It does not enforce that meaning, the names serve only as a suggestion. The formal semantics of the language, defined in addition to the syntax, is to provide that. Working with semantics may seem difficult and mathematical at first, because of the use of special symbols. Once adapted to the particular notation, using formal methods feels like programming.

Fig. 2 shows the abstract syntax of the subset of Clasp as described earlier. The syntactic category \( n \) represents numbers and \( s \) (and also \( s_1, s_2, s, \) and \( s_f \)) represent statements and stack operators.

It is customary to use an abstract rather than a concrete syntax for reasons of brevity. The difference between the two forms of syntax is that in an abstract syntax parentheses indicate how constructs are parsed. In a concrete syntax this information is encoded in the rules, which are therefore slightly more complicated. This rule encoded parsing information is not relevant for the description of the meaning of the constructs, and can be safely ignored.

Here is an example of a Clasp code fragment, with parentheses to enforce a correct parse:

\[
s = 5;(3;(2;(pickn; (2;(pickn;( <; test skip else 2;rotn end)))))) \]

It is possible to check that the statement above satisfies the syntactic requirements of Clasp. The meaning of the code fragment can only be guessed, which is not satisfactory. The next section introduces the formal aspects of describing the meaning of a code fragment such as that of (1).

3. Towards a formal description of the Clasp subset

In this section, the execution of the statement sequence in Clasp is presented and explained using English and diagrams (Section 3.1). Semantic rules are introduced (Section 3.2) and the derivation sequence of the Clasp example is presented (Section 3.3). The rules are modified and extended with the statement composition rules (Section 3.4). Finally, tags are added to distinguish between correct and incorrect behaviour (Section 3.5). Tables 2 and 3 summarise the final rules.

The application of formal rules makes an execution trace more readable, reduces ambiguity in-
3.1. Executing a Clasp statement sequence

This section presents the execution of a Clasp statement sequence using English and diagrams. It will show how elaborate an explanation can become when using these techniques instead of formal methods used later on.

Consider the statement sequence (1) again:

\[
5; 3; 2; \text{pickn}; 2; \text{pickn}; <; \text{test skip else} 2; \text{rotn end}
\]  (2)

The sequence consists of eight statements separated by the operator; . Brackets are omitted, for we assume the sequence is interpreted from left to right. All successive stack configurations in the computation are shown in Fig. 3. The stack grows downwards. Indexes of the elements are given to the right of the stack.

The first statement in the sequence pushes the number 5 onto the stack (b). The execution of the next two statements results in the numbers 3 (c) and 2, respectively, pushed onto the stack (d). After these steps, the sequence of statements is reduced to

\[
\text{pickn}; 2; \text{pickn}; <; \text{test skip else} 2; \text{rotn end}
\]  (3)

The statement \text{pickn} pops the stack, updates the index (e), and pushes a copy of the element with the index given by the popped element, that is, index 2, value 5 (f). The execution continues by pushing a 2 onto the stack (g). Again, the \text{pickn} statement pops the stack and updates the index (h). It pushes a copy of the element with the new index 2 (the popped element) onto the stack, i.e. the number 3 (i). The next statement, <, compares the top two elements of the stack. It pushes a 1 (representing true) onto the stack if the stack element with index 2 is less than the element with index 1 (the top element). Otherwise, < pushes a 0 (representing false). In the case of the example, < pushes a 0 for 5 \neq 3 (j). The sequence of statements is now reduced to

\[
\text{test skip else} 2; \text{rotn end}
\]  (4)

The \text{test} statement resembles an if...then...else construct, but it has no boolean expression. Instead, the
boolean on which the decision is based as to which branch to take, must have been pushed onto the stack before a test statement is encountered. The < statement in the sequence can be considered as the boolean expression in an if-then-else construct. The test statement inspects the top of the stack, finds a 0, pops it and subsequently executes the statements after the else. It results in stack configuration (k) and the sequence (2; rotn) to be executed. A 2 is pushed onto the stack (l). Like pickn, rotn pops the stack and updates the index (m). Then, it swaps the element with the value of the popped element as its new index and the element at this index minus 1. So, it swaps value 5 (index 2) and value 3 (index 1) (n).

Execution of the statement sequence results in stack configuration (n). If the initial statements of the sequence had been 3;5 instead of 5;3, the < would have pushed a 1 instead of a 0. The test statement would therefore have resulted in the execution of skip. This would have delivered the same final stack configuration (n), for skip does not change the stack.

The execution of the sequence

\[
2; \text{pickn}; 2; \text{pickn}; <; \text{test skip else 2; rotn end}
\]

(5)

will result in a stack with its top element larger than or equal to the element with index 2.

3.2. Introducing the formal rules

Instead of using pictures to illustrate the configuration of the stack and using English to explain what happens, formal rules can be used to specify the meaning of a sequence of statements more succinctly.

A rule has a standard formal notation. It consists of:
- a tuple \( \langle \text{statement}; \text{statements}, \text{stack} \rangle \), which represents a state in the execution,
- an arrow \( \Rightarrow \) to indicate a transition from this state to a second tuple,
- \( \langle \text{statements}, \text{stack}' \rangle \), the next state in the execution.

