PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Reporte Técnico RT 08-06

A Certified Access Controller for JME-MIDP 2.0 enabled Mobile Devices

Ramin Roushani Oskui Gustavo Betarte Carlos Luna

2008

A Certi ed Access Controller for JME-MIDP 2.0 enabled Mobile Devices
Ramin Roushani Oskui; Betarte, Gustavo; Luna, Carlos
ISSN 0797-6410
Reporte Técnico RT 08-06
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República

Montevideo, Uruguay, 2008

A Certified Access Controller for JME-MIDP 2.0 enabled Mobile Devices

Ramin Roushani Oskui[1], Gustavo Betarte[2], y Carlos Luna[2]

[1]FCEIA, Universidad Nacional de Rosario, Argentina
[2]Instituto de Computación, Facultad de Ingeniería, Univ. de la República, Uruguay

Abstract. Mobile devices, like cell phones and PDAs, allow to store information and to establish connections with external entities. In this sort of devices it is important to guarantee confidentiality and integrity of the stored data as well as ensure service availability. The JME platform, a Java enabled technology, provides the MIDP standard that facilitates applications development and specifies a security model for the controlled access to sensitive resources of the device. This paper describes a high level formal specification of an access controller for JME-MIDP 2.0. This formal definition of the controller has been obtained as an extension of a specification, developed using the Calculus of Inductive Constructions and the proof assistant Coq, of the MIDP 2.0 security model. The paper also discusses the refinement of the specification into an executable model and describes the algorithm which has been proven to be a correct implementation of the specified access controller.

1 Introduction

Java Micro Edition (JME) [1] is a Java enabled platform targeted at resource-constrained devices. JME comprises two kinds of components: *configurations* and *profiles*. A configuration is composed of a virtual machine and a set of APIs that provide the basic functionality for a particular category of devices. Profiles further specify the target technology by defining a set of higher level APIs built on top of an underlying configuration. This two-level architecture enhances portability and enables developers to deliver applications that run on a range of devices with similar capabilities.

The Connected Limited Device Configuration (CLDC) [2] is a JME configuration designed for devices with slow processors, limited memory and intermittent connectivity. CLDC together with the Mobile Information Device Profile (MIDP) provides a complete JME runtime environment tailored for devices like mobile phones and personal data assistants. MIDP defines an application life cycle, a security model, and APIs that offer the functionality required by mobile applications, including networking, user interface, push activation and persistent local storage. Many mobile device manufacturers have adopted MIDP since the specification was made available. As millions of MIDP enabled devices are deployed worldwide and the market acceptance of the specification is expected to continue to grow steadily – the version 3.0 has recently been released, any circumvention of the application security model would have serious consequences.

In the first version of MIDP [3], any application not installed by the device manufacturer or a service provider runs in a sandbox that prohibits access to security sensitive APIs or functions of the device. Although this sandbox security model effectively prevents any rogue application from jeopardising the security of the device, it is excessively restrictive and does not allow many useful applications to be deployed after issuance of the device. Version 2.0 of MIDP [4] introduces a new

security model based on the concept of *protection domain*. Each sensitive API or function on the device may define permissions in order to prevent it from being used without authorisation. An installed MIDP application (MIDlet) suite is bound to a unique protection domain that defines a set of permissions granted either unconditionally or with explicit user authorisation.

In its desktop platform (JSE), the Java programming language embodies two basic security components: the *Security Manager (SM)* and the *Access Controller (AC)*. The SM is responsible for enforcing the security policies that have been declared to protect sensitive resources. It intercepts function calls to those resources and delivers permissions and access decision requests to the AC, which in turns performs the access decision procedures, determinig whether the caller has the needed rights to access the resource in question. In the case of the JSE platform, there exists a high level specification of the access controller. The basic mechamism used for the decision procedure is that of *Stack Inspection* [5]. No specification is provided of the access controller for JME.

This article reports work concerning the development of a (high level) formal specification of an access control module of JME - MIDP 2.0. In particular, a certified algorithm that satisfies the proposed specification of an AC is described. The specification of the module has been defined as a conservative extension of the model presented in [6,7], where a formal specification of the JME - MIDP 2.0 security model developed using the proof-assistant Coq [8] is presented and described in detail.

