Project N^o: FP6-015905

Project Acronym: MOBIUS

Project Title: Mobility, Ubiquity and Security

Instrument: Integrated Project

Priority 2: Information Society Technologies

Future and Emerging Technologies

Deliverable D5.1

Selection of case studies

Due date of deliverable: 2007-03-31 (T0+18)

Actual submission date:

Start date of the project: 1 September 2005 Duration: 48 months

Organisation name of lead contractor for this deliverable: FT

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contributions

Site	Contributed to Chapter
FT	1, 2, 4, A
SAP	3,4
TL	2, 4
TLS	4

Executive Summary:
Selection of Case studies

This document describes the result of the work within Task 5.1 of the MOBIUS project. The objective of the task is to prepare the evaluation of MOBIUS tools developed in Work Packages 2,3 and 4 on the classes of properties defined in Work Package 1. This evaluation will be carried out during the subsequent tasks of Work Package 5.

Two application fields will be considered:

- checking midlets, small Java applications downloadable on mobile phones, to ensure that they are, at least, innocuous with respect to a generic phone security policy and, when applicable, that they are safe with respect to their intended usage.
- checking security properties of applications based on SAP NetWeaver and more particularly incorrect data validation of user input leading to code injection or cross-site scripting vulnerabilities.

This deliverable summarizes what must be taken into account to specialize verification tools to model accurately the runtime environment used by the applications of the two application fields.

There are different dimensions in the complexity of the case studies:

- Properties range from very simple checks on the value of some arguments that can easily be checked automatically to complex temporal and information flow properties that require human assistance to guide the proof.
- Application complexity range from very simple mostly independent sequential code fragments to full-fledged multi-threaded applications using different kinds of object-oriented programming patterns.
- Evaluation criteria will also include the level of automation achieved.

The proof carrying code infrastructure will be mostly evaluated in the context of the verification of midlets. Its evaluation will rely on the generation of proof certificates for properties already used as cases studies for logic-based and type-based tools.

Contents

1 Introduction 6
1.1 Classification of case studies 6
1.2 Applicative fields considered 7
1.3 Classification of properties 7
1.4 Overview of the expected capabilities of the tools 7
1.4.1 Automated type-based tools (WP2) 7
1.4.2 Logic-based tools (WP3) 8
1.4.3 Proof carrying code (WP4) 8
1.5 Structure of the report 8

2 Sample MIDP applications 9
2.1 Introduction 9
2.2 Generic technical constraints for the evaluation of MIDP applications 9
2.2.1 Modeling the MIDP runtime environment 9
2.2.2 Availability of source code 10
2.2.3 Availability of real examples with real security risks 10
2.2.4 Modification to the application framework 11
2.3 Modelling MIDP 12
2.3.1 Introduction to MIDP specifications 12
2.3.2 Levels of modelling 13
2.4 MIDP programming patterns 16
2.4.1 SUN Microsystem, message sender pattern 16
2.4.2 High level concurrency library 16
2.4.3 Sample applications conforming to a simple navigation graph 17
2.4.4 Implementation of multiple permissions 18
2.4.5 High level concurrency library 20
2.5 Real applications with source code 20
2.5.1 A tiny telnet client: Ktelnet 20
2.5.2 A web feed reader: RSSReader 20
2.5.3 A personal safe : Pocket protector 21
2.5.4 Momental Joker 23
2.6 Real applications without source code 24
2.6.1 Small/medium size applications from MIDlet.org 24
2.6.2 Map viewers 25
2.6.3 Other major industrial midlets 26

3 SAP NetWeaver 27
3.1 Introduction 27
3.2 SAP Portal 27
3.3 Enterprise SOA 30
3.4 Malicious Code Injection 35

4 Evaluation of verification tools 36
4.1 Automatic type-based tools 36
4.1.1 Metrics and measures of success for case studies 36
4.1.2 Resource policy verification 37
4.1.3 Information flow control 40
4.2 Logic based tools evaluation 42
4.2.1 Metrics and measures of success for case studies 43
4.2.2 Verification of resource bounds 43
4.2.3 Information flow 43
4.2.4 Verification of navigation graph 44
4.2.5 Combination of properties: Momental Joker 44
4.2.6 Verification of application specific properties of applications based on SAP NetWeaver 45
4.3 The proof carrying code framework 47
4.3.1 Introduction 47
4.3.2 Logic-based proof carrying code and Certified navigation graphs . . . 47
4.3.3 Lightweight proof carrying code 48

5 Conclusion 49
5.1 Coverage of requirements expressed in Work Package 1 49
5.2 Intended use of the case studies 49
5.3 The next steps 50

A Interface and security behaviour outline for JavaME MIDP applications 53
A.1 Rationale 53
A.2 Definition and semantics of navigation graphs 54
A.2.1 Traces and executions of a midlet 54
A.2.2 Definition of navigation graphs 56
A.2.3 Enhanced counting of resources 56
A.2.4 Extensions to threads 57
A.2.5 Information flows 58
A.3 Representation of navigation graphs 59
A.3.1 Graphical representation of navigation graph 59
A.3.2 Examples of diagrams 59

Chapter 1

Introduction

This report will present the different case studies that will be used throughout the evaluation Work Package to check the ability of the tools developed in the project to fulfill the security objectives that have been described in the Requirements Work Package (presented in [22, 23]).

In this short chapter we will present the main classification criteria that can be used to organize case studies and we will deduce the structure of the report.

1.1 Classification of case studies

Several criteria should be taken into account to classify case studies :

- the application:
  - its application field,
  - availability of the code : verification on source or on compiled bytecode;
- the security properties:
  - their objective (resource property, information flow, temporal property, etc.),
  - their complexity (property with an atomic objective, compound property mixing several objectives),
  - their genericity (application versus framework property).
- the verification tools used:
  - accessible to automatic tools or requiring manual reasoning,
  - proof performed on bytecode or source code,
  - use of the PCC paradigm:
    - simple verification without certificate generation,
    - generation of a certificate for an independent checker,
    - on-device checking of the certificate.

Applicative fields will provide the structure for the next chapters where applications are described. The project and Work Package structure will provide the organization of the chapter enumerating individual case studies.

1.2 Applicative fields considered

The scope of the MOBIUS project is the security architecture of the next generation of global computers and more specifically small mobile embedded terminals.

This is a very broad application field and we have to restrict it for the purpose of the evaluation done within the project. As the goal of the project is the security architecture and not the definition of application environments or security policies, we have decided to restrict evaluation of the technologies developed within MOBIUS to mature application fields where the execution environments already follow the principles envisioned for the future global computer infrastructure, and where well defined security policies are already available.

In Work Package 1, we had already chosen two main application areas:

- downloadable applications for mobile phones and more specifically the MIDP framework which is the most standard platform already available on more than 1.2 billion phones;
- an infrastructure for WEB services developed by SAP.

1.3 Classification of properties

Work Package 1 deliverables define and classify a set of security properties that should be met by applications belonging to one of the application fields considered.

Resource control properties This category contains all the properties related to counting. It can either be memory size, instructions executed, or specific events;

Information flow properties Those properties check where critical information is used. The goal is usually to check that the value returned by a method considered as producing private information (a datasource) is not transferred as argument to some other method that can publish this information outside the midlet (a datasink);

Framework properties these are generic properties that do not depend on the intended use of the application. They may either fall in the previous categories or express simple temporal constraints; Bounding the range of potential values of the arguments of dangerous methods is an important sub-category.

Application specific properties These are properties specific to an application. They can mix different kinds of simpler properties and express more complex temporal relations.

1.4 Overview of the expected capabilities of the tools

1.4.1 Automated type-based tools (WP2)

We will distinguish proof of concept tools developed for specific properties by individual groups from the final demonstrator (mainly developed by Trusted Labs).

The latter will only integrate techniques that have proved effective and reliable for industrial use. It will be able to treat complete Java programs but may be less accurate than some of the experimental prototypes.

1.4.2 Logic-based tools (WP3)

The integrated tool will allow reasoning both at source and byte code level. It will generate verification conditions for the core logic of the project (an extension of JML). It will interface with several provers (ESC/Simplify, Bicolano/COQ or Jack) and translate the MOBIUS core logic to the provers logic. There are also plans to integrate type-based automatic tools in the global architecture.

As for type-based tools, the level of maturity may vary between plugins and so they may not be able to treat the same kind of applications. Nevertheless we will provide the common infrastructure (this means mainly modelling the MIDP or NetWeaver APIs) so that they may treat any of the proposed application.

1.4.3 Proof carrying code (WP4)

This is the most advanced and experimental part of the project and it has the longest chain of technological dependencies, so it may be limited to the simplest applications.

It will heavily rely on the generation of certificates by the proof environment. Another path of research is the generation of certificates corresponding to the certified static analyser working on the Bicolano representation of programs. The certificate is then a minimal amount of information that is necessary to be able to check the result of the analysis in limited time on a on device checker.

1.5 Structure of the report

The report is mainly structured by application fields.

The next chapter presents the MIDP framework under the angle of its integration in proof environments and several midlets we plan to use as case studies. This chapter heavily relies on the notion of navigation graph to describe the behaviour of midlets.

Chapter 3 describes the SAP NetWeaver portal that will be used as an alternate application field and source of case studies.

Chapter 4 distributes the different properties associated to the different case studies between the proof tools and try to asses our coverage of both the categories of properties as defined in Work Package 1 and the capabilities of the tools as defined by the different Work Packages. Its structure also follows the structure of the different task of Work Package 5.

As an appendix we present an experimental formalism where extended automata, called navigation graphs, are used to describe midlet behaviours and especially security properties that midlets should respect. It provides a way to express in a systematic way application specific properties or parameterized framework properties.

Chapter 2

Sample MIDP applications

2.1 Introduction

First we present some of the constraints faced for providing a convincing set of case studies based on the MIDP framework. Then we explain how we will model the implementation of a MIDP compliant runtime environment used by the applications tested. Finally we will describe different kinds of applications ranging from small examples to industrial grade midlets.

We present the main characteristics of the different MIDP applications that will be used as case studies during the evaluation.

First we characterize the part of the MIDP framework necessary to execute the application: the configuration of the machine and the companion APIs used. We give a high overview of their behaviour using the navigation graph formalism introduced in appendix A. We also sketch the challenge they represent for verification tools.

2.2 Generic technical constraints for the evaluation of MIDP applications

2.2.1 Modeling the MIDP runtime environment

The behaviour of a midlet is defined both by the behaviour of the application code and the behaviour of the runtime environment that contains the MIDP libraries.

Specifying the behaviour of the runtime environment is a major issue because of the following constraints:

- the implementation of a runtime environment is much larger and much more involved than the typical application we are trying to check. Moreover significant parts are written in the native language of the platform.
- each implementation is only one instantiation of the Java standards. It may contain many spurious assumptions not implied by the standard itself. Furthermore, we are not interested in the abstract security properties of a midlet running on an ideal runtime but in the actual behaviour with the potentially bugged runtime available on the customer phone.

Nevertheless, we will try to abstract away from the runtime bugs issue and separate correctness of the application itself and the compliance of the phone runtime with the Java specifications. Still, in section 2.3.2, we show that getting a valid model is not easy.

2.2.2 Availability of source code

The goal of the evaluation in the MIDP application field is to check the ability of the tools to prove properties about any application. So we are looking for midlets with interesting security properties to enforce not developed within the consortium.

Unfortunately, most midlets are provided only as bytecode. MOBIUS aims at proving properties at the level of bytecode because in the current industrial practice, it is not reasonable to ask for the source code of evaluated midlets.

But for the sake of the evaluation, it is often much easier to work with source code because we want to be able to check by hand the proofs inferred by the tools and we want to do this on a readable source code rather than on the bytecode.

We also want to be able to modify a midlet that satisfies the security requirements to make dangerous variants and show that those variants are not validated by our tools.

Several kinds of applications with different usage will be proposed:

- some simple applications with full source code that can be used as basis for modified versions;
- a few more complex applications with source code (rare) and with very strong security requirements that can be used for checking application specific properties with logical based tools;
- a huge set of bytecode-only applications that can be used to check how well actual bytecode is supported by the tools and as a regression suite between different versions of the tools;
- a small set of well-understood bytecode-only applications that exhibit well known specific behaviours that we want to be able to check.

2.2.3 Availability of real examples with real security risks

Assessing MOBIUS tools as capable of finding a malicious applications means that such application are available as benchmarks. Fortunately, there are not a lot of applications breaking what can be considered as a sound policy.

The MIDP security model ensures that dangerous actions will not take place unless the user has explicitly granted that right to the application or the application has been signed by an operator. But operators will not sign an application if they have no control over its code.

So the MIDP dynamic security policy is effective at protecting malware against end user but it also prevents completely safe applications using too many critical APIS from running in an effective way : the number of security screens will deter potential end-users from using them.

So we will need to write some “malware application” that could prove to be dangerous if they were signed. It is then important to show that tools would not sign such applications but also that perfectly safe applications using the same APIs will be recognized as such and

```
controller() {          synchronized() {
   ...                     while(true) {
   set P to true              if (!P) obj.wait();
   obj.notify();              process()
   ...                        reset P
}                          }
                        }
```

Figure 2.1: Using notify to synchronize a worker thread

that the number of false positive raised by the tool is low. We will use existing applications available with source codes to write variants with an unwanted behaviour.

2.2.4 Modification to the application framework

Experimenting innovative security policies

Innovative security policies such as the proposal for grouping permissions requests (presented in [23] and in section 2.4.4) require changes to the implementation of the VM or at least its core libraries. Even changing a core Java library is not easy because they are "romized" ie. included in the data image of the process with the executable.

We can do those changes on available emulators using the recently open sourced Sun reference implementation but not on a real device as complete open implementation for actual handsets are not available or extremely complex to use: it is also theoretically feasible to change the Java execution environment on a phone with an open operating system (Symbian, Linux, Windows) but this is beyond the scope of the MOBIUS project.

Raising the level of the concurrency library

Synchronization between concurrent threads in MIDP is done through primitive mechanisms operating directly on shared memory. The memory model of Java is very complex making safe reasoning over memory invariants difficult.

Practical use of the model for synchronization is the following. We informally distinguish controlling and controlled processes. The basic principle is to use a computable predicate P over the memory state that can be changed by controlling processes. The status is polled by controlled processes who will act accordingly when the truth value of the predicate is changed. Usually controlled processes will reset the value to false. Most of the time this mechanism is used to tell another process that a data is ready to be used. Access to shared memory must be synchronized to ensure that what is read is valid.