The concatenation \( \text{statement}; \text{statements} \) defines a sequence of statements, in which statement is the first statement. It is followed by the sequence statements. Just as \( n \) is used to denote any number and \( s \) to denote any sequence of statements, \( u \) (user stack) is used to denote any stack configuration (including the empty stack). The stack grows from right to left, and stack elements are concatenated using the operator \( ; \). The symbol \( \emptyset \) explicitly indicates the empty stack. The notation \( e \) specifies the empty statement sequence.

Consider the rule for the statement \( n \):

\[
\langle n; s, u \rangle \Rightarrow \langle s, n:u \rangle
\]

(6)

The rule states that the execution of the statement \( n \) followed by the statements \( s \), with stack configuration \( u \), results in a new stack in which statements \( s \) are to be executed with the new stack configuration \( n:u \). Here \( n \) is the new top. As an example consider the execution of the first three statements of sequence (2) in Section 3.1:

\[
\langle 5;3;2,\emptyset \rangle \Rightarrow \langle 3;2,5:\emptyset \rangle
\Rightarrow \langle 2,3:5:\emptyset \rangle \Rightarrow \langle e,2:3:5:\emptyset \rangle
\]

(7)

This derivation sequence shows three successive applications of rule (6). The numbers 5, 3 and 2 are pushed onto the stack respectively, resulting in the final stack configuration 2:3:5:\emptyset, in which 2 is the top of the stack and has index 1. All statements are executed. Note that a statement can always be followed by the empty statement (sequence) \( e \), i.e. 

\[ \text{statement} = \text{statement}; e \]

in which \( \equiv \) means semantically equivalent. This implies that \( e \) can be omitted. In (7), for instance, \( 5;3;2 \equiv 5;3;2;e \). There is no particular reason for omitting \( e \), other than saving space.

A rule's left-hand side is viewed as a template to which the statement sequence to be executed is matched. The semantics is thus syntax directed. The starting state in (7), \( \langle 5;3;2,\emptyset \rangle \), and the left-hand side of the rule (6), \( \langle n; s, u \rangle \), can be matched by considering \( n = 5 \), \( s = 3;2 \), and \( u = \emptyset \). Applying the rule results in the state \( \langle 3;2,5:\emptyset \rangle \). Note that \( 2 \equiv 2; e \) is used to match \( 2 \) in \( \langle 2,3:5:\emptyset \rangle \) of (7) with the statement sequence \( n; s \) in rule (6).

The rule for the statement pickn is given by

\[
\langle \text{pickn}; s, n:u \rangle \Rightarrow \langle s, u[n]:u \rangle
\]

(8)

The top of the stack on the left-hand side of the rule, \( n \), is used as an index in the stack \( u \). \( u[n] \) is the element in the stack with index \( n \) and replaces \( n \) as the top of the stack.
Consider the following example of using $\text{pickn}$:

\[
\text{test}: (\text{pickn}, 1:3:5) = (\text{~}, 3:3:5) \quad (9)
\]

By considering $n = 1$, $u = 3:5$, $s = \varepsilon$ and $\text{pickn} \equiv \varepsilon$, applying the rule results in the state $(\varepsilon, 3:3:5)$, as, after reindexing, 3 is the top of the stack $u$ in $n: u$. It has index 1 and so $u[n] = u[1] = 3$.

Writing the stack as $n: u$ on the left-hand side of the rule (8) forces the stack to consist of at least one element.

The rule for the statement $\text{rotn}$ is given by

\[
\langle \text{rotn}; s, n: u \rangle \quad \Rightarrow \quad \langle s, u[u[n]/1, u[1]/2, \ldots, u[n-1]/n] \rangle \quad (10)
\]

This rule introduces a new notation for stack manipulations. A stack element at index $i$ to be substituted by a value $v$ is denoted by $u[v/i]$; a number of simultaneous substitutions may be denoted by including them as a comma separated list inside the square brackets. Whenever the value $v$ is defined as an element of $u$, it is extracted from $u$ before any of the substitutions has taken place.

The resulting stack in (10), denoted by the outer $u[\ldots]$, is obtained by simultaneously substituting each element with index $1 \leq i \leq n$, by its index-wise predecessor in the initial stack, denoted by the inner $u[i-1]$, and the top by $u[n]$. In effect, all elements with index $i < n$ are rotated one place downwards in the stack and the element with index $n$ is the top of the new stack.