The rest of the paper is organized as follows. Section 2 briefly describes the MIDP 2.0 security model and the formal specification that has been developed. Section 3 presents the formal specification of the access control module. In Section 4 some security properties satisfied by this specification are stated and hints on how they were formally verified are provided. Section 5 presents the algorithm and discusses correctness results. Finally, Section 6 concludes and describes further work.

2 The JME - MIDP 2.0 Security Model

This section presents, in the first place, an informal description of the basic components of the MIDP 2.0 security model, to proceed then to briefly describe the formal specification of this model reported in [6,7]. Section 2.2 introduces notation that shall be used in the rest of the paper.

2.1 Informal Description of the Model

In MIDP, applications (MIDlets) are packaged and distributed as suites. A MIDlet suite can contain one or more MIDlets and is distributed as two files, an application descriptor file and an archive file that contains the actual classes and resources. A suite that needs access to protected APIs or functions must declaratively request the corresponding permissions in its descriptor. MIDlet suites may request permissions either as required or as optional.

Version 2.0 of MIDP [4] introduces a security model based on the concept of *protection domain*. A protection domain can be grasped as an abstraction of the execution context of an application, and it determines the access rights to the protected functions of the device. Each sensitive API or function on the device may define permissions in order to prevent it from being used without authorisation. A protection domain consists of both a set of permissions which are granted unconditionally, without intervention of the device's user, and a set of permissions which require authorisation from the user. Associated to each permission is defined a *mode* (*oneshot*, *session*, *blanket*), which establishes the form and frequency in which the autorization shall be required.

An installed MIDP application (MIDlet) suite is bound to a unique protection domain. Untrusted MIDlet suites are bound to a protection domain with permissions equivalent to those in a MIDP 1.0 sandbox. Trusted MIDlet suites may be identified by means of cryptographic signatures and bound to more permissive protection domains. This security model enables applications developed by trusted third parties to be downloaded and installed after issuance of the device without compromising its security.

The set of permissions effectively granted to a suite is determined from its protection domain, the permissions the suite request in its descriptor and the authorisations granted by the user.

For a more detailed description of the mechanisms defined by the security model the reader is referred to [1,2,3,4].

2.2 The Language Used

The specification has been developed using the Coq proof assistant [8], an implementation of the Calculus of Inductive Constructions (CIC) [9]. CIC is esentially an extension of Coquand and Huet's Calculus of Constructions [10] with inductive definitions based on Martin- Löf type theory.

The fragments of the specification that are presented in the rest of the paper should be immediately accesible to anyone used in classical set theory. This should work if only *types* and *sets*, as used in the text, are uniformly interpreted as sets of the classical theory. There remain however some (hopefully minor) mismatchings: it is used the symbol : instead of ϵ, this distinction can be regarded as totally immaterial. Record types are used, just as in programming languages, as tuple sets. A record definition has the form $R = \{field_1 : type_1, \ldots, field_n : type_n\}$. If r is an object of type R then the projection of $field_i$ is denoted $r.field_i$. A type called $Prop$ is used, of whom it is evident that must have *propositions* as objects. A *predicate* on a set A then is grasped as a propositional function whose type is $A \rightarrow Prop$. Logical operators shall be denoted as usual: ($\wedge$, $\vee$, $\sim$, $\rightarrow$, $\forall$, $\exists$).

2.3 Formalization of the MIDP 2.0 Security Model

A specification of the kind that shall be examined here is ultimately a set (*type*). It is satisfied (implemented) by any of its elements. The objects to be specified (midlets suites, state of the device, events) have a number of components related in certain ways. Their structure shall be described, giving specifications of the components and of the relations they must satisfy.

Let $Permission$ be the total set of permissions defined by every protected API or function on the device and $Domain$ the set of all protection domains. As a way of referring to individual MIDlet suites, the set $SuiteID$ of valid suite identifiers is also introduced.

An application descriptor is represented as a record composed of two predicates, *required* and *optional* that identify respectively the set of permissions declared as required and those declared as optional

$$Descriptor = \{required, optional : Permission \rightarrow Prop\} \tag{1}$$

A record type $Suite$ is defined to represent installed suites, with fields for its identifier, associated protection domain and descriptor,

$$Suite = \{sid : SuiteID, domain : Domain, descriptor : Descriptor\} \tag{2}$$

Permissions may be granted by the user to an active MIDlet suite in either of three modes, only once (oneshot), until it is terminated (session), or until it is uninstalled (blanket). Let $Mode$ be the

enumerated set of user interaction modes, $\{oneshot, session, blanket\}$, and $\leq_m$ an order relation such that $oneshot \leq_m session \leq_m blanket$.

