1. Introduction

In systems such as mobile phones, smart cards, washing machines or cars, the number and size of software doubles at each new series. The absence of defects in these pieces of software is a major objective for all actors of these domains, as all errors lead to enormous extra costs (products recall, loss of clients…). Over the past few years, the Java programming language has enjoyed unprecedented success. Even though, there is a pause in its propagation, it is certainly the key language in Internet and wireless digital communication. These domains make profit of the two main features of the Java language: applications created in Java are secure and platform-independent. Software developers benefit by developing code only once, with no need to “port” their applications to every software and hardware platform.

Run-Time Errors are a particularly high-risk type of software fault whose consequences include processor halt, data corruption and security breaches. They also cause applications to send uncontrolled commands to external devices, causing non-deterministic, unpredictable behavior. Impacts on business and corporate image may be catastrophic (due to loss of service, loss of mission, etc.). Meanwhile, run-time errors are very costly to find and fix since tests do not directly detect them. Tests may catch their consequences (memory corruption, functional failure, and processor halt), but not the error itself. Although extremely time-consuming, debugging and code instrumentation are widely used techniques to trace the source of the error in the code. PolySpace Verifier is the industry's first tool designed to directly detect run-time errors and non-deterministic constructs in applications at compilation time. Further, PolySpace Verifier does not require execution and modification of the code or time-consuming test cases to be run. Instead, PolySpace Verifier exactly pinpoints the faulty code section that will cause a run-time error if the application was executed. PolySpace Technologies has already developed a suite of software for static runtime error detection for Ada 83 and C programming language. These tools are already widely used in critical real time software.

The Java programming language is now widely used in wireless communication as well as in electronic business. Run-Time Errors detection is a key problem also in these domains. Smart cards, for example, are not just any computing platform; they are intended to be trusted. They may contain a key to your bank account, your use-rights as a subscriber to pay-television channel or your emergency medical data. A runtime error may then lead to tremendous financial loss, or to private information violation.

This project aims at the development of a PolySpace Verifier for the Java programming language in collaboration with INRIA and Gemplus. As far as we know, there is no industry tool developed to directly detect run-time errors and non-deterministic constructs in Java.
applications at compilation time. This project is a technology breakthrough in the Java world that will enable users to prove their critical applications safe. In particular, the treatment of the Java programming language by abstract interpretation requires the resolution of method invocation, and exceptions. More generally, specific Object Oriented features have not yet been tackled for industrial size applications. The introduction of parallel computation into abstract interpretation is the other innovation of the project.

2. Runtime Error Detection

**General Overview**

It is generally accepted that the main criteria for correct software are:

- functional correctness: the software computes correct outputs;
- temporal correctness: the software computes outputs within specified time bounds;
- robustness: the software does not spuriously halt, crash or behave erratically because of runtime errors.

In this paper we address the robustness problem. More precisely, we aim at simultaneously help end users: automate specific software development and verification processes, and increase the reliability of software.

Conventional testing is of unquestionable practical value for validating the functional correctness of software components. However, while exhaustive identification of run-time errors is crucial from a reliability point of view, it's simply beyond the reach of conventional testing. PolySpace Verifier technology can and does provide exhaustive, automatic checking of all run-time errors. PolySpace Verifier relies on a breakthrough in Static Verification technology based on Abstract Interpretation.

A radical departure from existing testing and debugging tools that attempt to dynamically and iteratively verify software consistency, Abstract Interpretation enables PolySpace Verifier to exhaustively discover and check sections of code that are sources of run-time errors in a fraction of the time. As a result, it is the quickest way to achieve significant and immediate savings in the development process by eliminating the need to write test cases, instrument the code or execute the application.

**Static Runtime Error Detection**

Static Verification checks the dynamic properties of a software application without actually executing it, saving significant time. However, the difficulties of this approach grow exponentially with the size of the analyzed application. Abstract Interpretation, however, is a technique that provides an efficient solution for analyzing industrial-scale source code with low-cost computer resources.

By utilizing Abstract Interpretation, PolySpace Verifier works on an abstraction of the analyzed software and derived dynamic properties, instead of iteratively verifying software states. The cutting-edge technology in PolySpace Verifier provides a very high rate of selectivity (the proportion of the code that is diagnosed as safe, unsafe, or dead code) and significantly shortens the time required for exhaustive code reviews.

Now what does this really mean? It means that the PolySpace Verifier builds a virtual, abstract model of the application under analysis and propagates the domains of the application’s variables throughout the application. This creates a view of the application encompassing all possible executions, which is then used to identify run-time and non-determinism errors throughout the application.
The technological breakthrough lies in the scalability of the tool to industrial-sized applications. In order to generate the view encompassing all possible exact executions, a trade-off is made between precision in representing variable domains and program states, and in the analysis time. The Selectivity Rate is used to measure the level of success in identifying run-time and non-determinism errors. Until now, Abstract Interpretation techniques were limited to analyzing applications of a few hundred lines of code. With PolySpace Verifier, it is possible to analyze up to 150,000 lines of code, with a selectivity rate of 85 – 95%.