When $n = 2$, the rule defines a swap function, as was shown in Section 3.1. The rule precisely defines the behaviour of the statement $\text{rotn}$ for all $n$ (i.e. for all ‘correct’ $n$, for stack underflow and overflow are not taken into account, yet). As an example, consider:

\[
\langle \text{rotn}, 3:1:2:3:4 \rangle \Rightarrow \langle \varepsilon, 3:1:2:4 \rangle \quad (11)
\]


The $\text{test}$ statement has two possible executions. If a 0 resides on top of the current stack, the $false$ part after the keyword $else$ is executed. If the current top has value 1, $\text{test}$ continues by executing the $true$ part. Two rules define the behaviour of the state-

\[
\begin{align*}
\langle \text{test s, else s_f end; s, b: u} \rangle & \Rightarrow \langle s, s, u \rangle & (\text{test}^{\text{True}}) \\
\langle \text{test s, else s_f end; s, b: u} \rangle & \Rightarrow \langle s_f, s, u \rangle & (\text{test}^{\text{False}})
\end{align*}
\]

The rules introduce the notion of a precondition, or premise. A premise is a condition on the transition, i.e. the transition can only take place if the premise is true. The premise is written above the horizontal line and the actual transition from state to state, or derivation step, is written below it. By writing $b: u$ as the stack configuration on the rules’ left-hand sides, the top of the stack, $b$, can be used in the premise. If $b = 1$, the rule named $(\text{test}^{\text{True}})$ matches, execution continues by executing $s_f; s$. If $b = 0$, rule $(\text{test}^{\text{False}})$ is applied, and the sequence $s_f; s$ is to be executed. The rules are mutually exclusive but not complete, as the behaviour in case $b > 1$ is not defined. This will be amended in Section 3.5, but for now, it is assumed that the boolean $b$ is either 0 or 1.

The statement $<$ also has two possible execution paths. Depending on the outcome of the comparison of the top two elements of the stack, either a 0 or a 1 is pushed onto the stack. Having already introduced the notion of a premise, the rules for the statement $<$ are included (see Table 1) without further explanation.

The rule for the statement $\text{skip}$ is given by

\[
\langle \text{skip; s, u} \rangle \Rightarrow \langle s, u \rangle \quad (12)
\]

The execution of $\text{skip}$ does not affect the stack. If a $\text{test}$ statement was to be defined with only one execution path, for instance, $\text{skip}$ can be used to constitute the empty statement sequence as the other path.

Table 1 summarises the named rules introduced in this section.

3.3. Execution of the Clasp statement sequence using the formal rules

Consider again the statement sequence (2). Its execution trace is derived by the successive application of the formal rules (Table 1) to its constituents. This derivation sequence is given below. Each step is
annotated with the rule that was applied.

3.4. Adding the composition rules

\[ (5; 3; 2; pickn; 2; pickn; <; test skip else 2; rotn end, 0) \Rightarrow (\text{number}) \]
\[ (5; 3; 2; pickn; 2; pickn; <; test skip else 2; rotn end, 5:0) \Rightarrow (\text{number}) \]
\[ (2; pickn; 2; pickn; <; test skip else 2; rotn end, 3:5:0) \Rightarrow (\text{pickn}) \]
\[ (pickn; 2; pickn; <; test skip else 2; rotn end, 2:3:5:0) \Rightarrow (\text{number}) \]
\[ (2; pickn; <; test skip else 2; rotn end, 5:3:5:0) \Rightarrow (\text{pickn}) \]
\[ (<; test skip else 2; rotn end, 2:5:3:5:0) \Rightarrow (\text{< Fair}) \]
\[ (<; test skip else 2; rotn end, 3:5:3:5:0) \Rightarrow (\text{test Fair}) \]
\[ (test skip else 2; rotn end, 0:3:5:0) \Rightarrow (\text{number}) \]
\[ (2; rotn, 3:5:0) \Rightarrow (\text{rotn}) \]
\[ (rotn, 2:3:5:0) \Rightarrow (\text{pickn}) \]
\[ (\text{< Fair}) \Rightarrow (\text{test Fair}) \]
\[ (rotn, 213:150) \Rightarrow (\text{rotn}) \]

The derivation sequence contains all semantically relevant information in only 11 formal lines. It shows each configuration of the stack and what statements are left to be executed. The derivation sequence shows at least two of the advantages of using formal rules instead of English and pictures, namely compactness and clarity. Compactness and clarity help to make precise the behaviour of an execution or system, and therefore help to detect mistakes and errors.

In Section 3.2, the general form of a rule was defined as follows:

\[ \langle \text{statement}, \text{statements}, \text{stack} \rangle \Rightarrow \langle \text{statements}, \text{stack}' \rangle \]

This means that after executing statement, and perhaps changing the stack, statements are left to be executed. The first statement is of interest, but also the fact that statements follow it (though not their