To avoid active polling, Java provides a notification mechanism. A process waiting for an event may stop itself by calling wait() on an object. It will be waken up by another process calling notify() on the same object. notify is called when the process has changed the value of P on the current state to true. But in practice, notify can only be used as an indicator to wake up stopped processes (that have called wait). If the process is not already stopped when it checks the predicate P, there is always a shortcut that does not execute the wait statement. This also means that calling notify is not mandatory and some other processes

could trigger a synchronization that would change the value of P but would not call notify. The synchronization could then not be isolated as a pair of notify/wait constructs.

This is acknowledged by the Java standardization community and last version of JavaSE provides new high-level synchronization primitives (JSR 166 [14]). that should replace low level mechanisms in most programs [9]. Such a library should and probably will be adapted for the MIDP framework (in fact most of it is not SE specific) and we think we should anticipate this evolution to show how it could simplify verification.

2.3 Modelling MIDP

2.3.1 Introduction to MIDP specifications

MIDP is the profile for JAVA mobile phone applications developed on top of the JavaME CLDC configuration. A configuration defines the subset of the Java language supported by the execution platform and its core library. A profile defines the set of specialized library that can be used by the applications to access the resources of the platform.

Security properties are mainly expressed as properties of the applications on the use of those APIs. In order to specialize analysis tool to MIDP applications, it is necessary to model the behaviour of MIDP APIs and to specify properties on critical MIDP APIS.

Components of the MIDP specifications

MIDP as all the Java technologies is not defined by a single component but it is rather a set of related components building a framework. Different kinds of documents are needed:

- The initial virtual machine specification and the Java language specification define the bytecode language and the standard variant of Java. They have been developed by SUN.
- The CLDC specification presents a variant of the language and the virtual machine suitable for small devices with a set of core service libraries. It has been defined by SUN within the Java Community Process (JCP) which is the Java standardization body.
- The MIDP library defines a configuration, i.e. a framework suitable for a given application domain. MIDP is mostly used on mobile phones but its application field was not limited to this kind of devices. MIDP has been defined by Motorola within the JCP. The reference implementation has been developed by SUN.
- Various libraries are needed to perform specific operations such as sending a SMS (WMA), playing music (MMAPI), etc. These specifications have been defined within the JCP with various group leader (Nokia, BenQ/Siemens, Motorola, etc.).
- Finally umbrella JSR defines how to use those elementary specification to build a system for a given target field. JTWI and MSA are two such specification. They also specify minimum resource requirements for platforms implementing those specifications.

Access to MIDP specifications

MIDP is a proprietary technology defined by an industrial consortium lead by SUN Microsystems: the Java Community Process. Specifications defined within the JCP are made up of three components:

- the normative text itself, usually organized as a Javadoc document and refered as the JSR,
- the reference implementation (RI) is an implementation defined by the group leader,
- the TCK (Technology compatibility kit) is a test sequence that all conforming implementation should pass. It is also developed by the Spec group leader.

The leader of the specification group is the owner of the intellectual property developed within the group.

There are several implementations of the MIDP runtime. Compliance to each part of the specification is defined only by succeeding or not to the TCK suites. Unfortunately these test suites unavailable to end-users have proven to be very weak at limiting misinterpretation of the original specification.

Another view of compliance is to use the reference implementation as THE implementation.

A last view which is both more complex but more accurate is to loosely specify the behaviour of the runtime environment relying only on the text definition. Such a formal specification will probably be non-deterministic reflecting the weaknesses of the original document.

Such a formalization work is beyond the scope of MOBIUS. We will have to rely on actual code coming either from reference implementations or stubs extracted or derived from real implementations.

Availability of reference implementations

For the sake of the MOBIUS project, the most important recent news are:

- SUN has released all its reference implementations under the GNU Public Licence and among them CLDC [18], MIDP [11], WMA [12, 17], MMAPI [13], Web Services [15], SATSA [16]. But it has not released its TCK. They are available at https://phoneme.dev.java.net/.
- Motorola has released the TCK for Bluetooth [?] but not the reference implementation. It will release MIDP 3 RI and TCK with an open source licence when they will be available. The web site is http://opensource.motorola.com.

With those announcements, it is possible to develop formal models of the runtime environment based on actual reference implementations of the components of the runtime. It is also a strong incentive for limiting the scope of the project to MIDP libraries covered by those announcements.

2.3.2 Levels of modelling

Java Virtual Machine requirements

Before modelling the profile, it is necessary to ensure that the configuration requested is correctly modeled.

CLDC versus JavaSE MIDP programs and CLDC programs in general run on the KVM virtual machine a light variant of the conventional JVM used for the Standard Edition of the language. CLDC restrict both opcodes and basic support libraries.

- CDLC specification imposes the absence of JSR/RET opcode in bytecode (this requirements is also imposed on the last version of the standard edition).
- CLDC version 1.0 restricts number types to integers as there was no floating point hardware on ARM7 processors used in mobile phones. If we restrict ourselves to CLDC1.0 compliant midlets, we do not need floating point operations.
- Longs are supported but not much used except for system time.

CLDC restricts the use of native code extensions. There is no JNI interface and an application cannot dynamically extend the functionality of the application. On the other hand, there is a limited mechanism known as KNI and used by the profile that provides a way to access functionalities that cannot be described in the Java bytecode language (such as the implementation of the user interface).

At least some of those primitives must be formalized in some way to describe the behaviour of a midlet, but it is often easier to provide directly a high level formalization of the profile APIs rather than to formalize the low level primitives and then synthesize a model of the high level APIs from their implementation.

Bicolano current status Bicolano is the formalism used by the MOBIUS project to represent bytecode in the verification tools based on COQ. It only treats a fragment of the Java bytecodes. Most restrictions imposed on the KVM correspond exactly to Bicolano shortcomings.

On the other hand, Bicolano does not support:

- characters and string operations,
- multi-threading (but work on extending Bicolano with multi-threadin is underway in WP3.3),
- object and class manipulation (forName and instantiate).

String operations must be supported to make any reasonable benchmark as strings are used to represent key informations for security (URL, rights, passwords, etc.).

A lot of interesting examples rely on the use of multiple threads. Moreover the implementation of the user interface of a midlet is inherently multi-threaded. Nevertheless some interesting examples do not require the formalisation of multi-threaded executions.

Direct object manipulation (Class.forName and Class.newInstance) is usually used to address fragmentation issues (to check the availability of some components on the phone and access a feature only if it is available). There is no real need for these operations in midlets that target a single phone.

Basic stubs for MIDP libraries

For purely control flow based analysis, it is possible to develop opaque versions of the libraries. The simplest one is the following:

1. most APIs are just treated as empty stubs;
2. APIs registering callbacks effectively register the methods so that they appear in the callgraph of the application;
3. scheduler functions simulate arbitrary calls to registered method. They code an upper approximation of the potential callgraphs of the application;
4. arguments given to the callback function are just placeholders.

(Re)implementing the behaviour of MIDP classes

A first refinement to basic stub is to add code to limit behaviours of the stub to only valid execution traces of the schedulers. The solution is to add a minimal state to the implementation of the profile library that can be used by the schedulers to select the callbacks that can be called at a given point of the execution:

1. APIs unregistering a callback effectively unregister the methods from potential callbacks;
2. scheduler functions effectively implement the real state automaton and limit the list of potentially activable callbacks at any time to what a compliant implementation would do;
3. arguments to callbacks functions transmitted by schedulers effectively cover the range of potential arguments.

A good example is how the GUI is modelled. Whereas in the first version, we only register all the CommandListener callbacks, in the second version we must maintain which commandListener is associated to the current screen. Moreover the command and the screen object given as argument to the callback are determined by the current state of the application. On the other hand, the implementation should do perfect random choices on potential external events such as which key is pressed, so that the choice function does not bias the analysis.

(Re)implementing the information flow of MIDP classes

But it is necessary to refine this even more if we want to perform information flow analysis on applications as profile APIs can be used as channels to convey information between different parts of the application:

- At least any field that can be set and retrieved through a public API should be modelled,
- Many other channels exist. Some can be expressed through control (which callback is called, with which arguments).

JML specification of a subset of the MIDP profile

Another approach is to formalize the behaviour of MIDP classes as JML pre/post conditions using ghost variables to implement the state of their instances. This approach will be followed for the core of the MIDP specification. This work may have interesting side effects such as a clarification of the specification itself.

2.4 MIDP programming patterns

This section presents classical MIDP programming pattern that are frequently used and that have an impact on the analysis of the secure behaviour of midlets. For example, using a worker thread to perform blocking tasks such as communication is a recommended practice. We have also tried to capture the different ways of programming a user interface and keeping track of the internal state using MIDP lcdui high-level widget library.

We also present some new patterns using dedicated libraries that could enhance the perceived security (use of synthetic multiple permissions) or replace too low level mechanisms that are too difficult to analyze but also to use correctly for the average programmer (JSR 166 concurrency library).

This list is not exhaustive and may be completed during the evaluation itself as we will discover new programming practices.

2.4.1 SUN Microsystem, message sender pattern

This is a small midlet provided as a programming example with variant by SUN with its release of the Wireless toolkit (in the folder apps/NetworkDemo). The midlet shows how to use the “Worker thread” pattern to implement the sending of short messages (typically datagrams or SMS) over the network without disrupting the use of the GUI of the application. The principle was presented in a technical note for developers [19].

The main challenge is understanding the use of shared variables for transferring message text and notify signals for signaling between the GUI thread and the thread that calls the message sending primitive. This is important in the prospect of bounding the number of message sent as the worker thread is coded as an unbounded loop.

Variations over this midlet are also good candidates for information flow analysis as the worker thread may (should as a good programming practice) be reused by different parts of a program needing to send information over the network. It is important to distinguish the different use to understand the actual information flow.

We will use a graphical representation of midlets that we call navigation graphs. The navigation graph represents the potential screens of the application linked by edges showing the events that can trigger a transition between those screens and the critical actions (from a security point of view) that may occur during those transitions. They are formally defined in appendix A and figure A.1 can be used as a legend for reading them.

Figure 2.2 presents the navigation graph of this application. One can see that the use of threads make it very complex although one can also question the need for a separate thread that establishes the connection (SUN architectural choice).

2.4.2 High level concurrency library

It is unlikely that we can handle counting resources in practice in the presence of threads because the synchronization between threads is done at a too low level (directly on shared memory using arbitrary predicates on the memory state as synchronization locks).

A way to address this issue is to develop a high level concurrency library along the lines drawn by JSR 166 for Java SE.

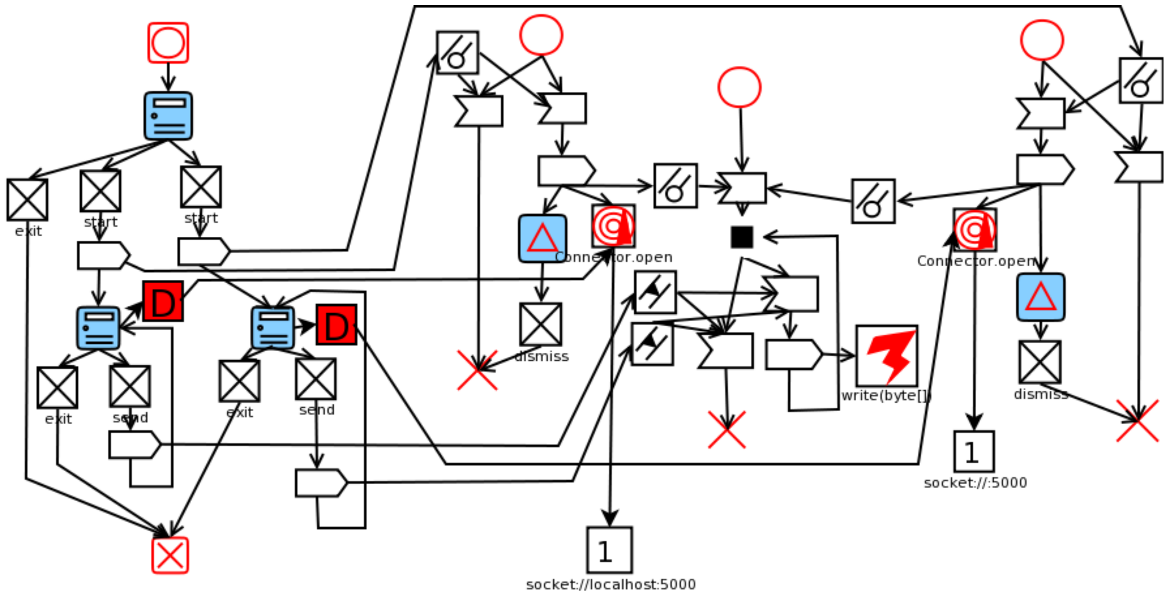

Figure 2.2: Socket pattern navigation graph

2.4.3 Sample applications conforming to a simple navigation graph

We propose to use a single navigation graph and very simple applications respecting this graph but using different implementation of the control flow.

Figure 2.3 presents such a diagram with a structure sufficiently complex but recurrent patterns (there is only two commands available on each screen, command reuse means that it cannot be used to guess the current node). There is also no symmetry so that nodes can be explicitly distinguished.

We will study different implementation strategy to develop an application compliant with this graph:

- The implementation coincide explicitely with the layout of the graph with one object screen per node of the navigation graph, a different command listener per object and

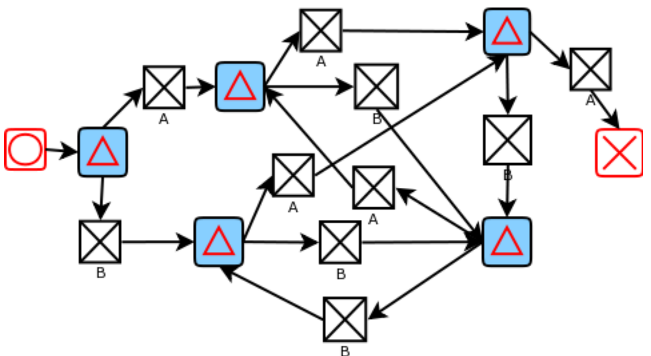

Figure 2.3: Simple navigation graph

simple tests on the command used to select the next screen to display.