**JavaCard PolySpace Verifier**
We have chosen to develop a static analyzer for a sub-set of JAVA to meet the scheduling requirements. JavaCard is a sub-set of JAVA that allows smart cards and other memory-constrained devices to run applications (called applet) written in the Java programming language. Because of this memory constraint, the JavaCard platform supports only a customized subset of the features of the Java language. This choice can be explained by the tight schedule on one hand, and on the fact that GEMPLUS applications are JavaCard applets.

The second choice made in this project for the verifier, is to take as input not the Java source code, but the Java bytecode. This choice comes from the fact that Java is interpreted, but also the specific way of distribution of applications through the Internet where you can buy or find Java binaries (which are machine independent), instead of Java sources. In Java a program is normally stored as binary files representing compiled classes and interfaces. These binary files can be loaded into a Java virtual machine, linked to other classes and interfaces, and initialized. As a result, a program is composed from some binaries you get from the Internet (or any other way) without corresponding source code, and some binaries you obtain by compilation of your own application. The most general way to treat such an application is therefore to analyze the bytecode without requiring the source code. We will see later that if the source code is present, the interpretation of the results obtained by the verifier is simpler.

We have observed that for some large Ada or C applications the whole analyze of the program may take a few days. As the analyze is quite modular, we foresee that applying parallelization techniques at some level could certainly help diminish the execution time of the verifier. Such an improvement should be possible for the JavaCard verifier as well as for the Ada and C verifiers. We would want to develop techniques easy to implement for all the PolySpace Verifier tools.

### 3. Consortium

**PolySpace Technologies**
PolySpace Technologies is a software products company which has developed the first testing tool to apply Abstract Interpretation Techniques and to automatically detect run-time errors in programs at compilation time.

PolySpace’s mission is to provide breakthrough and efficient tools able to master the even more challenging issues of testing: testing bigger and more complex programs at lower costs while maintaining or even improving the quality level of the application analysed.
Run-time errors like overflows, arithmetical exceptions, out-of-bounds array accesses may be critical by their consequences on delivered SW products (products recall, loss of mission and finally impact on business as a whole) so that they have to be erased from software. But they are so difficult and costly to detect that today 30-40% of errors detected during production (for tested and released) are run-time errors.

Today PolySpace applies Abstract Interpretation Techniques to automatically and exhaustively detect these errors at compilation time in C and Ada 83 programs. As a result, PolySpace provides the unique technology to statically analyse the dynamics of software applications and to pinpoint errors prior to execution. Furthermore the PolySpace approach only requests source code to run. This means:

- No test cases to write.
- No execution or instrumentation of the code.
- No change to existing development process.
- An exact location of errors which does not require debugging time to be found.

At the end of the project, PolySpace will have proven that its technology can be extended to Object Oriented programming languages and will get a prototype of verifier dedicated to Java.

**Apache Research Project**

APACHE is a parallel processing research project, hosted in the ID (Informatics and Distribution) laboratory of the Grenoble Institute for Computer Science and Applied Mathematics (IMAG), France. It is sponsored by CNRS, INPG, INRIA and UJF.

This project investigates a new approach to program parallel machines for compute-intensive applications. Our goal is to find a good trade-off between performance and portability, so that programs with various profiles can be executed efficiently on parallel machines with different resources and facilities.

The programming environment Athapascan is an answer to these issues: efficiency and portability. A multithreaded communication runtime system, Athapascan-0, has been designed and has proven to be adequate and efficient. A programming library prototype, Athapascan-1, is available. It is based on a multi-tasking programming model with data consistency and grants an automatic load balancing. Test applications are being developed on Athapascan: molecular dynamics, quantum chemistry, computer algebra, domain decomposition and discrete event simulation. Finally, a tracing tool allows debugging, performance evaluation and visualization of Athapascan and its applications.

**Gemplus**

Gemplus is one of the founders of the smart card industry. Working with more customers in more countries than any other player, Gemplus helps its clients offer an exceptional range of portable, personalized solutions that bring security and convenience to our everyday lives.

Gemplus is the world's leading provider of solutions based on smart card technology. In fact, it is the only company dedicated to smart cards with a truly global reach. It has operations in 37 countries, the largest production capacity in the industry, and a presence in every major marketplace.

In response to key market drivers, Gemplus has two main business units:
- Telecommunications (incorporating Mobile Telecom and Public Telephony).
Financial & Security Services (incorporating Banking, Government & Large Enterprise, Retail and Electronic OEM).