| Table 1 |
The simplified formal rules of the subset of Clasp |

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle n; s, u \rangle \Rightarrow \langle s, n; u \rangle )</td>
<td>(number)</td>
</tr>
<tr>
<td>( \langle \text{skip}; s, u \rangle \Rightarrow \langle s, u \rangle )</td>
<td>(skip)</td>
</tr>
<tr>
<td>( \langle \text{test s, else s, end}; s, b; u \rangle \Rightarrow \langle s; s, u \rangle )</td>
<td>(test True)</td>
</tr>
<tr>
<td>( \langle \text{test s, else s, end}; s, 0; u \rangle \Rightarrow \langle s; s, u \rangle )</td>
<td>(test False)</td>
</tr>
<tr>
<td>( \langle &lt;; s, n_1; n_2; u \rangle \Rightarrow \langle s, 1; u \rangle )</td>
<td>(&lt; True)</td>
</tr>
<tr>
<td>( \langle &lt;; s, n_1; n_2; u \rangle \Rightarrow \langle s, 0; u \rangle )</td>
<td>(&lt; False)</td>
</tr>
<tr>
<td>( \langle \text{pickn}; s, n; u \rangle \Rightarrow \langle s, u[n]; u \rangle )</td>
<td>(pickn)</td>
</tr>
<tr>
<td>( \langle \text{rotn}; s, n; u \rangle \Rightarrow \langle s, u[n]/1, u[1]/2, \ldots, u[n-1]/n \rangle )</td>
<td>(rotn)</td>
</tr>
</tbody>
</table>
precise contents). A different approach from the one used in Section 3.2 is to only concentrate on the individual statements. This allows for separation of concerns, bearing in mind the principle "if you can separate then separate". Sequencing and the actual computation are the two concerns here. Separation of the two will lead to shorter derivations and it is more extensible.

The rule for the statement n has been given by $(n; s, u) \Rightarrow (s, n; u)$. It is now defined as:

$$(n, u) \Rightarrow (e, n; u)$$

Hence, we are no longer concerned with the statements that follow n. The final tables of Section 3 summarises the adapted rules (with tags to be introduced in Section 3.5).

A mechanism is needed for the newly defined rules to be applied to a sequence of statements as before. For this purpose, two composition rules are introduced:

$$
\begin{align*}
(s_1; u) & \Rightarrow (e, u') \\
(s_1; s_2, u) & \Rightarrow (s_2, u') \\
(s_1, u) & \Rightarrow (s'_1, u') \land s'_1 \neq e \\
(s_1; s_2, u) & \Rightarrow (s'_1; s_2, u')
\end{align*}
$$

The first rule states that if the execution of $s_1$ will lead to a state in which there are no more statements to be executed (e) and in which the new stack is $u'$ (this includes the possibility that $u$ and $u'$ are the same, as would be the case after the execution of skip), the statement sequence $(s_1; s_2, u)$ changes to $(s_2, u')$, i.e. to a state in which $s_2$ is to be executed with the new stack configuration $u'$. The second rule differs from the first rule in that $s_1$ is now a compound statement and the one step execution does not lead to $e$ (which is made explicit by including $(s'_1 \neq e)$ in the premise), but rather to the statements $s'_1$. Here, the new stack configuration is also denoted by $u'$ for which same reasoning holds as for $u'$ in the first rule. The statement sequence $(s_1; s_2, u)$ now continues as $(s'_1; s_2, u')$.

To summarise, rules are now of the form

$$(statement, stack) \Rightarrow (e, stack')$$

and the composition rules were added to be able to reason about a sequence of statements. We have separated sequencing from doing the stack operations.

### 3.5 Adding tags

So far, it was assumed that statements that affect the stack can always be executed. In an implementation, this cannot be assumed. When using a stack, two kinds of errors may occur: stack overflow and stack underflow. The first occurs when a statement pushes data onto a full stack, or when the stack is addressed with an index that is larger than its size. Here, $(number)$ is the only rule that pushes data onto the stack. Several other rules may use an 'incorrect' index. A mechanism is needed to trap this error. The other error, stack underflow, occurs when an empty stack is popped, or when the stack is addressed with an index less than 1. Several rules may cause this error by using an 'incorrect' index. The rule $(<)$ compares the top two elements of the stack, but what happens when the stack is only one element large or even empty? A mechanism must be present to handle these errors, for rules must always be precise.

For this purpose, the notion of a tag is introduced, and each state is extended with a tag. Consequently, the general form of the rules is:

$$(statement, stack, ok) \Rightarrow (e, stack, tag)$$

in which tag is either ok or nok (not ok). Note that the tag on the left-hand side is always ok. If $tag = nok$ in the current state, no rule should match, for, in that case, an error has occurred and execution should be stopped (an exception handling mechanism is provided in Clasp, but limitation of space precludes us from explaining that here).

The execution of statement $n$ is now defined by two rules: one defining its behaviour when no stack overflow will occur, and one defining it when it will:

$$
\begin{align*}
\#u < \text{max} & \Rightarrow (n, u, ok) \Rightarrow (s, u, ok) \\
\#u \geq \text{max} & \Rightarrow (n, u, nok) \Rightarrow (s, u, nok)
\end{align*}
$$

Here, $\#u$ is the number of elements in the stack. If this number is greater or equal to a certain
Table 2
Part 1 of the formal rules of the subset of Clasp with the composition rules and tags added