The security policy on the device is represented as a constant *policy* of type

$$\begin{aligned} Policy = \{\ & allow : Domain \rightarrow Permission \rightarrow Prop, \\ & user\ \ : Domain \rightarrow Permission \rightarrow Mode \rightarrow Prop\ \} \end{aligned} \tag{3}$$

such that $allow\ d\ p$ holds whenever domain d grants unconditionally the permission p and $user\ d\ p\ m$ whenever domain d grants permission p with explicit user authorisation and maximum allowable mode m (w.r.t $\leq_m$). The permissions effectively granted to a MIDlet suite are the intersection of the permissions requested in its descriptor with the union of the allowed and user granted permissions.

To reason about the MIDP 2.0 security model, most details of the device state may be abstracted; it is sufficient to specify the set of installed suites, the permissions granted or revoked to them, and the currently active suite in case there is one. The active suite and the permissions granted or revoked to it for the session are grouped into a record structure

$$\begin{aligned} SessionInfo = \{\ & sid : SuiteID, \\ & granted, revoked : Permission \rightarrow Prop\ \} \end{aligned} \tag{4}$$

The abstract device state is described as a record

$$\begin{aligned} State = \{\ & suite : Suite \rightarrow Prop, \\ & session : option\ SessionInfo, \\ & granted,\ revoked : SuiteID \rightarrow Permission \rightarrow Prop\ \} \end{aligned} \tag{5}$$

Some conditions must hold for an state to be a valid one. For a suite to be installed on the device, for instance, the permissions requested in its descriptor must be a subset of the permissions defined by the protection domain bound to the suite. A compatibility relation, $\wr$, that makes it possible to specify this property is specified as follows

$$des \wr dom := \forall p : Permission, des.required\ p \rightarrow allow\ dom\ p \vee \exists m : Mode, user\ dom\ p\ m \tag{6}$$

therefore, the compatibility condition can be expressed with the following proposition

$$SuiteCompatible := \ \forall (s : State)(ms : Suite), s.suite\ ms \rightarrow ms.descriptor \wr ms.domain \tag{7}$$

For a detailed description of the validity conditions of a state the reader is referred to [6].

3 Formalization of the Access Control module

In this section we present the extensions done to the formal specification described in the previous section so as to model the behavior of the access control module.

First, the set $methodID$ of (method) identifiers that compose a midlet suite and the set $functionID$ of functions (or APIs) in a mobile device are inroduced. The definition of a *Suite* is extended with a predicate that characterizes the methods that belong to that suite

$$\begin{aligned} Suite = \{\ & sid : SuiteID,\ domain : Domain,\ descriptor : Descriptor, \\ & methodid : methodID \rightarrow Prop\ \} \end{aligned} \tag{8}$$

The state of the device is extended with the following predicates: one that specifies the functions or APIs registered in the device ($function$), one that describes the sensitive functions registered in the device ($functionSensible$), and one that models the association of a permission to a sensitive function ($funcperm$)

$$\begin{aligned} State = \{\ & suite : Suite \rightarrow Prop,\ session : option\ SessionInfo, \\ & granted, revoked : SuiteID \rightarrow Permission \rightarrow Prop, \\ & function,\ functionSensible : functionID \rightarrow Prop, \\ & funcperm : functionID \rightarrow Permission \rightarrow Prop\ \} \end{aligned} \tag{9}$$

A state now is valid whether it satisfies the conditions succinctly described in section 2.3 and the following ones (for details see [11]):

- $MIDletsAtLeastOneMethod$: every installed midlet has at least one method;
- $fAtLeastOnePerm$: every installed sensitive function necessarily has associated a permission;
- $fPermInstalledSens$: if a function has associated a permission then that function must be installed in the device and is sensitive;
- $fUniquePermission$: no function has associated more than one permission;
- $methodInOnlyOneMidlet$: all method identifier is unique, even in distinct midlets;
- $permStateCoherence$: if one permission was granted (revoked) to a midlet for the rest of its life cycle, then neither could have been revoked (granted), for the rest of its life cycle, nor could have been granted or revoked for the rest of the session. Likewise, if one permission was granted (revoked) for the rest of the session, then neither it could have been revoked (granted), for the rest of the session, nor could have been granted or revoked for the rest of its life cycle life; and,
- $policyCompatible$: if the security policy does not mention any association between the protection domain of a suite and a certain permission then that permission can not be registered as granted or revoked to that suite in the device.