- Some nodes are merged and represented in the implementation by the same Screen object. They differ through some other properties of the screen object.
- Some nodes are merged in a single implemented Screen and differ through some property of the global MIDlet state.
- The implementation uses one single command listener and relies on the value of the screen supplied as first argument to track the current node of the navigation graph.
- The single command listener used in the implementation relies on some MIDlet internal state variable to represent the current node of the navigation graph.
- The implementation uses more structured screens (Form) with Item object and ItemCommandListener objects. In that case the command is changed to be an event of the item listener.
- A critical action can be added to the graph to check that their handling as part of the path is supported:
  - the action is explicitely on the path between the begining of the callback and the instruction that changes the current screen,
  - the action is in the callback but is performed after the screen has been changed,
  - the action is in another thread.
- We will also try to generate midlets together with their navigation graph description, for example using Netbeans mobility pack. The goal is to show the potential of tying proof generation to an IDE. For example this graph can be described in netbeans and the skeleton of the midlet can be extracted from it.

2.4.4 Implementation of multiple permissions

It is necessary to find an effective way to limit the number of security screens seen by end-users to make the security policy more effective.

A solution is to implement a way to separate granting a permission and consuming that permission in the MIDP security framework. It can then be used to group several permission requests together. The main problem is then checking the correspondence between the number of permissions granted and the number of uses.

We will implement several frameworks with multiple permissions on top of the GPL version of Java and to evaluate for each version how easy it is to check their correct use.

Figure 2.4 presents an intuitive example of such an API use for obtaining the permission to send two SMS Single line comments name three important actions: ¡R¿ request for permission and ¡C1¿,¡C2¿ permission consumption.

We will also propose several tests midlets representing expected uses of the API based on the simple scenario given above. Figures 2.5 2.6 2.7 and 2.8 present usage scenarios as extended navigation graphs with increasing complexity. Blue rounded rectangles represent screens displayed on the handset. Arrows connecting them represent transitions between screens triggered by pressing a button. They are labelled with important actions. We

```
/* We create two permissions for sending two SMS. At this
   point they are not validated and are unusable */
Permission p1 = new Permission(”sms://01234”);
...
Permission p2 = new Permission(”sms://56789”);
...
/* We create a PermissionRequest object that will accumulate
PermissionRequest request = new PermissionRequest();
request.add(p1);
request.add(p2);
request.grant(); // ¡R¿
/* After this point we can use p1 and p2 */
...
msg1 = sms˙connection.newMessage(...,”sms://01234”);
msg1.setPayloadText(...);
sms˙connection.send(msg1,p1); // ¡C1¿
...
msg2 = sms˙connection.newMessage(...,”sms://56789”);
msg1.setPayloadText(...);
sms˙connection.send(msg2,p2); // ¡C2¿
```

Figure 2.4: Sample code for two permissions requested together and consumed separately

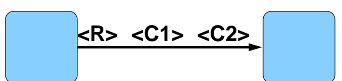

Figure 2.5: Two permissions requested and consumed in one visible user transition

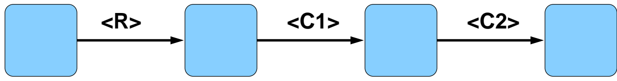

Figure 2.6: Two permissions requested and consumed in distinct user transition

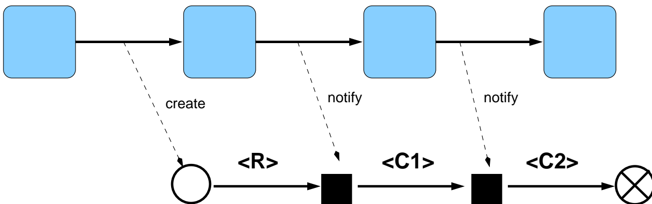

Figure 2.7: Two permissions requested and consumed in distinct user transition. Actions done sequentially in a simple thread

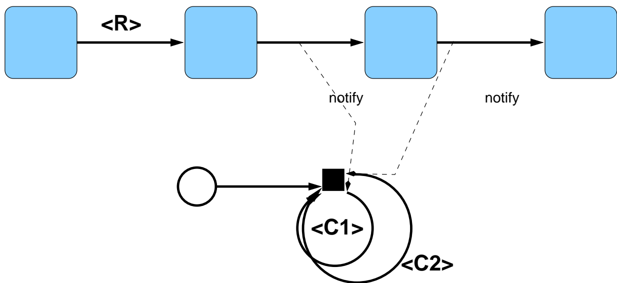

Figure 2.8: Two permissions requested and consumed in distinct user transition. Blocking actions done in a worker thread

will also use threads. Important states are represented by circles. Dashed arrows indicate synchronization actions like thread creation or condition notification to blocked threads.

2.4.5 High level concurrency library

As explained in section 2.2.4, analysing programs that use explicit channels to synchronize is much easier than analysing the usage of shared memory for synchronization.

We propose to rewrite some simple concurrent midlets that could prove difficult to check with MOBIUS tools, so that they use a simple equivalent of JSR 166 constructs.

2.5 Real applications with source code

2.5.1 A tiny telnet client: Ktelnet

Ktelnet is a very simple midlet providing a rudimentary telnet interface with a server. The code is contained in 270 lines of Java. The only point to check is the origin of the address used in the connection and the fact that data are transmitted to the destination. Its navigation graph is presented in figure 2.9.

Security objectives The application must connect only to the address specified by the user not to a potentially malicious proxy or a second location.

2.5.2 A web feed reader: RSSReader

This is a more complex application with a more complex user interface. The RSS reader interface offers to add new feeds to the internal state of the system or to read one of them. A feed is described by the address where the contents can be retrieved.

A feed is read offline. The contents is cached on the phone and is retrieved with the update command. The update is done with a background thread as usual.

Regarding security, the main difference with the socket applications are :

- the complexity of the user interface if one wants a view of the global behaviour of the application,
- the origin of the address it connects to,
- the more complex security objective.

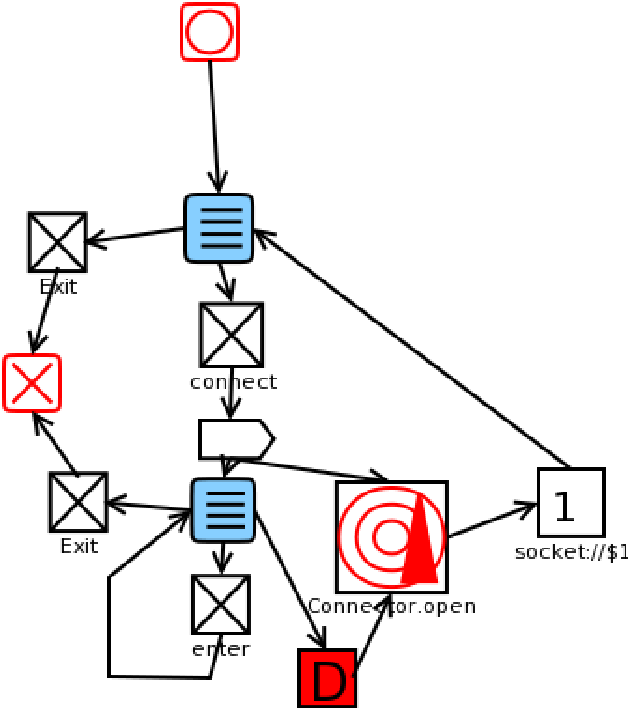

Figure 2.9: KTelnet navigation graph

Its navigation graph is detailed in figure 2.10.

Security objectives An RSS reader must keep feeds separate and also it knows all the names of the threads read by the customer, it should never send information to an information source about the other sources (for example names and last items retrieved).

This is an application property that is hard to capture in a generic setting as it introduces a notion of "feed context".

2.5.3 A personal safe : Pocket protector

This is a CLDC 1.0 midlet that uses the record store to store sensitive information using an open source lightweight cryptographic library called BouncyCastle for performing the cryptographic operations.

Security objectives

The goal of the pocket protector is to protect the data stored in it. The main risk is that the passphrase is captured by an unauthorized actor either because it has been transmitted or because it is stored on the phone and the phone has been stolen. We will not address risks such as the strength of encryption.

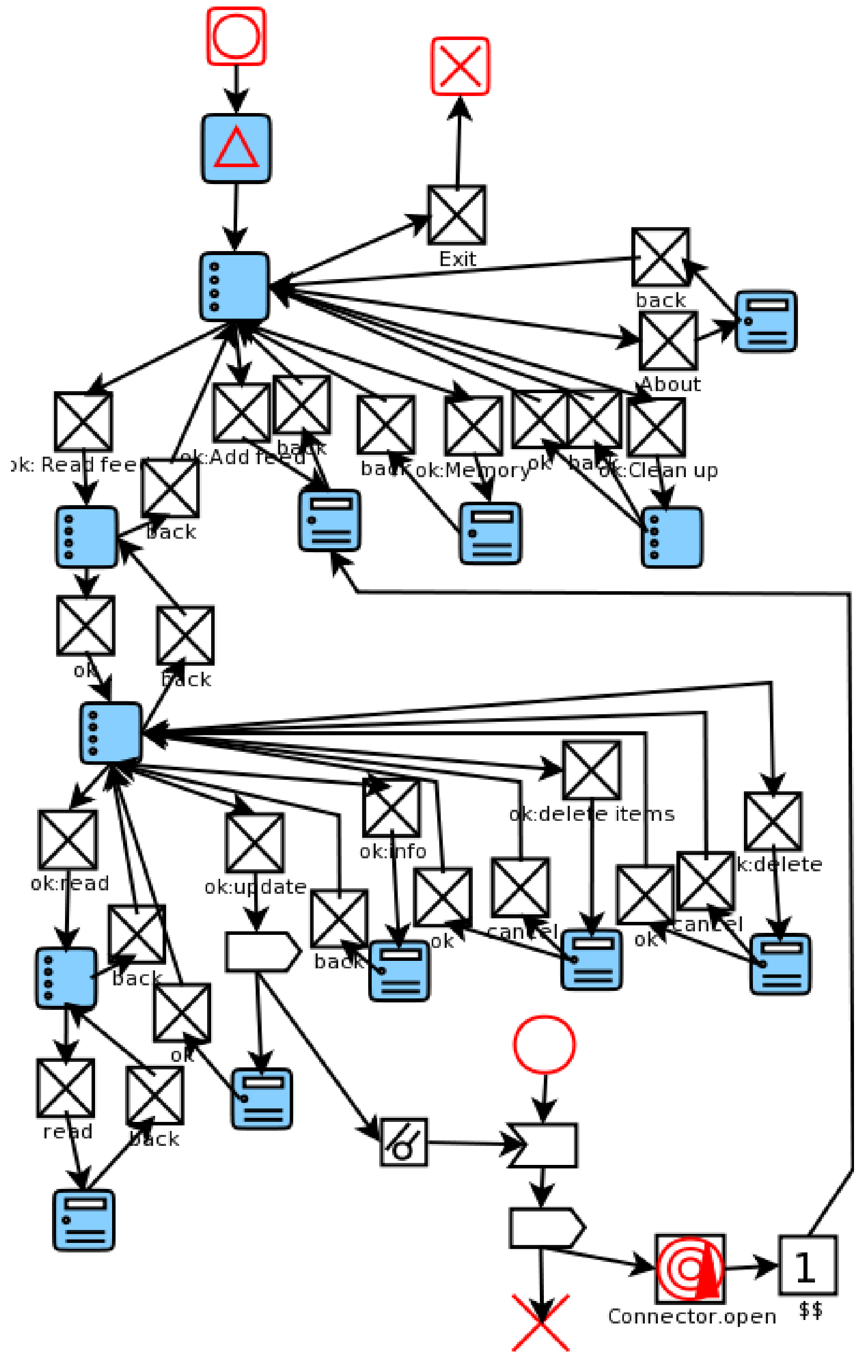

Figure 2.10: RSS reader navigation graph

Navigation graph

The GUI and the handling of the data are performed by two different packages. Figure 2.11 presents the navigation graph without the handling of the store. It is a behavioral graph and does not take into account the creation of a thread for the slow task of opening the record store (in fact it does not make the graphical user interface more responsive as the screen displayed is blocked).

The implementation is tricky as far as inferring the graph is concerned:

- The PocketAlert forms are in fact implemented by a single object used in different contexts with different commands (the ok command is only accessible when it is used after an Unlock screen).
- The sequence of screen from Unlock to Decrypt including StoreOpen is determined by the state of an internal variable not by the previous screen.
- A thread is used to display those screens.
- Some screens are only accessed on errors so when an exception is raised.

Use of information

The passphrase is used to decrypt the contents of the record store. It is kept in memory (in a variable) as long as the store is opened. The passphrase is retrieved from the user with a textfield in the unlock screen. It cannot be changed and is set the first time when no store exists.

Fields stored on the phone are encrypted with the passphrase given and the complete record should be encrypted.

2.5.4 Momental Joker

This is an application developped by a student group at Chalmers, under the supervision of A. Sabelfeld and A. Askarov. It has been developped in JIF [26] a variant of Java with an enhanced type system tracking information flows.

It implements mental poker, a distributed poker game without a trusted third party but where cheating can nonetheless be detected. The basic idea is that both players shuffle and encrypt cards. To reveal a card one needs the collaboration of both players. Commutativity properties between encryption and permutation makes partial release of information possible during the game.

The assets to protect are the player's hand, the keys used to encrypt cards not yet revealed and for signing actions and the permuation of cards.

Security objectives The main objective is to ensure that the player can safely use its application to play poker, whatever is the implementation of his opponent. The main risk is that information on his hand is leaked out from his phone.