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#u ≥ max</td>
<td>( \langle n, u, ok \rangle \Rightarrow \langle e, u, nok_{full} \rangle ) (number/full)</td>
</tr>
<tr>
<td>#u &lt; max</td>
<td>( \langle n, u, ok \rangle \Rightarrow \langle e, n; u, ok \rangle )</td>
</tr>
<tr>
<td>skip, u, ok ⇒ ( e, u, ok )</td>
<td>(skip)</td>
</tr>
<tr>
<td>#u &lt; 1</td>
<td>( \langle test s_i \ else s_f \ end, u, ok \rangle \Rightarrow \langle e, u, nok_{empty} \rangle ) (test/empty)</td>
</tr>
<tr>
<td>test s_i else s_f end, u, ok ⇒ ( e, u, nok_{empty} )</td>
<td>( b &gt; 1 )</td>
</tr>
<tr>
<td>test s_i else s_f end, b: u, ok ⇒ ( e, u, nok_{empty} )</td>
<td>( b = 1 )</td>
</tr>
<tr>
<td>test s_i else s_f end, b: u, ok ⇒ ( s_i, u, ok )</td>
<td>( b = 0 )</td>
</tr>
<tr>
<td>( s_{i+1}, u, ok \rangle \Rightarrow \langle e, u', ok \rangle</td>
<td>(( i' ))</td>
</tr>
<tr>
<td>( s_{i+1}, s_i, u, ok \rangle \Rightarrow \langle s_i, u', ok \rangle</td>
<td>(( i' ))</td>
</tr>
<tr>
<td>( s_i, u, ok \rangle \Rightarrow \langle s_i', u', ok \rangle</td>
<td>(( i' ))</td>
</tr>
<tr>
<td>( s_i, s_i', u, ok \rangle \Rightarrow \langle s_i, s_i', u', ok \rangle</td>
<td>(( i' ))</td>
</tr>
<tr>
<td>( s_i, s_i', u, ok \rangle \Rightarrow \langle s_i, u', nok \rangle</td>
<td>(( i' ))</td>
</tr>
<tr>
<td>( s_i, s_i', u, ok \rangle \Rightarrow \langle e, u', nok \rangle</td>
<td>(( i' ))</td>
</tr>
</tbody>
</table>

The predefined number `max` which denotes the maximum stack size, rule (number/full) applies, and the tag is set to `nok`. The subscript `full` is used to distinguish between different kinds of errors. If #u < max, the statement is executed as before, i.e. \( n \) is pushed, and the tag is not changed. Note that the two rules are mutually exclusive and that they completely cover all possibilities. This is a requirement when proving properties (Section 4.1).

In Section 3.2, the value `b` used in the premise of the two `test` rules was assumed to be either 0 or 1. The two rules are mutually exclusive, but not complete. We now have a mechanism with which it is possible to add another rule to replace the ad hoc assumption, and that is to be used when \( b > 1 \). This new rule is given by:

\[
\langle test s_i \ else s_f \ end, b: u, ok \rangle \Rightarrow \langle e, u, nok_{errb} \rangle \] (test/errb)

In addition, a rule is needed here to deal with the error of stack underflow. If the number of elements in the stack is less than 1, the statement cannot be executed. Execution is therefore stopped, and the tag is set to `nok`:

\[
\langle test s_i \ else s_f \ end, u, ok \rangle \Rightarrow \langle e, u, nok_{empty} \rangle \] (test/empty)

The other two rules, (test/True) and (test/False), are as before, maintaining the `ok`-status, and are included in the two final tables, Tables 2 and 3. Note that #u ≥ 1 is not included in the premises of these two rules. The construction `b: u` implicitly denotes that there must be at least one element in the stack.

In the `pickn` statement, \( n \) is used as an index in the stack. The simplified rule (pickn) is now extended with a premise to only match when the index \( n \) is 'correct'. Two rules are added to define the transitions to error states (see Tables 2 and 3).

The form of the rules introduced in Section 3.4 makes the rules more extensible than the form of the simplified rules introduced in Section 3.2. If we had extended the simplified rules with the tags, Tables 2 and 3 would be twice as big. We would have had to define the behaviour of statements in \( \langle statement; statements \rangle \) for every possible execution of statement.

Table 2 and 3 summarise all the rules with the tags and the composition rules added. Note that the tag `nok` in the premise of the composition rule (\( i' \)) is used to pass the tag `nok` with the message `empty`, `full` or `errb` (hence the \( x \)) to the transition below the horizontal line.

4. Using the formal description

We have discussed the formal description of the subset of the Clasp language. The rules precisely describe, in a clear and compact way, what happens in a normal situation and in case some error occurs (proper behaviour). In this section, this formal description is used. First, it will be shown how it can be used to prove properties of the language. In the second part, a more general view is given on using
the formal description in building support tools for the software development process.

4.1. Proving properties

Security is an important aspect of a smart card system. Runtime errors like stack overflow, stack underflow and non-determinism of code impinge on the security. Trust can be gained by proving that these errors will not occur, or that if they do, they will be dealt with. The formal rules allow for such proofs. Since the rules are precisely the formal specification of the language Clasp, properties proven using the rules are properties proven for Clasp programs.

All proofs follow the same strategy [10] (this still holds when the subset is extended with loop constructs): prove that the property holds for all derivation sequences of length 0. Then prove the property holds for all other derivation sequences as follows: assume that the property holds for all derivation sequences of length at most k (this is the induction hypothesis) and show that it holds for derivation sequences of length k + 1.