The events related with the security are modeled in [6,7] as constructors of a type $Event$. Among these events we can mention, for example, those that correspond to the installation ($install$) and removal ($remove$) of a suite, and the permission request of the active suite active ($request$). The Event type is extended with a new constructor $call : methodID \rightarrow functionID \rightarrow Event$ that represents a method call to a function of the device. The behavior of the events is specified in [6,7] in terms of pre- and postconditions. The preconditions (Pre) are defined in terms of the device state (s), while postconditions (Pos) are defined in terms of the before (s) and after (s') states and an optional response of the device. In what follows a portion of the formal definition of the pre- and postconditions of the $call$ event is presented and explained.

Let $suite$ be a middlet suite, $meth$ be a method, $func$ be a function invoked by that method, and dom be the protection domain associated to $suite$. Then, the precondition of $call$

$$\begin{aligned} & \textbf{Pre}\ (call\ meth\ func) := PreCall\ meth \wedge \\ & (\ s.functionSensible\ func \rightarrow \\ & \exists\ perm : Permission,\ s.funcperm\ func\ perm \wedge PreRequestNone\ perm \wedge \\ & (\forall\ (ua : UserAnswer)\ (m : Mode), \\ & (\ ua = uaAllow\ m \rightarrow PreRequestUserAllow\ perm\ m\ ) \wedge \\ & (\ ua = uaDeny\ m \rightarrow PreRequestUserDeny\ perm\ ))) \end{aligned} \tag{10}$$

requires in the first place ($PreCall$) that it should exist an active session, that *suite* is the active suite and *meth* belongs to it. Then, if $func$ is a sensitive function, there must exist a permission *perm* associated to the function such that *perm* has been declared as required or optional inside the descriptor of *suite*. In the case that it is not required any user intervention, the precondition specifies ($PreRequestNone$) that the security policy either unconditionally allows *dom* to access $func$, or *dom* specifies a user interaction mode for *perm*, and the user has granted a permission which is still valid. In the case it is required user intervention and the user denies access, the precondition establishes ($PreRequestUserDeny$) that *dom* specifies a user interaction mode for *perm*, and the user has not granted any permission that is still valid. In the case where it is required user intervention and the user allows the access the precondition additionally requires ($PreRequestUserAllow$) the user authorization mode to be less, according to the order defined on permission modes, than the maximum allowed mode specified in the security policy.

The postcondition of the *call* event has been formally defined in Coq as follows

$$
\begin{aligned}
&\textbf{Pos}\ (call\ meth\ func) := s.functionSensible\ func \rightarrow \forall\ perm : Permission,\\
&(\ s.funcperm\ func\ perm \rightarrow PosRequestNone\ perm \vee\\
&(\ \exists\ ua : UserAnswer,\ \exists\ m : Mode,\\
&((\ ua = uaAllow\ m\ ) \wedge PosRequestUserAllow\ perm\ m\ ) \vee\\
&((\ ua = uaDeny\ m\ ) \wedge PosRequestUserDeny\ perm\ m\ ))) \wedge\\
&\sim s.functionSensible\ func \rightarrow s = s' \wedge r = Some\ allowed
\end{aligned}
\tag{11}
$$

The postcondition in the case it is not required any user intervention ($PosRequestNone$) essentially specifies the response of the access controller when a *call* event is executed. For example, if the user has previously allowed (denied) authorization and it is still valid then the module will respond allowing (denying) the access to the function. If the security policy specifies that the access should be allowed unconditionally, then the module will respond allowing permission. In the case that is required user intervention and the user grants the permission, the specification establishes ($PosRequestUserAllow$) that the response of the controller should be to allow the access. Similarly, if the permission is granted at session level (*session* mode) or for the whole life cycle of the application (*blanket* mode), this authorization should be registered in the device. If it is granted in *oneshot* mode, then the entire state of the device remains unchanged and the controller just reacts granting the access. $PosRequestUserDeny$ is similar to $PosRequestUserAllow$ but the controller shall deny the access and the update will be on the fields that register the access denial (depending on whether the denial is for the rest of the session or for the rest of the life cycle of the suite).

The complete formalization is available in [11].