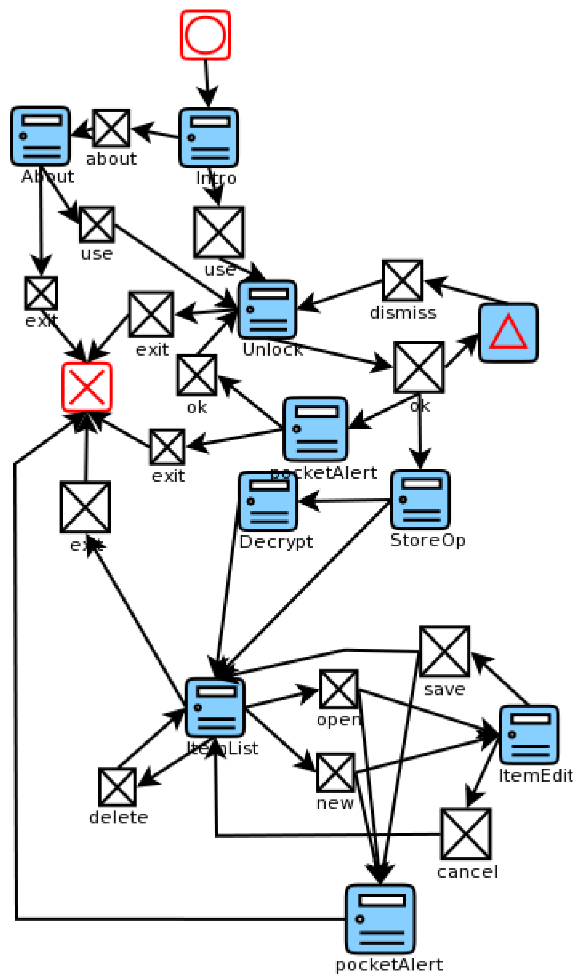

Figure 2.11: Pocket Protector navigation graph

2.6 Real applications without source code

Those applications are much more complex than the previous ones and rely on several external libraries. Being able to treat them is a proof of industrial maturity of the tools which is not the immediate goal of the project.

2.6.1 Small/medium size applications from MIDlet.org

First we propose to use an archive of several hundred simple midlets as a “regression test suite” for tools. They can be used to prove several facts about analysis tools developed:

- Tools can handle bytecode of real life examples.
- Tools can spot simple properties on those examples although we cannot prove that the analysis is exhaustive.
- As such, it is a benchmark that can be used to compare the precision of the results of

the analysis performed with different tools as those tools are not complete even if they are sound.

Security objectives We will check some framework specific properties on those applications : control on the use of system properties, which connections are established, how many times items are sent over the network, etc.

2.6.2 Map viewers

We propose to use three widespread midlets for viewing maps as main examples as the implementation of some of them contain interesting features that one may want to track. Furthermore the degree of complexity of the analysis required is variable.

Google Local

Google Local is an interface to the Google Maps service for mobile phones. This midlet is one of the first available map viewers for mobiles with industrial strength quality. It is a pure MIDP midlet (MIDP2.0 and CLDC1.0). It uses a single URL whose value is in the JAD file to establish a connection with the map service but it can also open a phone connection when the user wants to contact a service identified in the map.

Microsoft LiveSearch

This is the main competitor to Google Local and it provides the same kind of services.

Like J2MEMap (discussed below) it also sends information on the customer phone (but less than MobileZoo) to a distant site, but this time the information is buried in the HTTP POST request. Although the information transferred is less critical than the analysis done by the next application J2MEMap, this is a bad practice that could be spotted by static analysis if the content of the request sent can be analysed (still this is a straightforward, non obfuscated action).

Security objectives we must prove that System.getProperty results are sent back to the server.

J2MEMap

This is a viewer for Google Maps developed by Thomas Landspurg (CTO of In Fusio a major mobile game provider). It provides additional services such as :

- interfaces with an internal localization service or an external bluetooth connected GPS.
- Interface to other information layer such as Wikimapia,
- Saving kml files and other information on the flicker web service,
- Use of JSR 75 to save files,
- Sending “recommendation” SMS to friends...

Several versions exist depending on the capabilities of the phone. The most complete version (for Sony Ericsson K750 phones) requires the following JSR: CLDC 1.1, MIDP2.0, JSR 120 (Wireless messaging), JSR 135 (Mobile media), JSR 082 (Bluetooth), JSR 179 (Location), JSR 075 (PDA optional packages), JSR 172 (Web Services).

In addition to the actual service, this midlet includes a kind of spyware library known as mobilezoo that checks the capabilities of the user phone and sends a report to a distant server where those data are accumulated to build a database on handset characteristics for developers (mobilezoo.biz). The library checks the availability of several APIs with Class.forName and several phone properties with System.getProperty. The names of the items to check are retrieved (for some of them) from the network (note: the contents exchanged with servers is crypted enough to make it hard for an analyser to retrieve the actual value, but the information flow should be visible).

The spy library is a separate package and is freely available to developers who would like to use it. As a consequence the spying code and the connection to the distant site are separate from the core of the application. It is just called once from the regular midlet. It is hidden to most end-users, only because most phones do not give the name of the site they connect to when an HTTP connection is opened.

Security objectives

spyware detection we must be able to detect the connection to several services and how information is exchanged between those different sources. We must show that the value of the arguments for Class.forName and System.getProperty ultimately depend on data received on a connection with an external server and that the results of System.getProperty are sent back to this server.

collocation the use of the location API must be controlled. Only the map retrieval should use customer coordinates. As there are several sources of coordinates, both must be controlled.

SMS check how SMS are used (the origin of the destination address used is user input and contents must not include any personal information).

2.6.3 Other major industrial midlets

GMail This is an interface to the GMail portal. It may open phone calls.

Opera-mini web browser This is a web browser. It is a MIDP 2.0, CLDC 1.0 application that also uses WMA and MMAPI. Among the non intended behaviour, it also checks a lot of properties like the IMEI and can create capture multimedia players (audio and video).

YahooGo and AOL mobile applications They are similar to the commercial midlets presented so far. The only reason to consider them is that they may use different kinds of connection and have different information flow (For example, do they retrieve phone information as J2MEMap ?).

Chapter 3

SAP NetWeaver

3.1 Introduction

SAP NetWeaver is a web-based, open integration and application platform that serves as the foundation for enterprise service-oriented architecture (enterprise SOA).

SAP added a Java stack to the SAP NetWeaver platform in 2004. The Java stack in SAP NetWeaver 2004s is fully J2EE 1.3-compliant. SAP is also an early adopter of Sun‘s newest Java specification, Java EE 5 (Java Platform, Enterprise Edition 5) and is using it in its Java EE 5-compliant SAP NetWeaver application server.

NetWeaver provides a web-based access to business applications, through the NetWeaver Portal. It is also possible to access the underlying components and data via web services and develop custom applications based on these.

Both the web-based system (Portal) and the web services infrastructure (enterprise SOA) could be vulnerable to Cross-site scripting (XSS) and SQL injection attacks [21]. In section 4.2.6 we will discuss these vulnerabilities more in detail, and elaborate on how the ideas presented in section 5.1.2 Temporal requirements on method calls of Deliverable 1.2, would allows us to verify these systems.

In the following sections we will give a detailed description of both the Portal and Enterprise SOA systems, and the interaction between them, at the hand of a practical example which is part of a Defense PLM (Plant Maintenance) application. The example will describe how maintenance emails can be sent through the Portal, and at the same time are integrated with Microsoft Outlook, through a web service. This can be seen as a sample of the jointly developed Duet Software of SAP and Microsoft.

3.2 SAP Portal

For the development of applications on the Portal, SAP provides the SAP Portal Development Kit for Java (PDK for Java). It is a set of Java APIs and Web Dynpro components. Web Dynpro is SAP’s model-driven approach to building Web-based user interfaces. The PDK for Java enables developers to create Web Dynpro applications that access services, and to integrate the Web Dynpro applications into the Portal. The PDK also enables the creation of portal services that run on top of the Portal Runtime engine.

The Model-View-Controller (MVC) paradigm [28] is used, to create Portal applications. Users need elements within the user interface to interact with the application. The Web

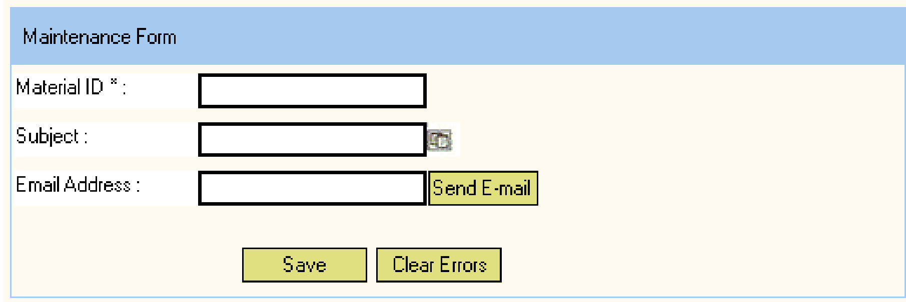

Figure 3.1: Maintenance form for PLM

Dynpro concept allows you to split the user interface into an arrangement of views, so-called iViews.

As an example we have a simple iView with three input fields for Material ID, Subject, and E-Mail Address. The field Material ID is marked as mandatory. Figure 3.2 demonstrates this.

Using the Send E-Mail pushbutton, the user can navigate to a second view in which an e-mail message can be edited and sent. This navigation link should only be followed up by the Web Dynpro runtime environment if the input field for the e-mail address contains an entry and the person in question is actually responsible for the maintenance of the material. We will not go into details for the Send E-Mail iView. Entering a Material ID is required. This is marked by an asterisk at the label in front of the material field. If the input check is successful after you have pressed the Save button, the system issues an appropriate message. By pressing the button Clear Errors, the user can then return to the initial status of the application example, if error messages to be processed are displayed. In this case, the input check is suppressed.

The implementation of the view controller is divided up into the following procedures:

- First of all, additional methods are declared for the input check and for the initialization of value attributes.
- The input check for the input fields Material ID, Subject, and E-Mail Address is started in the event handler for the Save action using the appropriate method calls.
- The user should be able to reset all input fields by selecting Clear Errors, even if they contain incorrect entries
- The navigation to the EMailEditor should only be possible when the user has entered a valid e-mail address.

The IWDMessageManager interface is used in different controller methods to display messages. The methods checkDesired(), and checkMandatory() are called by the event handler of the onActionSave() action, as shown in Code Fragment 1. They contain the actual input checks. The relevant error message is stored internally. By calling the report methods of the IWDMessageManager interface, it is possible to use the temporary storage of the Web Dynpro runtime environment for these messages.

```
//@@begin imports
import com.sap.tc.webdynpro.progmodel.api.IWDMessageManager;
import com.sap.tc.webdynpro.errorbehavior.wdp.IMessageSimpleErrors;
import com.sap.tc.webdynpro.errorbehavior.wdp.IPrivateForm;
//@@end
...
/** declared method */
public void checkDesired(java.lang.String fieldName) {
  //@@begin checkDesired()
  IWDMessageManager messageMgr =
    wdComponentAPI.getMessageManager();
  Object attributeValue =
    wdContext.currentContextElement().getAttributeValue(fieldName);
  IWDAttributeInfo attributeInfo =
    wdContext.getNodeInfo().getAttribute(fieldName);
  if (attributeValue instanceof String) {
    if (((String) attributeValue).length() == 0) {
      if (fieldName.equals(IPrivateForm.IContextElement.E_MAIL_ADDRESS))
        messageMgr.reportContextAttributeMessage(
          wdContext.currentContextElement(),
          attributeInfo,
          IMessageSimpleErrors.DESIRED_E_MAIL,
          null,
          true);
    }
  }
  //@@end
}
/** declared method */
public void checkMandatory(java.lang.String fieldName) {
  //@@begin checkMandatory()
  IWDMessageManager messageMgr =
    wdComponentAPI.getMessageManager();
  Object attributeValue =
    wdContext.currentContextElement().getAttributeValue(fieldName);
  IWDAttributeInfo attributeInfo =
    wdContext.getNodeInfo().getAttribute(fieldName);
  if (attributeValue instanceof String) {
    if (((String) attributeValue).length() == 0) {
      String fieldLabel =
          wdContext.getNodeInfo().getAttribute(fieldName)
            .getSimpleType().getFieldLabel();
      messageMgr.reportContextAttributeMessage(
        wdContext.currentContextElement(),
        attributeInfo,
        IMessageSimpleErrors.MISSING_INPUT,
        new Object[] { fieldLabel },
        true);
    }
  }
  //@@end
}
}
...
```

Figure 3.2: Code Fragment 1

```
...
/** Hook method called to initialize controller. */
public void wdDoInit() {
  //@@begin wdDoInit()
  this.initialize();
  //@@end
}
...
/** declared method */
public void initialize() {
  //@@begin initialize()
  wdContext.currentContextElement().setMaterialID("");
  wdContext.currentContextElement().setSubject("");
  wdContext.currentContextElement().setEMailAddress("");

wdContext.getNodeInfo().getAttribute(IPrivateForm.IContextElement.E_MAIL_
ADDRESS)
           .getModifiableSimpleType().setFieldLabel("E-Mail Address");
  //@@end
}
...
```

Figure 3.3: Code Fragment 2

The declared initialize() method is used to initialize the value attributes in the controller context and the data type information field label that belongs to the EMailAddress value attribute. The method is called from the wdDoInit() method and from the non-validating action event handler onActionClear() directly after instantiating the view controller, this is illustrated in Code Fragment 2.

Now we show the implementation of the method onActionSave in Code Fragment 3. Navigation to the input of an e-mail message in the EMailEditor view is only started in the Web Dynpro runtime environment if no error is reported in the checkMandatory()method.

In the last step, the event handler is implemented for the non-validating Clear action. Such an event handler is called before individual user entries are validated and stored in the controller context. This allows you to initialize the context again despite incorrect user input. See Code Fragment 4.