Now, it is proven that stack overflow will not occur, i.e. execution continues if execution of the current statement will not cause the stack to overflow, and execution is stopped with the tag set to nok if it does.

Assumption: Execution is started with 

\[ \text{#stack} \leq \text{max} \]

Throughout the proof stack is used to denote any stack, and, as before, max is used to denote the maximum stack size.

Case k = 0: The property 

\[ \text{#stack} \leq \text{max} \]

still holds, for if no statements are executed, 

\[ \text{#stack} \leq \text{max} \]

holds by the initial assumption.

Case k + 1: Every possible statement to be executed as the (k + 1)th statement must be looked at:
Subcase \( n \): When \( n \) is encountered, \( \# \text{stack} \leq \text{max} \) holds by induction. Now, suppose there is stack overflow danger, i.e., \( \# \text{stack} = \text{max} \). Then, rule \((\text{number}^{\text{all}})\) applies and deals with this in the proper way, i.e., execution is stopped and the tag is set to nok. There is no other rule that matches this situation. The other possibility is that \( \# \text{stack} < \text{max} \). Now \((\text{number}^{\text{ok}})\) applies, and \( n \) is pushed. Consequently, the stack will grow, but \( \# \text{stack} \leq \text{max} \) still holds.

Subcase skip: Again, by induction this statement will be encountered with \( \# \text{stack} \leq \text{max} \). skip does not change the stack, so \( \# \text{stack} \leq \text{max} \) still holds.

Subcase test: By induction, \( \# \text{stack} \leq \text{max} \) holds when encountering statement test. Here, \( \text{stack} = b:u \), and in all the test rules except \((\text{test}^{\text{emptY}})\), one element is taken off the stack, namely \( b \). In effect \( \# \text{stack} \leq \text{max} \) still holds. \((\text{test}^{\text{emptY}})\) does not change the stack so, because \( \# \text{stack} \leq \text{max} \) holds before this rule is applied, it still holds.

Subcase: In all; rules, the single step in the premise is the last step in the induction, i.e., it is the step from \( k - 1 \) to \( k \). So, after execution of the premise \( \# \text{stack} \leq \text{max} \) holds. Now, none of the; rules thereafter change the stack. Therefore, \( \# \text{stack} \leq \text{max} \) still holds.

Subcase pickn: By induction \( \# \text{stack} \leq \text{max} \) holds when encountering the statement pickn. \((\text{pickn}^{\text{emptY}})\) does not change the stack, and \((\text{pickn}^{\text{nk}})\) pops an element, so, in both cases, \( \# \text{stack} \leq \text{max} \) still holds after execution. \((\text{pickn}^{\text{nk}})\) replaces the top element by another element, so, again, if by induction \( \# \text{stack} \leq \text{max} \), it still holds now.

Subcase rotn: Similar to pickn, for it also replaces (just more) elements in the stack, and does not change the size of the stack. So, \( \# \text{stack} \leq \text{max} \) still holds after executing the statement.

Subcase \( \langle <\text{emptY} \rangle \) does not change the size of the stack. The other two rules replace the top two elements by one element and therefore do not increase the number of elements in the stack (in fact, they are decreasing it). So, in each case, \( \# \text{stack} \leq \text{max} \) still holds after execution.

The property is now proven for all possible statements to be executed next. This finishes the proof for case \( k + 1 \). Because \( k \) can be any natural number, the property is proven for any sequence length.

The above proof constitutes an informal proof, in the sense that it does not use any formal notation as is used in a formal proof. It is just as thorough though.

It was proven that the error stack overflow will be dealt with. A similar proof can be given for the error stack underflow. Instead of ensuring that \( \# \text{stack} \leq \text{max} \) holds all the time, it must be ensured that \( \# \text{stack} \geq 0 \) holds all the time. Execution is started with \( \# \text{stack} \geq 0 \) (assumption). As an example, consider the subcase for statement rotn. Using the induction hypothesis, \( \# \text{stack} \geq 0 \) holds when encountering a statement rotn. Suppose that \( \# \text{stack} = 0 \), and popping an element will cause stack underflow. This situation is intercepted by rule \((\text{rotn}^{\text{emptY}})\). Apart from ensuring that there is a rule that matches, it also needs to be ensured that there is no other rule that matches. Indeed this is the case, for it can immediately be seen that the other rules all write their starting stack configuration as \( n:u \), i.e., at least one element on the stack. The other possibility to cause stack underflow is when \( n \) is not right, i.e., \( n < 2 \), for at least two elements will be needed in order to do a rotation. Here, rule \((\text{rotn}^{\text{nk}})\) matches, which also traps the possibility of \( n > \# \text{stack} \). Note that the two premises of \((\text{rotn}^{\text{nk}})\) do not have to be included in the rule \((\text{rotn}^{\text{emptY}})\). The only other possibility left is that \( n \) is right. In this situation, only rule \((\text{rotn}^{\text{nk}})\) matches. Note also that the premise \#u \( \geq 2 \) need not be added. If \#u \( \leq 1 \) then \( n \) would have to be either 0 or 1, because \( n \leq \#u \) is also part of the premise. The premise \( n \geq 2 \) would not be true, and using the boolean construct the premise as a whole would not be true.