4 Verification of Security Properties

Several security properties concerning the behavior of the access controller have been proved to be satisfied by the access control module model. Additionally, it has been verified that the extension is conservative, in particular the invariants satisfied by the core model, and described in [6,7], are preserved in the extended model, including those regarding the validity of the state of the device.

The proof, for instance, that the execution of any event preserves the compatibility property of installed midlet suites described in section 2.3, follows by first proving that the *call* event does not modify the state of the device and then proceeding as shown in [7].

Due to space limitations, in what follows only two of the properties described in section 3 are formally stated and hints are provided on how it was proved they are satisfied by a *call* event execution. For the complete set of analyzed properties, including their proofs, the interested reader is referred to [11].

Lemma permStateCoherence:

$$
\begin{aligned}
&\forall (s : State)(sid : SuiteID)(p : Permission)(ses : SessionInfo),\\
&s.session = Some\ ses \rightarrow ses.sid = sid \rightarrow\\
&(s.granted\ sid\ p \rightarrow \sim s.revoked\ sid\ p \wedge \sim ses.granted\ p \wedge \sim ses.revoked\ p) \wedge\\
&(s.revoked\ sid\ p \rightarrow \sim s.granted\ sid\ p \wedge \sim ses.granted\ p \wedge \sim ses.revoked\ p) \wedge\\
&(ses.granted\ p \rightarrow \sim s.granted\ sid\ p \wedge \sim s.revoked\ sid\ p \wedge \sim ses.revoked\ p) \wedge\\
&(ses.revoked\ p \rightarrow \sim s.granted\ sid\ p \wedge \sim s.revoked\ sid\ p \wedge \sim ses.granted\ p)
\end{aligned}
$$

Proof: Each of the four subproperties (propositions in the conjunction formula) is proved following a similar strategy. In the case that no user intervention is required, either because is established by the security policy or because the invoked function is not sensitive, the proof follows directly because the state of the device remains unchanged. In the case that user intervention is required, the proof proceeds by performing case analysis on the response provided by the user to the permission request (allow or deny in oneshot, session and blanket modes).

Lemma policyCompatible:

$$
\begin{aligned}
&\forall (s : State)(sid : SuiteID)(p : Permission)(ses : SessionInfo),\\
&s.session = Some\ ses \rightarrow ses.sid = sid \rightarrow\\
&\forall\ ms : Suite,\ s.suite\ ms \rightarrow ms.sid = sid \rightarrow\\
&(\sim policy.allow\ ms.domain\ p \wedge \sim \exists\ m : Mode,\ policy.user\ ms.domain\ p\ m) \rightarrow\\
&(\sim s.granted\ s\ sid\ p \wedge \sim s.revoked\ s\ sid\ p \wedge \sim ses.granted\ p \wedge \sim ses.revoked\ p)
\end{aligned}
$$

Proof: In the case that user intervention is not required, the proof follows directly because the state of the device remains unchanged. Otherwise, the proof follows by absurdity, because a request for user intervention would contradict the hypothesis that there is no relation in the security policy between the permission and the protection domain of the active suite.

5 A Certified Access Controller

The high level formalization that has been described in the previous sections is appropriate and general: it is a good setting to reason about the properties of the MIDP 2.0 security model and, in particular, about the behavior of the access control module without conditioning possible implementations. Nevertheless, in order to construct and certify an algorithm that satisfies the access control behavior that has been specified, first the high level model has been refined into a more computationally oriented one and then it has been proved the soundness of the refinement process. Then, the algorithm was defined using the functional language of Coq and a theorem that establishes that the obtained algorithm satisfies the concrete specification of the access controller has been formally proved with the help of the proof assistant.

5.1 Model Refinement

The refinement of the high level specification consists in providing a representation for both the state of the device and the events that can be directly implementable using a high level functional language. In addition to that, the relation which specifies the behavior of the events should be refined down to a deterministic function that computes explicitly the state transformation specified by those events.