3.3 Enterprise SOA

After business functions have been implemented, a Web service interface, which is visible to the Web service consumer, has to be created. This interface provides an abstraction layer and consequently independence from the specific implementation. Based on this interface, the Web service is configured and can be accessed during runtime. Full UDDI [7] client capabilities can be used to publish the Web Services to a UDDI registry: Web service definitions and deployed Web services can be stored in a UDDI registry. WSDL documents [6] provide the basis for the Web service client and can be found in the UDDI using a browser or the standard UDDI API's. In the SAP Web AS, UDDI client and server functions are provided. You can search in all, and publish to all, registries that conform to the standard. A UDDI server is part of the SAP Web AS, so you can create your own registries. SAP also provides a public UDDI Business Registry under uddi.sap.com

Open integration can only be achieved if it is based on generally accepted standards.

```
...
/** declared validating event handler */
public void
onActionSave(com.sap.tc.webdynpro.progmodel.api.IWDCustomEvent
wdEvent )
{
  //@@begin onActionSave(ServerEvent)
  this.checkMandatory(IPrivateForm.IContextElement.MATERIALID);
  this.checkDesired(IPrivateForm.IContextElement.E_MAIL_ADDRESS);
  wdComponentAPI.getMessageManager().raisePendingException();
  wdComponentAPI.getMessageManager().reportSuccess(
    "The sample form data was successfully saved!");
  //@@end
}
...
```

Figure 3.4: Code Fragment 3

```
...
//@@begin javadoc:onActionClear(ServerEvent)
/** declared non validating event handler */
//@@end
public void
onActionClear(com.sap.tc.webdynpro.progmodel.api.IWDCustomEv
ent
                              wdEvent )
{
  //@@begin onActionClear(ServerEvent)
  // Re-initialize this view´s context
  this.initialize();
  //@@end
}
...
```

Figure 3.5: Code Fragment 4

```
//@@begin javadoc:wdDoInit()
/** Hook method called to initialize controller. */
//@@end
public void wdDoInit() {
  //@@begin wdDoInit()
  // create a new instance of the Web Service ModelClass
  Request_SendEmailPortType_sendEmail req =
    new Request_SendEmailPortType_sendEmail();

  // bind new instance of the Web Service ModelClass to the
  // independent Model Node 'WebServiceEmail'
  wdContext.nodeWebServiceEmail().bind(req);
  //@@end
}
```

Figure 3.6: Code Fragment 5

The SAP Web Application Server implements the basic Web services standards eXtensible Markup Language (XML), SOAP, Web Service Definition Language (WSDL) and Universal Description, Discovery, and Integration (UDDI). The following basic standards driven by the Java and Web service community are included as well:

- WS Interoperability: WS-I BP 1.0 [2]
- Java API: JAXP 1.2, JAXM 1.1, SAAJ 1.1, JAX-RPC 1.0 [8]
- XML: SOAP 1.1 w/ Attachments, WSDL 1.1, UDDI v2, Web Services Inspection Language [6]
- Security: WS-Security (1.0), SSL / TLS [27]

Web Dynpro tools provide special modeling tools for defining a Web Dynpro model. With Web Services as a possible model type, Web Dynpro offers the possibility for connecting a Web Dynpro front end to an existing back end. The developer can point the model wizard to an arbitrary Web Service on the Internet and have the code for accessing this Web Service generated automatically.

Each Web Dynpro component has a corresponding Component Controller. In order to create a connection between the Component Controller and the model, you will bind the context of the Component Controller to the created model structure.

In every controller there are predefined locations where you can insert the source code. The standard methods wdDoInit() and wdDoExit() belong to these locations. wdDoInit() is always controlled if the controller is instanced. For this reason, at this point there is an object which - at runtime - represents the entire input, including all the input parameters for calling the Web Service business method. This is presented in Code Fragment 5. In addition, you will need a further controller method in which the actual call of the Web service method is triggered. So the actual Web-Service call for sending the email message is implemented in the public method sendEmail() in the component controller, as shown in Code Fragment 6.

To understand the two important code lines

```
//@@begin javadoc:onActionSendMail(ServerEvent)
/** Declared validating event handler. */
//@@end
public void sendEmail() {
  //@@begin onActionSendEmail(ServerEvent)
  IWDMessageManager msgMgr = wdComponentAPI.getMessageManager();

  try {
    // call Email Web Service and update dependent model node 'Response'
    wdContext.currentWebServiceEmailElement().modelObject().execute();
    wdContext.nodeResponse().invalidate();

    BigInteger result = wdContext.currentResponseElement().getResult();
    String msg = "Email Web Service returned " + result.toString();

    if (result.intValue() == 0) {
      msgMgr.reportSuccess("Your email was successfully sent ("
        + msg + ")!");
    } else {
      msgMgr.reportWarning("Your email was not successfully sent ("
        + msg + ")!");
    }
  } catch(Exception ex) {
    msgMgr.reportException(ex.getLocalizedMessage(),true);
  }
  //@@end
}
```

Figure 3.7: Code Fragment 6

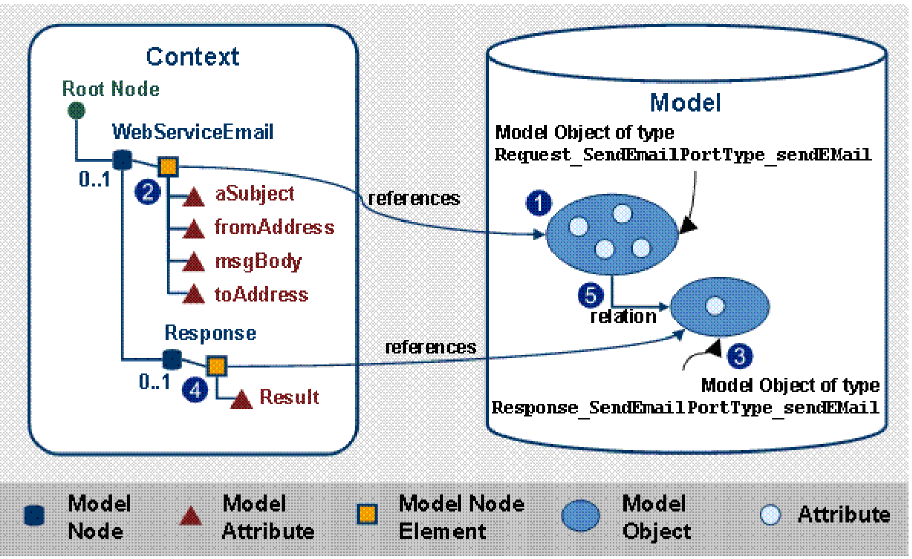

Figure 3.8: Web Services Model in Web Dynpro

wdContext.currentWebServiceEmailElement().modelObject().execute();and

wdContext.nodeResponse().invalidate(); see Figure 3.3:

The actual Web service call is executed via calling the execute()method of the executable model object (1) which is currently referenced in the context model node (2). This already contains the data entered by the user (through data binding and context mapping). The generation of the model object (1) and its binding to the node WebServiceEmail was implemented before in the method wdDoInit().

After calling the execute() method of the executable model object, the Web-Service returns a corresponding response object (3). The relation (5) of the executable model object (1) points to this second model object. However, the model node element (4) in the controller context does not yet point automatically to the corresponding response object (3) in the model. Due to this fact, you have to explicitly invalidate the model node Response (contained as an inner node below the node WebServiceEmail in the context). This is executed with the code line wdContext.nodeResponse().invalidate(). Thus, the Web Dynpro runtime environment is told to track the corresponding relation (5) during access to the response node element (4). In doing so, the reference of the response node element (4) in the context is adjusted to the response object (3) returned from the Web Service in the model.

The returned result (in the example application this is just a single integer value) is then displayed in an appropriate message text in the message bar of the Web Dynpro application.

After the implementation of the component controller, you only have to call the public method sendEmail() of the component controller in the action event handler onActionSendEmail() of the view controller to trigger the email Web-Service call. See Code Fragment 7.

```
//@@begin javadoc:onActionSendMail(ServerEvent)
/** Declared validating event handler. */
//@@end
public void onActionSendEmail(
  com.sap.tc.webdynpro.progmodel.api.IWDCustomEvent wdEvent ) {
  //@@begin onActionSendEmail(ServerEvent)
  wdThis.wdGetEmailWSComponentController().sendEmail();
  //@@end
}
```

Figure 3.9: Code Fragment 7

3.4 Malicious Code Injection

When an email address is entered by the user, there will be an underlying check to see if the format is correct, if the person exists in the organization and is still responsible for the maintenance of the material. The format check will be limited to verifying the presence of the @ character. However, the check to verify the person in the organization will access a database containing the master data, or the "organizational" data of the maintenance companies. These data could be stored at the client company or at the maintenance company. It is possible to craft SQL commands in the first fields of the email address that might delete entire databases with organizational data, or return confidential info, if there are no correct input filters in place. Also the invocation of the web service email function, could exploit vulnerabilities. Microsoft has documented vulnerabilities (and available patches) for various malicious code injections attacks on the Office Suite and SQL Server. We will explain more details behind the SQL injection attacks, in section 4.2.6.

Chapter 4

Evaluation of verification tools

4.1 Automatic type-based tools

This first category of tools include all the tools that use types to perform the analysis. One of the most important characteristics of such tools is that their use can often be automated, as it does not require the involvement of a human operator in order to carry out proofs.

4.1.1 Metrics and measures of success for case studies

Typing tools all work by approximating the behavior of a running program, in a way that makes it possible to reason on all the possible states of a program without actually running the program in all possible ways.

As a result, the objective of a tool which ensures a safety property is to prove that it is not possible, during the execution of a given program P, to find a program state that violates a given rule. Most type-based tools are correct, in the sense that they are able to identify all the programs that actually violate a given rule. However, all these tools are incomplete, in the sense that they will all identify as violating the rule programs that are harmless, and that will not lead to any unwanted state.

Such a program, which is correct, but for which a typing tool cannot determine that it is, is called a false positive. One of the metrics of a typing tool therefore is the proportion of programs that it identifies as false positive. This rate must be very low for the program to be usable as an automated tool.

The same reasoning applies to information flow analysis, but tools will report interference between data considered as inputs and outputs instead of invalid states.

This issue of false positives is a well-known issue with most static analysis techniques, and particularly of type-based techniques. Such tools are nevertheless used widely as pre-deployment scanning tools, mostly because of their performance and because of their possible automation. A type-based tool can check an application in a few seconds. In order to keep the proportion of false positives low enough, type-based tools are often only used to check simple framework-specific properties.

As these tools are automated and used as applications scanners, they should be able to process all kinds of applications, regardless of their origin. This is true for the simplest properties, which do not require any awareness from the developer. However, for some properties that are either more complex or more difficult to follow strictly, the tools can

only be useful if the developers are aware of the properties that their applications need to follow.

In terms of metrics, it is therefore very difficult to measure the performance and accuracy of a given type-based tool with a single figure. We therefore propose to select a few properties among the ones defined in deliverable 1.2, and to analyze how these properties can be verified on a few typical applications of various origins, sizes, and complexity. In each case, we will study the application in order to determine whether or not it verifies each property, and whether or not the automated type-based tool is able to determine that it actually verifies it. We will also consider the various challenges that have been identified in section 3.15.2 of Deliverable 1.2.

4.1.2 Resource policy verification

In Java, most resources in a mobile device are accessed through an appropriate API. Therefore, a resource policy is often stated as a limitation on the way in which a given API may be used. At the Java level, this corresponds to limiting the values that the arguments to methods can take.

We will first review a few objectives that are relative to such verifications. The second section is devoted to the other requirements expressed in deliverable 1.1.

Limitations on method calls

Simple properties These properties are those that can be easily checked, without requiring an in-depth semantic analysis of the application. They include in particular the following properties:

- Properties that forbid the use of given class or method.
- Properties that constrain a method argument that is expected to be a constant.

Here are a few examples of such properties, taken from Deliverable 1.2:

G-13 The application only accesses MSA system properties.

G-67 The application only uses private record stores.

In the first property, the MSA specification defines a list of properties that are supported by all MSA-compliant phones. Properties are regular strings normally used as a parameter to the System.getProperty method. In most cases, the argument to this method is expected to be a constant (the name of the property); this is why the property is simple to verify.

In the second property, the property entails two distinct subproperties. First, the application must only create record stores without sharing them. In terms of API, this means that the application should not use the openRecordStore method with four arguments, or if it does, it must ensure that its third argument takes the value AUTHMODE˙PRIVATE, ensuring that the newly created record store will not be accessible by other applications. The second sub-property is that the application must not open any record store that it has not created. In terms of API, this means that the application should not use the openRecordStore method with three arguments (which specifies a vendor name and a suite name).

Both properties are very simple to express in plain English, and their translation into code properties is reasonably straightforward. Although simple, such properties are widely used

in security policies, in particular in the policies that need to be enforced through automated tools. It is therefore necessary to ensure that these properties are proved correct in most applications. In particular, the rate of false positives must be as low as possible, in order to reduce the number of applications that need to be handled by human evaluators.

In terms of assessment, the idea is here to use a large number of publicly available applications (their source code is not a necessity) to verify that the properties are adequately identified.

Pattern matching In this category, the properties remain quite simple. The main difference is that the condition imposed on the method arguments appear more complex.

G-25 The application sends messages to a number matching a given pattern.

G-37 The application only establishes HTTP connections.

Both rules end up as constraints on String arguments, which are formatted as URL's. The first rule will need to analyze all the URL's that start with the string "sms://". The second rule will simply ensure that all network connections are initiated by URL's that start by "http://" or "https://", which in fact is implemented by verifying that the other network protocols are not used, because the rule must not forbid the use of non-network connections, such as messaging connections.

In both cases, the rules remain fairly simple to express. The main difference with the simple rules above is that applications are not expected to use constants as arguments to these methods; instead, these URL's will be the result of complex computations. This leads to many difficulties, such as the ones explained below:

- The URL's may be stored in containers such as arrays, vectors, and hashtables. If many URL's are stored in the same container, the abstraction may not be able to differentiate them.
- These URL's are likely to be stored together with the other application preferences. This means that they will be serialized to a record store, in which all preferences will be mixed. It is then difficult to infer any information about the data read from the record store, hence making the analysis difficult.
- Some application preferences can be edited by the end user, which means that their value can be a runtime data, which by definition is not known before execution. If this runtime data ends up mixed with the interesting data, it may be difficult to verify the property.

All these issues have in common the fact that they make the analysis of the application's behavior more difficult, and they may yield a large proportion of false positives. For this reason, such properties are expected to only be included in security policies of which developers are fully aware. This means that the developers should know that the policy will be enforced on their code and that they need to comply with it.

In that case, assessment is more difficult. Using a large number of applications will provide a rough idea of the algorithm performance, but the conclusion can only be significant after analyzing the reason of the failures. It seems more appropriate in that case to work with a smaller set of applications, to analyze carefully which features these applications use, and to assess how the type-based tool deals with these features.

Another possibility is to modify the applications, like we would do if we were the developers of these applications, realizing that something must be done in order to get the application somehow certified. In that case, the metric is the amount of work required to make the application pass the type-based tool.

Assessment of these properties could easily be based on the applications and security objectives that have been identified in Chapter 2.