In all cases, the Business Units support their clients by providing solutions which strengthen customer relationships, enhance security, and increase profitability in a converging world.

The success of Gemplus is founded on a track record of technological innovation, which has generated over 100 patents. It led the way with SIM cards, and is now setting the pace in open operating systems. In addition, it is working to make the online and wireless environment a safer, more convenient place to do business.

Gemplus breakthroughs often become benchmarks, since the company plays a key role in defining industry standards. From the pre-paid phone card to GSM and now Javacard, Gemplus technologies stand the test of time, and make it easier for clients to adapt and grow.

Gemplus maintains this technological advantage through its commitment to research, with the largest R&D team in the industry. Its engineers work in centers around the world, developing core competencies in areas like chip design, predictive Customer Relationship Management, application platforms, and Card Management Systems.

4. Specification of JavaCard PolySpace Verifier

JavaCard

The Java Programming language is a general-purpose, concurrent, class-based, object-oriented language. It is designed to maximize portability. Source code is compiled into Java bytecode (machine independent), which are designed to be run on the Java virtual machine (JVM).

JavaCard technology combines a subset of the Java programming language with a runtime environment optimized for smart cards and other memory-constrained devices.

JavaCard supports only small primitive data types (boolean, byte, shorts) and one-dimensional arrays. It does not support dynamic class loading, security manager, garbage collection and finalization, threads, object serialization and cloning. At this cost, applets can run on smart cards that are the smallest computing platforms in use today. The memory configuration of a smart card might have on the order of 1K of RAM, 16K of EEPROM, and 24K of ROM. This choice can be explained by the tight schedule on one hand and on the fact that GEMPLUS applications are Java Card applets.

JavaCard Runtime Errors

The objective of static testing approaches is to analyse the dynamic properties of software applications without executing them. This subject is well-known as to challenge computation time and exponential memory requirements issues as soon as the size of the source code analysed increases.

Abstract Interpretation is a breakthrough and efficient solution to analyse industrial-sized code on modestly equipped computers. The basic of Abstract Interpretation is to deal with an abstraction of the application analysed instead of enumeratively analysed its states.

JavaCard PolySpace Verifier automatically and exhaustively checks:

- Pointers De-referencing Issues (Null);
- Out-of-bounds Array Accesses;
- Invalid Arithmetic Operations (division by zero, square root of negative numbers…);
- Illegal type conversion (Impossible cast);
• Virtual Methods Invocation Issues (Null).

PolySpace Verifier automatically generates warnings for Non Termination of Call or Non Termination of Loop. Verifier generates only warnings here because in both cases it can be an error, or a programming feature. For Java, one more warning is implemented: an Exception Termination is computed. This warning only means that a method always terminates by throwing an exception, and will be refined in next versions of the tool.

PolySpace Verifier may also identify dynamically unreachable code, IE for which it does not exist any test case that may activate it. It is composed of the verifier that performs the analysis and generates some files to store the results, and the viewer that shows the results throughout an easy to use visualization interface.

JavaCard PolySpace Verifier
PolySpace Verifier is the server that makes the analysis. It relies on the Java bytecode and runs as a batch tool on Linux or Sun Solaris servers. The prototype of PolySpace JavaCard verifier has been developed as a Front-End to the common kernel already developed for the Ada and C verifiers.

During the project, the Front-End (specific to JavaCard) that translates the Java bytecode into the kernel internal representation (KIR) has been developed. This Front-End computes all the necessary information (call tree, flow graph, symbol tables, variable possible aliases, variable possible types…) and translates it into the KIR taken as input in the general kernel. It takes any Java program as inputs and rejects it if it is not a JavaCard applet (mini-application). In this way we have been able to validate the first steps of the Front-End on full Java programming Language.

After this first translation step, the general kernel common to the Ada and C dedicated verifiers is invoked to perform the analysis.

The processing phases that PolySpace Verifier achieves are:
Control and Data Flow Analysis:
- PolySpace builds the application call tree and the global data dictionary.

Software Safety Analysis:
- PolySpace checks each operation in the source code against all possible combinations of the input values of the program analyzed. The result is the color coded source code with associated diagnostics.

Requested parameters for this stage are:
- PolySpace Precision level.

The PolySpace Verifier color code is the following:
Green : Runtime error-free code section.
Red : A run-time will occur for any execution of the code section.
Gray : Dynamically unreachable code section. It does not exist any test case able to reach such code section.
Orange: Un-conclusive diagnostic that may be review depending on criticality criteria and development process stage. Some executions of such code sections may lead to a runtime error.