In a similar way, other properties that are important to smart card software can be proven [10]. Determinism of the language is such a property. A formal proof is not given here. Informally, one should prove that every possible execution for every possible statement has one, and only one, matching rule that defines its behaviour. This is not difficult to prove for the subset of the Clasp language, i.e., the rules in Tables 2 and 3. For instance, the rules for the statement \( n \) define the only two possible executions, and they are mutually exclusive. Again, induction can be used to prove the property for compound statements.

New rules or new restrictions in the language can also be defined. Consider, for instance, the possibil-
ity of byte overflow. The only statement that pushes a number onto the stack, namely \( n \), cannot cause overflow, for \( 0 \leq n \leq 255 \) by definition. The statement \(<\) also deals with bytes, for it compares two bytes. The rules, however, define its behaviour: either a 0 or a 1 is pushed onto the stack. So, \(<\) will not cause any problems. Of course, the inequality operators use arithmetic operators, and these operators should also be 'checked'. Our subset of Clasp has \(-\) as its only arithmetic operator, so, this is the only operator that needs to be looked at. We now do the reverse of what we did before: we design a rule for this arithmetic operator, in such a way, that it will behave correctly. The rules can be defined as:

\[
\begin{align*}
\text{Rule } &\sim\textempty\text{ simply ignores byte overflow by using the } \text{mod} \text{ function. An alternative approach is to generate an exception (Clasp includes exception handling, but it is not shown here). The point here is that any rule can be defined, and a choice must be made when designing a language.}
\end{align*}
\]

4.2. Tool support

A formal specification can play an important role in the software engineering process. Fig. 4 identifies the major 'components' in the process. We use the term component here as a general term for such diverse items as the concepts on which a system is designed, the specification of the system, and also the documentation, implementation and 'derived components'. The latter includes various tools, such as simulators of the system. Fig. 4 shows only some of the many relationships between the components. For example the connection between concept and specification represents an iterative process of design and redesign. The connection between specification and derived components represents program generation tools, and programming and design efforts.

A formal description of the subset of the Clasp language has been given and its use demonstrated. This is just one component of the software for smart cards. Any system is as weak as its weakest component so there is a need for formally specifying the complete Clasp language, and also Tosca, the compilers, the interpreters and the assembler. Then we can begin to reason about the system as a whole. We have formally specified a substantial part of the Clasp language, its compiler and interpreter. Currently we are working on the specification of Tosca and on completing the Clasp specifications.

Work on the components should be supported by adequate tools, which basically take as input the specification and produce as output a derived component. The following are useful derived components, which implement all or part of the functional behaviour of the component. The non-functional behaviour (that is how much resources are being used) of the derived components is generally different from the non-functional behaviour of the component itself.

- An animator implements the functional behaviour as specified, and supports interaction. The key feature of an animator is that it allows the user to inspect and even change data whilst the data are being processed.
- A simulator implements the functional behaviour as specified, or an abstraction of the behaviour, and supports statistics gathering like timing.
- An emulator implements precisely the functional behaviour as specified, and is often used as a prototype.

These definitions are not static, and vary considerably across different fields of computer science. Tool support is necessary to keep the development of these derived components, the specification, the documentation and the testing and verification procedures, and the real component in step.
A formal specification in the style that we have presented in the previous section is abstract in the sense that it only deals with the functional behaviour. At the same time it is detailed because it describes precisely what should happen in a normal situation and also in case some error has occurred. Our formal specification can thus serve as a starting point for building the derived components as well as the real implementation. The programming activity involved should be mainly concerned with choosing the right programming language and with creating the non-functional behaviours desired for the derived and real components. The functional behaviour is always the same.

We have built a simple tool to generate the major part of the animator from our Clasp specification. The animator is written in the functional programming language Miranda\textsuperscript{5} [15]. We are now in the process of building a generator for simulators. Here is a fragment of such a simulator, written in C:

```c
void test(statement *s, stack *u, tag *o) {
    if (o->ok_nok_enum == Ok) {
        if (stack_size(u) < l) {
            s->statement_enum = Empty;
            tag_set_nok(o, Nok, "stackempty");
        } else {
            int b = stack_pop(u);
            if (b > 1) {
                s->statement_enum = Empty;
                tag_set_nok(o, Nok, "errb");
            } else if (b == 1) {
                *s = s->statement_union.test.test_time;
                tag_set_ok(o, Ok);
            } else if (b == 0) {
                *s = s->statement_union.test.test_false;
                tag_set_ok(o, Ok);
            }
        }
    }
}
```

The structures `statement`, `stack` and `tag` correspond to the statements `s`, the stack `u` and the tag `ok`/`nok` of the specification of Tables 2 and 3. The function `test` corresponds to the four rules \((\text{test}^{\text{empty}}), (\text{test}^{\text{errb}}), (\text{test}^{\text{true}})\) and \((\text{test}^{\text{false}})\).