We can either estimate bounds on the number of method calls for a given method independently of the arguments, or estimate such bounds for a given value (linked to a given instance) of one of the arguments.

In the first case, we are most often interested by raw bounds such as there is a risk of calling this method in a loop or not. This is well suited for type based analysis.

Security property 1. Check that a given method m (eg. sending an SMS) is never called in a loop.

The second case corresponds to the use of permissions separated from the consumption as presented in 2.4.4. Some of the cases may be beyond the scope of type based tools, especially if we want to allow parameters and not only constants in the number of permissions granted.

Security property 2. Check that for each permission P granted, its use cannot overrun the threshold declared when it was granted.

Counting resources

The previous section focused on resource restrictions that mostly consisted of restrictions applied to the way in which API's (and associated resources) are used, but in which resources could not be counted. There are many cases, however, where counting is an important factor.

There are many things that can be counted, as well as many ways to count things. The following rules give three examples, which correspond to different ways of counting:

C-4 The application only uses a number of threads.

G-32 The length of an SMS message sent does not exceed the payload of a single SMS message.

G-64 The MIDlet uses at most a number record stores at any time.

Counting the threads can be performed by analyzing the call tree of the application, and by modeling carefully the API and the possible interactions with the end-user. Computing the payload of an SMS message may be more complex, but there are ways to make this data easy to infer. Counting record stored also uses a different technique, but there are cases in which the counting is made easier by the way in which record stored are named. These three rules have in common the fact that they correspond to rules that may be enforced in a generic policy, and would need to be followed by all applications.

In addition, some properties may be used to described the specific features of an application. Such properties could be interesting for a developed, because they may allow her to explain to a potential partner/customer that her application actually behaves in an acceptable way.

C-9 The application sends no more than a number messages in each session.

G-66 The MIDlet stores at most a number kilobytes of information in its record stores.

In both cases, it is here required to count some events that are global to a session, or even to the lifetime of the application. For instance, the second property may be provable when a record store holds a fixed number of fixed-size records, but it will be very difficult to prove in the general case. In both cases, one of the issue is that, even if the application maintains a count of the messages sent or of the record store size in a variable, it is very difficult for the type-based tool to identify that variable and to prove the property. In such cases, logic-based tools, in which these variables can be identified, would do a better job.

Execution time estimation

Security property 3. A midlet UI should not be blocked and the time spent to react to user events should be below a given threshold.

The analysis would first use simple examples such as those presented in sections 2.4.1 or 2.4.3. The regression suite described in section 2.6.1 can then be used to assess whether the analysis scales well to real examples.

The code activated by user events is called from well-isolated callbacks. For basic high-level UI, it is either in a CommandListener object for a simple button or an ItemCommandListener or an ItemStateListener for a command in a form. A first rule is that blocking commands should not be called within those listeners. But this is too coarse if we want a really reactive midlet.

The verification of such a property could make use of the execution time estimation presented in section 3.4 of [25].

4.1.3 Information flow control

Generic requirements

We may require that some system properties are never transmitted on the network. Or at least we may require that such properties are known

Security property 4. the calls to MIDlet.getSystemProperty where the outcome is sent over the network must be identified

This property is generic. We can use midlets (see section 2.6.1) for which the code is not available to check how it is enforced.

RSS Reader

Security property 5. The information streams agregated by the reader must be kept separate and no information on the name of the readers must be transmitted on the network.

As this is a global requirement on the internal state, it should be easily translated as an information flow property provable by type based tools.

Pocket Protector

Security property 6. The contents of records stored on the RMS must be encrypted.

This means that all the contents of the arguments of method storing contents depend only (are) the output of the cryptography primitives. Those primitives must only be used in a context initialized with the right password. These are rather simple flow as we require equality between the origin (the result of a method call) and the destination (an argument of a method call).

J2MEMap

Getting a location from JSR 179 The current geographical location of the phone is followed by an object of interface LocationListener that contains callbacks called when a phone move is seen. The second argument of the method locationUpdated is the information to follow.

void locationUpdated(LocationProvider provider, Location location)

Listeners are registered by LocationProvider objects:

```
abstract void setLocationListener(LocationListener listener,
                                  int interval, int timeout, int maxAge)
```

Applications can also use a direct query with a method in: LocationProvider:

```
static Location     getLastKnownLocation()
abstract Location   getLocation(int timeout)
```

Getting a location from Bluetooth The link to data is not so obvious as InputStream objects can be created by different objects, not related to the bluetooth connections.

A call to a bluetooth service is obtained by opening a btgoep:// URL. The result is a Connection object but more precisely a ClientSession object that can perform get and put method calls. The results of those methods are Operation objects that contain the I/O streams.

```
interface ClientSession extends javax.microedition.io.Connection –
  ...
  Operation     get(HeaderSet headers)
  Operation     put(HeaderSet headers)
  ...
”

interface Operation extends javax.microedition.io.ContentConnection –

”
```

Security property 7. Identify the output streams on which information obtained from either kind of location service is leaked out.

There are two levels of analysis that can be performed: a simple one identifies the fact that there is an information flow between a bluetooth server and a WAN connection : one only need to identify the operation object.

In a second level of analysis, we link the Operation object to its originating bluetooth connection. This is needed if we want to use several bluetooth services. Only the URL can identify the kind of service used. Several information can also be retrieved from the headers associated to the operations performed.

Security property 8. Characterize the bluetooth services and how they are used by the application.

At least each individual analysis step required should be accessible to type based services.

Warning The interesting version has been compiled for Sony Ericsson K750, a CLDC 1.1 compliant phone. Support for floats and doubles is required to execute this version. At least the tool should not crash even if it cannot interpret floating point objects.

It also requires many libraries. Some of them have quite complex behaviours.

Information flows from user screens to network

Password protection

Security property 9. Passwords or any information got from the user must not be displayed in the User interface.

It is easy to list the methods that display something in the user interface (StringItems and textboxes, name fields of most UI components, etc.) Sensitive information should be entered in a field identified as TextField.PASSWORD.

Unfortunately this is not the case for every application and we need to specify application specific rules for these:

Security property 10. Pocket Protector : Check that the passphrase entered in the Unlock screen is not stored in the RMS (database on the phone).

Security property 11. Encryption is not done with a password stored in the application but with a passphrase typed by the user.

This rule applies to property 6. It also requires that we identify the screen where the password is taken/set.

Information sent over the network

Security property 12. For each text input widget, identify the network destination that may receive some information on what was typed by the user in the box

4.2 Logic based tools evaluation

This category includes all the tools not covered by the previous section, in particular tools that evaluate thread's concurrency and temporal dependency of some events. In most cases it will not be possible to check such properties in a fully automatic way.

Most of these tools will employ navigation graphs as a way of describing properties of the midlets. Detailed definition and their power of expression is included in Appendix.

4.2.1 Metrics and measures of success for case studies

Logic based tools trade-off complete automation for expressive and proving power. This trade-off should be justified by the kind of properties that can be proved.

We will try to show that the use of logic based tools allows us to treat much more complex properties, which nevertheless may be very important:

- either because the code is too complex and the developer had good reason for not sticking to a coding style that would have allowed automatic tool to extract the information : navigation graphs for complex applications are a good example of such properties,
- or because the property to check is too involved and its expression cannot be captured in the simple property language recognized by automatic tool. Properties relying on the naming of object instances such as security property 13 are good examples.

Even if their use is not completely automated, the level of automation and its opposite the amount of user interaction necessary to prove a property is also a good metric because it may refrain the adoption of such techniques by developers and validators.

Finally as proofs are intended to be used in certificates, the size of the original proof is a significant measure even if compression techniques are employed later to reduce the size of the actual certificate.

4.2.2 Verification of resource bounds

Although generally this property belongs to automatic type-based tools, some complex examples (as presented in property 2) may require the use of logic based tools handling complex arithmetic properties.

For example when the number of permissions has been defined on the size of a collection and it is then consumed on each element of the collection. It can even involve non-linear arithmetic if the collection is a multi-dimensional array for example.

Our analysis will be performed on midlets with known source code and include manual assess of complexity of such bounds. The measure of these test will depend on successes in various cases.

4.2.3 Information flow

Security property 13. Property of RSS reader: no information on the elements read of a given stream may be transmitted on another stream.

Although this property is not so interesting per se in this case, it may work as an example of some potentially important property. It is a pattern for applications aggregating multiple streams with sensitive data and where the privacy of the user with respects to each of those streams must be guaranteed.

The main difficulty is that streams are distinguished by the instance objects associated to them (the connection), not by the structure of the code as for property 5. Therefore, isolation of this streams can not be described by the navigation graph alone.

4.2.4 Verification of navigation graph

Given a midlet M and a navigation graph N, the goal of the experiment is to check:

Security property 14. M is conforming to the navigation graph N (all its potential behaviours are described by N).

The verification may be done at different level depending on the tools used:

- at the source level using for example ESC/Java2,s
- directly at the bytecode level using a formalization in Bicolano.

A formal model of the core of the high level graphic interface APIs (LCDUI) is required: Display, some Screen objects, Command objects and CommandListener and at least a partial model of a midlet life-cycle will be useful but there is no other dependency on the MIDP specification.

The first case study will be based on very small examples presenting several kinds of GUI implementations as described in section 2.4.3.

Then we will use midlets presented in 2.5 as examples of real midlets. For example, the PocketProtector mixes different ways of characterizing a state of the navigation graph.

Such examples will allow us to check the integration of type based techniques (see 4.1.3) and logic based verification of navigation graph.

4.2.5 Combination of properties: Momental Joker

We can also combine navigation graphs temporal properties and information flows. The Momental Joker application presents several occurrences where declassification is safe only at a certain point in time.

Security property 15. Temporal release of information flow in Momental Joker: declassification of card, player key and permutation only occur when this should occur.

As explained in [1], seals can be used to check dynamically that some temporal constraints are verified before revealing data. A seal is a boolean value associated to a temporarily private data. It is switched only once when the data becomes public. As long as the game is running, the integrity of seals is checked. Before using the data as declassified, one must switch the boolean value with the unseal primitive.

With the help of navigation graphs, it is now possible to statically check those constraints that ensure no runtime errors.

Security property 16. unseal is never used before assertIntegrity on a given seal. declassify is never used before unseal on a given seal.

Checking this property on turned cards will be much harder than for the keys and permutation as the property is tied to a particular instance of a card object.

4.2.6 Verification of application specific properties of applications based on SAP NetWeaver

Inadequate data validation is the most common cause of security exploits suffered by web applications today [21]. There is an ever increasing number of applications exploited through weak validation. This in spite of the simplicity of such an attack.

SQL Injection happens when a developer accepts user input that is directly placed into a SQL Statement and does not properly filter out dangerous characters. This can allow an attacker to not only steal data from the database, but also modify and delete it. Certain SQL Servers contain Stored and Extended Procedures (database server functions). If an attacker can obtain access to these Procedures it may be possible to compromise the entire machine.

A cross-site scripting vulnerability (XSS) is also caused by the failure to validate user supplied input, before returning it to the client system. "Cross-Site" refers to the security restrictions that the client browser usually places on data (i.e. cookies, dynamic content attributes, etc.) associated with a web site. By causing the victim's browser to execute injected code under the same permissions as the web application domain, an attacker can bypass the traditional security restrictions which can result not only in cookie theft but also account hijacking, changing of web application account settings, etc.

We can solve these issues by just correctly filtering out the input provided by the users. Unfortunately, as engineers are under increasing pressure to complete work on time and because of insufficient knowledge, most of the applications remain vulnerable to these attacks.

In section 5.1.2 of Deliverable 1.2, temporal security requirements are introduced, that restrict the history of method parameters, i.e., it describes which operations must have been applied to a method parameter before the method call. We described how JML ghost fields can be used to encode these properties.

In Chapter 3 we presented a practical example, where a maintenance message can be sent through the NetWeaver Portal. If the parse method is not correctly filtering and validating the input, we can have unexpected behaviour. The method itself can be quite complex since it has to be parameterized for the different maintenance companies. Therefore it is sometimes difficult to track for a developer if the variables have been passed through the filter which removes dangerous characters. It should also be clear from the code examples given in Chapter 3, that the filtering can be handled at different places in the data flow.

```
String addressfield = IPrivateForm.IContextElement.E˙MAIL˙ADDRESS;
String firstname = parse("FirstName", addressfield);
String lastname = parse("LastName", addressfield);

String sql = "Select * From PLMOrgData Where FirstName = '" +
firstname + "' AND LastName = '" + lastname + "'"

final Statement s = con.createStatement();

final boolean rsReturned = s.execute(sql); if (rsReturned) –
  ResultSet rs = s.getResultSet();
  rs.close();
˝ else –
  // do something ?
```

```
″

if (rs != null) –
 // user exists ″

else – throw new NoSuchUserException(””); )

s.close(); con.close();

″
```

When a user enters following email address:

```
Bob';DROP PLMOrgData --'.Smith@Lockheed.com
```

It will create this sql query:

```
"Select * From PLMOrgData Where FirstName = 'Bob';DROP PLMOrgData--'
AND LastName   = 'Smith'"
```

It will delete the entire PLMOrgData database.

The most critical step in preventing against these attacks is to employ server-side validation on any input data used as part of a database call. Validation code should test for validity, ensuring that input data is of the expected type and length, rejecting any invalid data. To further prevent against this attack, we should prefer the use of java.sql.PreparedStatement over java.sql.Statement and java.sql.CallableStatement when calling stored procedures. Also we should never establish a database connection using an administrator account. Instead, provide role-based logins with the minimum privileges required to carry out a particular task. These rules are known, but it remains difficult to let them be enforced by the developers.

As described more detailed in example 5.1.1 of D1.2, in order to store whether a database input has been validated, we can use a ghost field:

```
public class Input – ...
//@ ghost public boolean valid; // a boolean ghost field
//@ initially valid == false; // constructors must initialise this
// to false
/*@ ensures valid == true; @*/

public void validate () throws InvalidException– ...

/*@ set valid =true; @*/ // assign true just before return ″ ...
// all mutators ensure valid == false
// the object state is properly encapsulated
... ″
```

This specification expresses that after application of validate() an input object is considered valid. To ensure correctness of the specification of method validate(), we have to insert appropriate set-annotations at every possible normal (i.e. exception-free) exit point of the method. Note that here validity means merely that validate() has been applied. The specification says nothing about the functionality of validate(), in particular, it does not say that validate() implements input validation correctly. Still, this specification is useful if the Input class is trusted or its functional correctness has been verified by other means. We can now restrict a non-validating database class to only accept valid input:

```
public class DataBase – ...
 /*@ requires in.valid @*/
 public void enter (Input in) –...″
 ...
```

Note the comments at the bottom of the Input class: In addition to validating input before database entry, it is just as important to ensure that the input does not mutate after validation. Therefore, mutating methods must reset the ghost field valid to false. Moreover, it is important that Input objects cannot be mutated through aliases that bypass the API, that is, the state of Input objects must be properly encapsulated.