In the high level formalizations that have previously been described, the elements that inhabit big inductive types are those whose types contain elements of type $Prop$, namely the propositional functions. For the refinement of a decidable predicate P over a finite set A, it has been adopted the following approach: if the intended meaning of P is to characterize a set, then it shall be represented using lists of elements of type A, as in programming languages. On the other side, if the intended meaning is that of a decision procedure, its representation shall be a function F from A to a type isomorphic to the type $bool$. Formally, a function F is said to refine a predicate P if it holds that $\forall\, a, P\ a \iff F\ a = true$. It should be noticed that the refinement of a predicate defined on a certain set into a boolean function over that set is computationally safe if the predicate is decidable and in addition its domain of definition is finite. This is the case for the predicates involved in the formal models discussed here. The use of this refinement approach as a means to obtain a concrete model that represents properly the abstract one can be found in [11]. A general treatment of model refinements is discussed in [12].

To illustrate how the components of the concrete model looks like, it follows the definition of the set $CState$, the concrete representation of the state of the device. Notice that concrete components of the specification are prefixed by a (capital or small) c

$$\begin{aligned} CState = \{\ & csuite : list\ CSuite, \\ & csession : option\ CSessionInfo, \\ & cfunction,\ cfunctionSensible : functionID \to bool, \\ & cfuncperm : functionID \to Permission, \\ & cgranted,\ crevoked : SuiteID \to Permission \to bool\ \} \end{aligned} \tag{12}$$

5.2 The Algorithm

Before proceeding to describe the algorithm, it is worth remarking that in the specified model each method inherits or gets the protection domain that has been associated to the midlet suite to which the methods belong. Therefore, every method of a middlet suite is related to the permissions that the protection domain embodies.

The algorithm has been built as a quite direct implementation of the specified behavior of the access controller in the concrete model: basically, when a method invokes a protected function or API the access controller checks that the method has the corresponding permission and that the response from the user, if required, permits the access. If this checkup is successful, the access id granted; otherwise, the action is denied and a security exception is launched. The complete definition of the algorithm in Coq can be found in [11]; here it is only included a pseudo-code version. It can be noticed that its structure is that of a case expression where the guards of the branches express properties involving some of the following components: the state, the policy of the device, the method and the invoked function

$$
\begin{aligned}
&\textbf{cAccessCtrlExe}\ (cs : CState)\ (cpolicy : CPolicy)\ (ua : UserAnswer)(m : methodID)\ (f : functionID) : \\
&\qquad\qquad CState\ \times\ (option\ Response) := \\
&case\ (methInCurrSuite\ cs\ m) = false :\ (cs, None) \\
&case\ (cs.cfunctionSensible\ f = false) :\ (cs, Some\ allowed) \\
&case\ (ms.cdescriptor.crequired\ p \vee ms.cdescriptor.coptional\ p) = false :\ (cs, Some\ denied) \\
&case\ cs.cgranted\ cs.csession.csid\ p = true :\ (cs, Some\ allowed) \\
&case\ cs.crevoked\ cs.csession.csid\ p = true :\ (cs, Some\ denied) \\
&case\ cs.csession.cgranted\ p = true :\ (cs, Some\ allowed) \\
&case\ cs.csession.crevoked\ p = true :\ (cs, Some\ denied) \\
&case\ cpolicy.callow\ ms.cdomain\ p = true :\ (cs, Some\ allowed) \\
&case\ (cpolicy.cuser\ ms.cdomain\ p\ oneshot \vee cpolicy.cuser\ ms.cdomain\ p\ session \vee \\
&\qquad cpolicy.cuser\ ms.cdomain\ p\ blanket) = true : \\
&\{ \\
&case\ ua = uaAllow\ oneshot :\ (cs, Some\ allowed) \\
&case\ ua = uaAllow\ session :\ (cs\ except\ cs.csession.cgranted\ p = true, Some\ allowed) \\
&case\ ua = uaAllow\ blanket :\ (cs\ except\ cs.cgranted\ (cs.csession.csid)\ p = true, Some\ allowed) \\
&case\ ua = uaDeny\ oneshot :\ (cs, Some\ denied) \\
&case\ ua = uaDeny\ session :\ (cs\ except\ cs.csession.crevoked\ p = true, Some\ denied) \\
&case\ ua = uaDeny\ blanket :\ (cs\ except\ cs.revoked\ (cs.csession.csid)\ p = true, Some\ denied)) \\
&\} \\
&case\ (cpolicy.cuser\ ms.cdomain\ p\ oneshot \vee cpolicy.cuser\ ms.cdomain\ p\ session \vee \\
&\qquad cpolicy.cuser\ ms.cdomain\ p\ blanket) = false :\ (cs, Some\ denied)
\end{aligned}
\tag{13}
$$