Without going into the detail of the C fragment, we can see that the structure of the function reflects that of the four rules from the specification. The first conditional makes sure that the `test` statement is only executed if the `tag` is `ok`. The next conditional checks whether the stack contains at least one element. If this is the case, the final conditionals distinguish between the case that the boolean value `b` on the top of the stack represents a non-boolean value, true or false.

The result of executing the `test` statement is recorded by modifying the structures that are passed to the function `test`. This is the only major difference between the specification and the simulator. It represents our efforts to make the simulator run fast, whilst maintaining the appropriate functional behaviour.

5. Related work

An approach similar to ours in the domain of building a high integrity compiler is described by Stepney [12]. Her compiler is for a small general purpose language, which by coincidence is also called Tosca.

Much work has been done to develop formal methods specifically for reasoning about protocols. The specification formalism LOTOS [16] is widely used in this area. The protocols that are used in smart card systems could also be modelled using formalisms such as LOTOS but we do not know whether that has been done. The work at GMD [4, 9] uses a Petri net based method to perform model checking on the protocols used in the STAR-COS and STAR-MOD systems [13].

In our own previous work, we have looked at the modelling of a smart card protocol with a view to proving liveness of that protocol [7]. We have recently built an animation system [5] for the protocols that are defined in the CEN European standard Inter Sector electronic purse [3]. This work explores the use of the visual programming language Prograph\textsuperscript{6} [1] as an animation support system. The connection between a formal specification of the

\textsuperscript{5} Miranda is a trademark of Research Software Ltd.

\textsuperscript{6} Prograph is a trademark of Pictorius Inc.
protocols and the production of the animation is still to be made.

6. Conclusions

Using a formal approach to building trustworthy software is not widely practiced in the smart card community. This is to some extent surprising: the reason why smart cards are used is because they rely on cryptographic protocols. Cryptology is a branch of mathematics that is devoted to proving properties of cryptographic algorithms and protocols. In practice, when it comes to implementing these protocols in smart cards, the attention shifts to more technical issues, such as getting the implementation to work and making it work with limited resources, leaving correctness assurance aside.

We have shown that formal methods can profitably be used to build trustworthy components for a smart card system. Our main argument is that smart cards are tiny, and so the languages used to program the software components should be highly focused and small. This makes it feasible to specify such languages formally. Programs written in such languages have a well-defined meaning. They are therefore amenable to formal reasoning. We have shown how properties can be proved, when given a small subset of the language that plays an important role in our smart card system. However, the methods used can be applied to any language or system. We have discussed the necessity of tool support in smart card software engineering. By giving a sample component of a simulator we have shown how formal rules can be used to help build suitable tools.

Acknowledgements

We thank Hugh McEvoy, Michael Butler, Andy Gravell, Jon Mountjoy and the referees for their comments on a draft version of the paper.

References

Pieter H. Hartel received the Master's degree in Mathematics and Computer Science from the Free University of Amsterdam in 1978 and the Ph.D. degree in Computer Science from the University of Amsterdam in 1989. He has worked at CERN in Geneva and the University of Nijmegen and is currently a Senior Lecturer at the Universities of Southampton and Amsterdam. His current research interests are the theory of programming languages, programming language/compiler design, high integrity compilation/computer architecture and trusted systems design.

Eduard de Jong received the Master's degree in Computer Science in 1989 from the University of Amsterdam. He started his career in smart cards as the software architect in David Chaum's electronic purse project. In 1992, he co-founded QC Technology BV of Zaandam, Netherlands. In 1995, he co-founded Integrity Arts Inc. of San Mateo and joined the company as chief scientist in June 1996. He has applied for patents on a software architecture for secure transactions and electronic value transfer systems. His research interests are in high integrity software development and computer architecture for embedded systems.

Marjan I. Alberda is finishing her studies in Computer Science at the University of Amsterdam, for which she hopes to receive the Master's degree in early 1997. For her Master thesis, she did a 7-month project at the University of Southampton, UK, on the use of formal methods in the development of smart card operating systems. She is working as a software engineer for QC Technology, a Dutch company specialising in smart cards and the securing of data communication.

Eduard de Jong received the Master's degree in Computer Science in 1989 from the University of Amsterdam. He started his career in smart cards as the software architect in David Chaum's electronic purse project. In 1992, he co-founded QC Technology BV of Zaandam, Netherlands. In 1995, he co-founded Integrity Arts Inc. of San Mateo and joined the company as chief scientist in June 1996. He has applied for patents on a software architecture for secure transactions and electronic value transfer systems. His research interests are in high integrity software development and computer architecture for embedded systems.

Pieter H. Hartel received the Master's degree in Mathematics and Computer Science from the Free University of Amsterdam in 1978 and the Ph.D. degree in Computer Science from the University of Amsterdam in 1989. He has worked at CERN in Geneva and the University of Nijmegen and is currently a Senior Lecturer at the Universities of Southampton and Amsterdam. His current research interests are the theory of programming languages, programming language/compiler design, high integrity compilation/computer architecture and trusted systems design.