4.3 The proof carrying code framework

4.3.1 Introduction

As a principle, any proof should have an equivalent as a certificate and so any proof performed to prove a property described in the previous chapter should have an equivalent certificate. The main challenges are then more of a technological nature:

- Is there a bound on the size of the certificates ? Performance evaluation will be an important success criteria: memory consumption, CPU required, etc.
- How much of the execution environment models will we support ?

There are a few use cases where PCC techniques have a more immediate impact. Those cases will be presented in the next sections.

4.3.2 Logic-based proof carrying code and Certified navigation graphs

Description

Certificates will encode Coq proofs written in the Bicolano framework. Proofs will be done directly at the bytecode level. The goal of this case study is to generate proof certificates for proofs of conformance of a midlet to a given navigation graph. The potential graphs and applications have been presented in 4.2.4.

Usage scenario

The navigation graph of an application is best known by its developper as classically, development begins with the design of a "mockup" that represents the structure of the various screens of the application and their relations (For a presentation of a development methodology, one can read Sun style guide for MIDP [5]).

Implementation constraints may blur this initial design because due to size and efficiency constraints it may be necessary to reuse objects for several screens. Moreover there are usually very few command listener objects implemented to handle potential events, making the discovery of the potential transitions difficult for an external validator.

But if the validation of a given navigation graph is done by the programmer, we need a way to convey the proof to the validator so that he can safely use the information contained in the graph.

Generating and proving a navigation graph may be a tedious task. A powerful IDE that generates part of the GUI code (such as Netbeans Mobility Pack [1] may assist to generating the graph (it is a direct side product of the GUI representation in the IDE) and the conformance proof.

4.3.3 Lightweight proof carrying code

Introduction

The goal is to test the on-device proof carrying code infrastructure developed in the MOBIUS project. It will rely on the framework for proved static analysis developed by INRIA team LANDE [3]. This framework also relies on Bicolano, but here the analysis is performed by an analyzer extracted from a certified algorithm.

This analysis can be seen as extensions of the principles used in the two phase bytecode verification algorithm used in the implementation of CLDC and in last versions of JavaSE. First the property is checked off-line, the proof involves the discovery and the construction of a complex graph-like structure associated to the representation of the abstract state result of the fixpoint computation of the analysis. A minimal part of this structure is extracted and sent along with the code so that a checker on-device can compute again a solution to the problem but in linear time using only local operations.

Additional information required by the checker can be supplied as custom code attributes (see Java attribute format definition in section 4.7 of [20]).

Several analysis can be performed, we present two such analyses that are the best candidates both in terms of simplicity and potential industrial use.

Counting accesses to controlled resources

[4] presents a simple resource analysis that can check that a multiple permission granted to a controlled API is not consumed beyond what was requested.

It is important to check that APIs whose use is charged (SMS mainly) are not called in a loop and that the application will not dynamically fail due to an incorrect value in the number of authorized uses requested.

[1] www.netbeans.org/products/mobility/

Chapter 5

Conclusion

5.1 Coverage of requirements expressed in Work Package 1

The applications presented in this deliverable try to cover the full range of complexity from academic examples that pinpoint a specific programming practice to complete complex industrial applications.

We try to use existing examples as much as possible : first to avoid using project resources on tasks that are not directly related with the goal of the project but also because it is much more convincing to demonstrate tools on code we have not developed.

Unfortunately, in the case of midlets, it is difficult to get access to midlet code of good quality. Even if most of those programs are innocuous and do not try to misuse resources of the host system, they enforce very few of the specific security properties tied to their intended use that one would expect them to fulfill.

So we have decided to limit our set of examples to relatively trusted codes.

Properties have been presented classified per verification technique to ensure a reasonable coverage of the work done in each work package but they also cover the contents of deliverables 1.1 and 1.2.

The list of case studies is by no way closed. If good example with interesting properties arises, for example provided by a member of the End-User-Panel, we will seize the opportunity if it does not require too much overhead to take the environment (mainly libraries) of the application into account.

5.2 Intended use of the case studies

As previously mentioned, the objective of these use cases is to provide practical examples to be analyzed with the various tools developed in MOBIUS. These use cases will be used by the teams involved in the evaluation of MOBIUS output (mostly industrial partners) but also by developers to test their own tools.

However, these examples are based on applications easily available today. They will be used to test tomorrow's tools, which may introduce some new requirements (for instance, making some new intermediate security requirements explicit). Depending on the tool's characteristics and on the properties to be verified, the use cases may be annotated and/or modified, in order to check a specific feature/property. The same use case may be used in different contexts/configurations in order to compare some tools or techniques. This may lead

to the use of a combination of tools to prove a single property which will validate the proof framework approach.

The notion of navigation graphs presented here as a way to capture different kinds of properties deserves a special mention. Its definition and semantics will probably evolve and be clarified as its use with the various tools will become clearer. As mentioned earlier, the availability of a unifying formalism describing in an univocal way the expected behaviour of a MIDP application as far is concerned, has a potential use that goes beyond the mechanized verification of those properties. But the capacity to treat those properties with the tools developed within MOBIUS will be a strong evidence of its adequacy.

Some properties have been outlined in the present report, but the result of the final assessment will not be a "pass/fail" result. Instead, many parameters will be taken into consideration, and the testing may require to work with additional use cases, either derived from the existing ones, or added for checking a specific property.

5.3 The next steps

This deliverable has also made explicit some requirements on the tools so that that they can treat the examples proposed: first they must support all the constructs present in the examples but they must also take into account the environment. We expect that we can share most of this specialization work between different tools. We will address these problems in the following months to prepare the evaluation phase.

In the meantime, new case studies may be added if a need appears.

Bibliography

[1] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In European Symposium On Research In Computer Security, number 3679 in Lecture Notes in Computer Science. Springer-Verlag, September 2005.

[2] K. Ballinger, D. Ehnebuske, M. Gudgin, M. Nottingham, and P. Yendluri. Basic profile version 1.0, 2004.

[3] F. Besson, T. Jensen, and D. Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. Theoretical Computer Science, 364(3):273–291, 2006. Extended version.

[4] Fréderic Besson, Guillaume Dufay, and Thomas Jensen. A formal model of access control for mobile interactive devices. In Computer Security — ESORICS 2006, Proceedings of the 11th European Symposium on Research in Computer Security, number 4189 in Lecture Notes in Computer Science. Springer-Verlag, 2006.

[5] C. Bloch and A. Wagner. MIDP Style Guide for the Java 2 Platform, Micro Edition. The JavaSeries. Addison-Wesley, 2003.

[6] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language, 2006. W3C Recommendation 27 March 2006.

[7] L. Clement, A. Hately, C. von Riegen, and T. Rogers. Uddi version 3.0.2, 2004.

[8] Tyler Jewell Dave Chappell. Java Web Services. O'Reilly, 2002.

[9] B. Goetz, editor. Java Concurrency in practice. Addison-Wesley, 2006.

[10] Unified Testing Initiative. Unified testing criteria for Java technology-based applications for mobile devices. Technical report, Sun Microsystems, Motorola, Nokia, Siemens, Sony Ericsson, May 2005. Version 2.0.

[11] JSR 118 Expert Group. Mobile information device profile (MIDP), version 2.0. Java specification request, Java Community Process, November 2002.

[12] JSR 120 Expert Group. Wireless messaging API. Java specification request, Java Community Process, 2003.

[13] JSR 135 Expert Group. Mobile media API. Java specification request, Java Community Process, June 2003.

[14] JSR 166 Expert Group. Concurrency utilities. Java specification request, Java Community Process, 2004. Final release.

[15] JSR 172 Expert Group. J2ME web services specification, version 1.0. Java specification request, Java Community Process, March 2004.

[16] JSR 177 Expert Group. Security and trust services API for J2ME. Java specification request, Java Community Process, September 2004. Final release.

[17] JSR 205 Expert Group. Wireless messaging API (version 2.0). Java specification request, Java Community Process, June 2003.

[18] JSR 218 Expert Group. Connected limited device configuration (CLDC), version 1.1. Java specification request, Java Community Process, 2002.

[19] J. Knudsen. Networking, user experience, and threads. Sun Technical Articles and Tips http://developers.sun.com/techtopics/mobility/midp/articles/threading/, 2002.

[20] T. Lindholm and F. Yellin. The Java^{TM} Virtual Machine Specification. Second Edition. Sun Microsystems, Inc., 1999. http://java.sun.com/docs/books/vmspec/.

[21] V. Livshits and M. Lam. Finding security vulnerabilities in java applications with static analysis. In USENIX Security Symposium, 2005.

[22] MOBIUS Consortium. Deliverable 1.1: Resource and information flow security requirements, 2006. Available online from http://mobius.inria.fr.

[23] MOBIUS Consortium. Deliverable 1.2: Framework-specific and application-specific security requirements, 2006. Available online from http://mobius.inria.fr.

[24] MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic, 2006. Available online from http://mobius.inria.fr.

[25] MOBIUS Consortium. Deliverable 4.1: Scenarios for proof-carrying code, 2006. Available online from http://mobius.inria.fr.

[26] A. C. Myers. JFlow: Practical mostly-static information flow control. In Principles of Programming Languages, pages 228–241. ACM Press, 1999. Ongoing development at http://www.cs.cornell.edu/jif/.

[27] A. Nadalin, C. Kaler, P. Hallam-Baker, , and R. Monzillo. Oasis web services security: Soap message security 1.0 (ws-security 2004), 2004.

[28] M. Veit and S. Herrmann. Model-view-controller and object teams: A perfect match of paradigms. In Aspect-Oriented Software Development Conference, pages 140–149, 2003.

Appendix A

Interface and security behaviour outline for JavaME MIDP applications

A.1 Rationale

Deciding statically whether an action performed by a midlet is dangerous for the user requires more information than the mere knowledge of the dangerous method called by the application and their parameters although this is a prerequisite for any analysis.

Let us assume that we know that a method call in the application may send a SMS, this may be a perfectly acceptable behaviour, for example to send a "highest score" when the game is finished. But this would not be acceptable if:

1. Several SMS are sent in a tight loop in a single transition of the user interface.
2. SMS are sent in several (too many) transitions of the user interface (for example each time the back key is pressed to return to the previous screen).

The first point may be solved by a resource analysis of the midlet. The intended behaviours can be captured by rather simple security automata involving the critical actions performed by a code segment of the midlet.

The second point requires that we analyze also when the action occur with respect to a global view of the potential user interactions. Deciding whether a behaviour is acceptable or not is probably out of reach of formalization, so we will concentrate our work on defining a formal view of those behaviour that can be analyzed by a human being and checking in the most efficient and automated way that midlets effectively follow those behaviours with the tools developed in the project.

This chapter describes navigation graphs, a first attempt to provide a formalism that describes when critical actions occur in a synopsis of the user interface of a midlet. This view will also be used to express information flow relations between different elements of the midlet. For example between a form entry in a given screen and an opened network connection.

Navigation graphs will mainly be used with logic based tools. The goal of verification will be to establish a link between the code of the application and a given navigation graph that establishes that any execution trace of the application can be associated to an abstract execution in the navigation graph. But the representation will also be used in this document to present the behaviour of applications.

This formalism may be used independently of the tools developed in the course of the project but its use with proof tools will help us to assess if it is adequate for the assigned task. Navigation graphs may be used as a basis for a standardized formal representation of midlet behaviours and as a replacement to the informal representation used by the Unified Test Initiative (see an example on page 9 of the version 2.0 of UTC specification [10]).

A.2 Definition and semantics of navigation graphs

This section presents an overview of what could be a formal semantics of navigation graphs. We first present an extended semantics of midlets that takes into account:

- the effect of the program on external resources,
- concurrency.

The main result is the characterization of the midlet behaviour as a trace of side effects.

We then formally describe navigation graphs and executions as a walk in those graph. We can then give a precise definition of "being compliant with a navigation graph" by relating the two notions of traces.

Then we present extensions of the formalism to handle three important notions introduced by MOBIUS:

- enhanced counting of resources,
- the modelling of concurrency in midlets,
- the coding of some information flow properties in navigation graphs.

A.2.1 Traces and executions of a midlet

Modelling the machine state

The small step semantics of a java program (see for example section 2.3.2 in [24]) can be stated as a relation $\rightarrow$ between representations of the machine states of the virtual machine.

Classically,a machine state S consists of a representation of the heap h, the current frame f and the frame stack sf . To take other threads into account, it is necessary to add a thread pool p where idle threads are represented by their current frame and frame stack. We will write (h, f, sf, t) such a state where f can be decomposed as (m, pc, l, s) where m is the current method, pc the program counter in this method l the locals and s the operand stack. We will write $I(m, pc)$ the instruction of m at index pc. We will write $m(S)$, $h(S)$, $f(S)$ etc. the current method, heap, frame of state S.

A trace of a program is a sequence $(S_i)_i$ where $S_i \rightarrow S_{i+1}$ and S_0 is the initial state.

Modelling the external resources

This model does not take into account the external resources of the program that are implemented as native primitives such as the GUI or network connections for example.

To address this, we will also consider that the state of external resources is faithfully described by the state of internal data structures on the heap. We will define an interpretation function $[\![h]\!]$ that will give back a model of the external resources (mainly the user interface).

$[\![\ ]\!]$ provides a faithful view of the heap and so should have all the properties of a Galois connection with its co-concretization function.

In our setting, the range of $[\![\ ]\!]$ is $UI \times A$ the product of a user interface state with a current action state.

We will model the MIDP environment as a pure Java program, representing external events as the result of some non deterministic function.

User interface

The user interface state $ui \in UI$ is modelled as a triple made of an environment function ρ that defines all the existing objects and a current state coding a visible screen (also called displayable) $d \in D$, and a queue of events e where each event is coded as $(c, d) \in C \times D$.