5.3 The Certification

The certification of the algorithm that implements a correct access controller reduces to prove the (correctness) theorem that establishes the following property: if the precondition of the *call* event is satisfied by a given state, then the state resulting from executing the algorithm satisfies its corresponding postcondition. Formally this is stated as follows:

Theorem Correctness:

$$
\begin{aligned}
&\forall\ (cs\ cs' : CState)(cpolicy : CPolicy)(ua : UserAnswer) \\
&(m : methodID)\ (f : functionID)\ (r : option\ Response), \\
&CPre\ cs\ cpolicy\ (ccallFunc\ m\ f) \rightarrow \\
&cAccessCtrlExe\ cs\ cpolicy\ (ccallFunc\ m\ f)\ ua = (cs', r) \rightarrow \\
&CPos\ cs\ cs'\ cpolicy\ r\ (ccallFunc\ m\ f).
\end{aligned}
$$

Proof: The proof follows directly the structure of the algorithm and is constructed performing case analysis on the conditions of the branches and makes use of several properties of the model, in particular those related to the validity of the state of the device.

6 Conclusion and Further Work

This article reports work concerning the formal specification of the MIDP 2.0 security architecture. A high level formalization, using the CIC and the proof assistant Coq, of the JME-MIDP 2.0 Access Control model has been developed which extends the model reported in [6] so as to consider, for instance, the event that represents the invocation of the device functions by an application. This extension is shown to be conservative with respect to the security properties satified by the core model. Additional relevant security properties satisfied by the specification of the access control module are also stated and discussed. With the objective of obtaining a certified executable algorithm of the access controller, the high level specification has been refined into a computationally oriented equivalent one, and an algorithm has been constructed that is proved to satisfy that latter specification. To our knowledge, this is an unprecedented result. For a complete and detailed description of this work the reader is referred to [11].

The formal specification that has been developed assumes that the security policy of the device is static, that there exists at most one active suite in every state of the device, and that all the methods of a suite share the same protection domain. The obtained model, however, can be easily extended so as to consider multiple active suites as well as to specify a finer relation allowing to express that a method is bound to a protection domain, and then that two different methods of the same suite may be bound to different protection domains. Further work, already in progress, is the study and specification of different sort of permission models to control the access to sensitive resources of the device, and the definition of the corresponding algorithms for enforcing the security policies derived form those models.

References

1. Java Platform Micro Edition, `http://java.sun.com/javame/index.jsp`, last access: May 2008.
2. JSR 139 Expert Group: CLDC. Version 1.1. Sun Microsystems, Inc. and Motorola, Inc. (2006)
3. JSR 37 Expert Group: Mobile Information Device Profile for Java Micro Edition. Version 1.0. Sun Microsystems, Inc. (2000)
4. JSR 118 Expert Group: Mobile Information Device Profile for Java Micro Edition. Version 2.0. Sun Microsystems, Inc. and Motorola, Inc. (2002)
5. Wallach, D., Felten, E.: Understanding Java Stack Inspection. In Proceedings of the 1998 IEEE Symposium on Security and Privacy. Oakland, CA (1998)
6. Zanella Béguelin, S.: Especificación formal del modelo de seguridad de MIDP 2.0 en el Cálculo de Construcciones Inductivas. Master's thesis, UNR Argentina (2006)
7. Zanella Béguelin, S., Betarte, G., Luna, C.: A Formal Specification of the MIDP 2.0 Security Model. In: T. Dimitrakos et al. (eds.) FAST 2006. LNCS, vol. 4691, pp. 220-234. Springer, Heidelberg (2007)
8. The Coq Development Team: The Coq Proof Assistant Reference Manual Version V8.1 (2006)
9. Coquand, T., Paulin-Mohring, C.: Inductively Defined Types. In Per Martin-Löf and Grigori Mints, editors, Conference on Computer Logic, volume 417 of LNCS, pages 50-66. Springer-Verlag (1988)
10. Coquand, T., Huet, G.: The Calculus of Constructions. Information and Computation, 76(2/3):95-120, February/March (1988)
11. Roushani Oskui, R.: Especificación Formal en Coq del Módulo de Control de Acceso de MIDP 2.0 para Dispositivos Móviles Interactivos. Master's thesis, UNR Argentina (2008). `http://nursystem.com/ramin.roushani`, last access: May 2008.
12. Spivey, J.M.: The Z Notation: A Reference Manual. International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK (1989)