An environment function ρ links d to a triple made of an object reference o, to a set of commands $\mathcal{P}(C)$. A command is coded by its object reference.

Now we can state rules for MIDP methods manipulating the interface state.

- We consider full executions of MIDP routines :

$$(h, (m, 0, l, a_0, a_1 ... a_n), sf, t) \rightarrow^* (h', (m, pc, l, ..), sf', t')$$

  where $I(m, pc) = return$.

- $[\![h]\!] = ((\rho, d, e), a)$ and $[\![h']\!] = ((\rho', d', e'), a')$

If $m =$ Display.setCurrent then $\rho(d') = (a_1, cn)$, $\rho' = \rho$ and $e = e'$.
If $m =$ Displayable.addCommand then $\rho(d') = (a_1, cn)$, $\rho' = \rho$ and $e = e'$.
We can also state how events are added to the interpretation:

> if $(h, ...) \rightarrow (h', ...)$, then $e' = e$ or $e = (c_0, d_0) :: e'$ or $e' = e@(d, c)$ where $\rho(d) = (a, C)$ and $c \in C$

Actions

Actions $a \in A$ are kept abstract and we will not further describe their domain now. There is at most one action active at a time and $\perp$ is an element of A representing that there is no action. The intuitive idea is that the interpretation of the heap gives back a value different from $\perp$ only when a critical function is executed.

If $(h, ...) \rightarrow (h', ...)$, then $a = a'$ or $a = \perp$ or $a' = \perp$.

Visible execution trace

An abstract trace of P is a sequence $([\![S_i]\!])_i$ where $(S_i)_i$ is a trace of P.

An event trace is a sub-sequence $([\![S_f(i)]\!])_i$ (f being a strictly increasing function) of an abstract trace such that for all i, let $S_i = ((\rho, d, E), a)$ and $S_{i+1} = ((\rho', d', E'), a')$ then the following condition holds:

- if $\forall j. i \neq f(j)$ then $(d = d' \ \wedge \ (E = E' \ \vee \ E = E'@e) \wedge (a' = \perp \vee a' = a)$.
- if $i = f(j)$ then $(E = e :: E' \ \wedge \ d = d') \vee (a = \perp \wedge a' \neq \perp \wedge E = E' \wedge d = d') \ \vee \ (E = E')$

In the first case of the second item we say e is the witness w_j, , in the second case a' is the witness and in the third case, it is d'. Trace steps corresponding to the first item have no associated witness.

The witness trace associated to the execution is the associated sequence of witnesses $(w_j)_j$. Elements belong to $D \cup C \cup A$ Because we take into account transitions that do not change the screen, there is not a unique witness trace.

A.2.2 Definition of navigation graphs

A navigation graph is an automaton whose state specify the current screen s and transitions represent the consumption of an event leading usually to a screen change.

Definition

States $n \in N$ are either sets (predicates over) display objects as defined above or one of the special *Start End* state.

$$N = \{Start, End\} \cup N' \text{ where } N' \in \mathcal{P}(\mathcal{P}(D))$$

Transitions are four-tuple in $N \times C \times (A \to \mathbb{N}^\infty) \times N$ describing the source, the command that triggered the transition, a multiset of actions and the destination screen. $\mathbb{N}^\infty$ denotes natural numbers augmented with $+\infty$.

Executions

An execution of a program (N, T) is a sequence of transitions $(t_i)_i$ where $\forall i.t_i \in T$ and if $t_i = (s_i, c_i, A_i, d_i)$ then $s_{i+1} = d_i$ of a program (N, T).

A trace $(x_i)_i$ associated to an execution is a sequence of elements of $D \cup C \cup A$ such there exists a strictly increasing function f $x_f(i), ...x_{f(i+1)-1} = ((d_i, c_i, a_{i_1}, ..., a_{i_p})$ such that if $t_i = (s_i, c'_i, A_i, s'_i)$ then

- $d_i \in s_i$: the current screen is a member of the source node
- $c_i = c'_i$: the command is the command associated to the transition
- $\forall a \in A.card\{j/a_{i_j} = a\} < A_i(a)$: the number of actions performed is less than the number of potential actions indicated on the graph.

Compliant programs

An execution of a program is compliant with a navigation graph if its associated witness trace is an execution of the navigation graph.

A.2.3 Enhanced counting of resources

The semantics given here assumes that the number of occurrences associated to an action is fixed and so independent of the context of the action. There are several use cases where this is not the case for example when a message should be sent to all the contacts selected by the user. This may be specified by the length of a data structure given back by an API. The list may be built by the user by several iteration over a given screen.

We must introduce new constraints in the specification to take this problem into account. The idea is similar to the way multiple permissions are formulated. Some actions explicitly (it is a permission request) or implicitly (the user selects fields) grant rights that are consumed later.

Those rights can be modelled by counters that are modified by actions. Potential actions may set (reset if the associated value is 0), increase or decrease the value of the counter.

Some values associated to an action may be dependent on the structure of an argument of this action (for example, its length for a list).

Those counters must remain positive (if one omit reset, actions of increasing and decreasing describe Dyck words).

Instead of using multisets coded as function $A \rightarrow \mathbb{N}$, we use a more complex definition $A \rightarrow (\mathbb{N}^{\infty} \times (CO \rightarrow \mathbb{Z}) \times (CO \rightarrow \mathbb{N}^{\perp}))$ where CO denotes the set of counters, $\mathbb{N}$ positive integers, $\mathbb{Z}$ integers. The first element denotes the absolute number of , the second element describes how each action increments counters and the last one how they set counters. The value associated is $\perp$ if the counter is not reset.

The definition of an execution must keep track of the current value of the counter states.

A.2.4 Extensions to threads

Threads can be dynamically created by the midlet. Their main purpose impact is that some critical actions are now performed independently of the user interface behaviour.

There are two approaches to code threads:

- a structural approach where each thread described in the code is coded as an automaton
- a denotational approach based on bisimulation equivalence where we only model the temporal relations between actions and events.

The structural approach

Each class implementing Runnable and used to create a thread is described by a separate automaton. We extend the notion of interpretation of the state to thread states (ie, their current frame and frame stack). Events are internal virtual machine events that act on the thread state so that it changes from a blocked state to an active state (notification, reception of data, creation, join event).

Corresponding action in other threads that may create the event must be registered as critical actions (notify, end of threads, calls to Thread.start).

Links are established between automata and represent how actions on threads and thread events are related.

An execution of a program is compliant with a graph if and only if its abstraction for the user interface is compliant and the abstraction along each thread is also compliant with an automaton that describes the class implementing the code of this thread. Moreover synchronizations must refer to the same object.

The complete formalization of such a semantics has not been made yet.

Denotational approach

In the second approach, we only model the visible behaviour of the application, not how it is implemented. We modify the definition of transitions to include two multi-sets of actions.

Transitions are five-tuple in $N \times C \times (A \rightarrow \mathbb{N}) \times (A \rightarrow \mathbb{N}) \times N$ The first one describe synchronous actions that will happen between the current event and the next one and the second one asynchronous ones that may happen eventually after the next event.

An execution of a program (N, T) is a sequence of triples of $T \times (A \rightarrow \mathbb{N})) \times (A \rightarrow \mathbb{N})$: $(t_i, b_i, u_i)_i$ where $\forall i.t_i \in T$ and if $t_i = (s_i, c_i, A_i, A'_i, d_i)$ then $s_{i+1} = d_i$ and $b_i + A'_i = b_{i+1} + u_i$ where $+$ is the multi-set union defined as a point-wise addition on the characteristic functions. b_i is the set of current deferred actions that may eventually happen and u_i is the set of deferred actions that have been consumed at that state.

A trace $(x_i)_i$ associated to an execution is a sequence of elements of $D \cup C \cup A$ such there exists a strictly increasing function f $x_f(i), ...x_{f(i+1)-1} = ((d_i, c_i, a_{i_1}, ..., a_{i_p})$ such that if $t_i = (s_i, c'_i, A_i, A'_i, s'_i)$ then

- $d_i \in s_i$: the current screen is a member of the source node
- $c_i = c'_i$: the command is the command associated to the transition
- $\forall a \in A.card\{j/a_{i_j} = a\} < A_i(a) + u_i(a)$: the number of actions performed is less than the sum of potential immediate actions as indicated on the graph and of consumed delayed actions as indicated in the trace.

There are several advantages:

- This is more promising if we want to reuse navigation graphs with different implementations of the applications.
- The specification is usually much simpler.
- We do not need to change the definition of compliance.

On the other hand, this level of abstraction has a cost: it is more difficult to relate the behaviour of an implementation with a navigation graph. Furthermore complex interactions coming from hidden events may be hard to describe and the expressive power is reduced: there is no way to describe how deferred events are sequenced.

A.2.5 Information flows

The main purpose of incorporating information flows information in navigation graph is to track how data (and especially PIN, password and other kinds of credentials) coming from user input are used by the midlet.

If an item $d \in D$, described as a sub-component of a screen node of a navigation graph that can receive text input is flagged as an information source, then for all execution of a midlet compliant with this graph, when an object is associated with its node and there is a corresponding sub component, this sub-component must be considered as a datasource for information flow analysis.

Every write on a network connection, or write to the store is considered as a visible action (an element of A) and as potential data-sink.

We identify authorized transfers of information as a relation IF in $D \times A$ that links some data-sources with some data-sinks.

The only datasink that can leak out information coming from an identified datasource must have an explicit link in the navigation graph. Rephrased in term of independence, this means that the value output by actions that are not linked to a given data source must not depend of the value input in those data source.

A.3 Representation of navigation graphs

We propose an XML syntax for the graphs and an associated graphical representation. The XML syntax will be the reference syntax for the tools. Its contents may evolve as problems appear but it should be rather stable.

The scope of the graphical representation is more limited as it is tied to and limited by the simple tool we have designed to build navigation graphs. It will be used in this deliverable to present the selected case studies.

As the goal of this deliverable is the presentation of examples, we will put more emphasis on the graphical representation used throughout the deliverable.

A.3.1 Graphical representation of navigation graph

navigation graph graphical syntax is heavily dependent on the tool used for drawing graphs : dia.

As it is difficult to annotate arrows and give them specific styles, the constraint is that to each element defined in the textual syntax corresponds a node of the graph. Specific annotated links are represented by a sequence of a regular link, an annotated object and a link.

The resulting visual syntax is presented in figure A.1. It only partially models the extensions presented before. Moreover

A.3.2 Examples of diagrams

This is not a manual for the tool and it only provides information to understand diagrams but it does not list the syntactic restrictions that make them parsable. The interested reader should use the tool manual instead.

Classical transition

Figure A.2 describes a classical transition between an alert and a form triggered by pressing the 'command' button and performing a send action. Node (3) acts as an intermediate fake node to which all actions are linked.

Threads

We present the syntax for explicit structural representation of threads. Internal thread states are represented by anonymous black square. Events are synchronization events. Actions are coded with the same syntax as for the GUI automaton.

Threads synchronizations are described by an element put between the action of the controlling thread and the receiving thread. A typical thread creation is given in figure A.3. Node (3) which is an action node and (7) which is used for special events that are not user events act as the origin and destination of the special link established between the automata. The semantics of the link is described by node (5). Here it is a thread creation, but it could be a notification too.

Note that the second automaton is an application thread so its node have a different syntax (see node (6) and (10)). The first automaton could have been an application thread too but here it is the GUI thread.

Basic nodes	
	The entry point of the graph. It should be unique.
	The exit point of the graph. It should be unique too.
	The entry point of a thread (one per thread).
	The exit point of a thread (one per thread).
	an anonymous state for a thread.
Screen nodes	
	An anonymous screen (unspecified class)
	An alert screen
	A form screen
	A list screen
	a canvas
	A textbox
Event nodes	
	A command event
	a thread related event (described by the incoming event kind)
	a keyboard event (canvas)
	a pointer event (canvas)
Action nodes	
	the action of a transition described by a set of thread event kind or a critical action description)
	start thread notification
	kill thread notification (use deprecated)
	wake-up notification (call to Java notify method)
	critical action description
	a network connection
	private user data
Critical calls arguments	
	base argument of a virtual method call
	first argument of a method call
	second argument of a method call
	third argument of a method call

Figure A.1: Dia elements

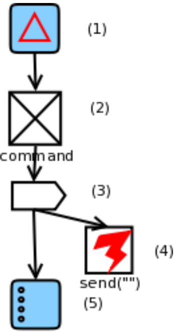

Figure A.2: A transition in a diagram

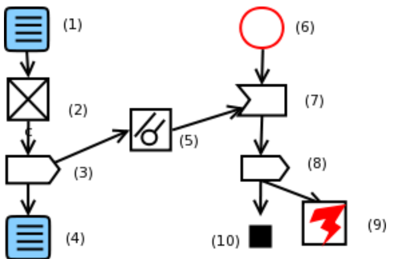

Figure A.3: Creation of a thread

Critical calls

A call is represented by a special kind of nodes used as a child of the action node in a transition. A call can take several arguments represented by nodes containing the position of the argument. The argument 0 is the base object for the call. Each argument contains its possible values as text or is linked to other objects that can provide parts of the values (mainly textbox and textfields)[1].

There are too kinds of critical calls: those that create a network connection and the others. When a connection is created, it can be seen as an object too. That can be used as a sink for private data (it is used to send private data over the network).

Private data belong to screens that contain the component where the user data is entered (a textfield or a textbox). In figure A.4, we have represented a connection. Its first argument is the URL of the other endpoint. It is an http connection as indicated by (3) and its destination address comes from a user field (textbox (1)). It is used to exchange data coming from the second textbox (4). The data typed by the user in those textboxes are considered as private and cannot be used elsewhere.

The reader should note that this graph does not track where the actual sending occurs, but only the opening and the flows. It is unclear that knowing when the transmission occurs would add any security information.

[1]We will probably need to refine this part.

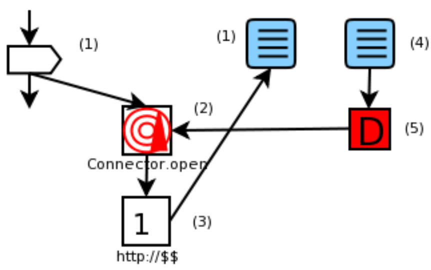

Figure A.4: A network connection