JavaCard PolySpace Viewer
The general viewer has been customized to deal with JavaCard specific checks, but also to show the checks with only line localization in the source. The second point is due to the fact that the bytecode contains no column localization but only line localization.

The viewer shows all the results of the analysis:

- **Call Tree**: One of the useful elements produced by the analysis is the bi-directional call tree for the application. Beyond the usefulness of the call tree as a result in itself, it is also a precious tool used for navigating within the application. The Call Trees may be generated as Excel spreadsheets.

- **Dictionary of Global Data**: This dictionary provides an overview of all global data elements in the application, together with their possible Java types. The global data dictionary may be generated as a Excel spreadsheet.

- **Run-Time Error**: The Run-Time Errors View displays the source code architecture (files and functions) for a quick location of errors in the source code. This view may be filtered to focus on a specific kind and severity of runtime errors. This view may be generated as an Excel spreadsheet.

- **Color coded source code**: The source code is shown as a colored text file in which each line is annotated with a colored symbol @. If it is clicked on, the colors of all the checks within the line are shown. If the source code is not available, the viewer shows no result. The Source Code View may be generated as a HTML document.

The Java bytecode contains no information on the column in the localization. Due to this lack of initial precision, the viewer shows all the checks at the beginning of the line in the source code view. This is not the case for the Ada and C tools, for which the source code is precisely colored (questionable operations and variables are colored).

### 5. Using PolySpace Verifier

**Methodology**

From the User’s perspective, the analysis of an application using PolySpace Verifier is nearly the same regardless to the target language. For JavaCard, there is not syntax correction to perform on the source to comply to the norm as the input is the Java bytecode, but there is a program entry point to write as an applet does not contain an entry point (static argument free method from which the whole program is analyzed).

As a result, the analysis of a JavaCard applet bytecode is composed of the following activities:

- **Writing the Program Entry Point**
  The JavaCard PolySpace Verifier requires an entry point, it must be a method declared as `public static void main ()`. It must simulate any possible execution of the applet: if you consider that the applet starts from a state and computes a new state by the generic method process, an infinite loop that invokes the method process on the applet object with a generic APDU (application protocol data unit) as argument simulates any sequence of transitions. Such a simulation is general, but simulates sequences of transitions that make no sense. Any constraint on the possible sequences implemented in the main will then result in a significant gain in precision of the analysis.

- **Initial Stubbying**
  The user can implement two levels of stubbing:
  Elementary-level stubs, which provide syntactical support for the analysis, and describe the effects of outside functions / procedures on global variables. These do not provide constraints on the input and output parameters of the outside functions / procedures.
Advanced stubs that reflect the constraints provided by the outside functions. These constraints can reflect software constraints, such as a division using a passed parameter (value must be non-zero), or may reflect real-world constraints.

- Setting Analysis Parameters
  The initial parameters for describing the application should be set. These parameters are of two categories:
  1) Those which describe the application characteristics which cannot be derived from the code alone.
  2) Those which tune PolySpace Verifier to get the right balance: selectivity vs. analysis time.

- Running the Analysis
  In this phase, the machine does all the work – the user simply checks from time to time to see whether or not the analysis is finished.
  NB: An analysis may encounter a serious (red) error in the application at any time, causing the analysis to halt (default mode). These must be fixed for the analysis to go on (see below). The analysis results obtained up to this point, including the red error(s) detected, are available.

- Fixing serious (red) errors
  For each internal phase executed by the tool, the rate of selectivity is increased. However, if the analysis encounters errors that are certain to occur (called ‘red’ errors), it cannot proceed to the next internal phase (default mode).
  For this reason, the user needs to fix the red errors found – other than NTC (Non-terminating Call) – then run the analysis again, to get the highest selectivity rate possible.
  Any modification to the source code, which is generally the case when fixing red errors, implies running the analysis again from the beginning.

- Increase the Precision through Better Modeling
  Based on the initial analysis results - indicating where precision needs to be increased - the user can improve the stubs to provide PolySpace Verifier with the real constraints on variable domains.

- Analyze Remaining Potential (orange) errors by hand
  Once all possible efforts have been made to get the selectivity rate as high as possible (again, this should be in the 80-90% range), then remaining orange warnings can be checked by hand.
  In this stage, the user may want to analyze only a subset of the oranges, depending on the criticality level of the application.

6. Application of the JavaCard PolySpace Verifier
We choose to apply the PolySpace Verifier on a case study which is too big for the current chip. If the tool can process the complete case study in a reasonable time, the scalability of the tool will match all the smart card applications. The Gemplus case study comes from the research lab leading to too major bias in the evaluation. First the development did not follow the regular software development process and second the software was already developed, and we applied the tool \textit{a posteriori}. Such a tool must be more profitable when applied during the development.
The case study is an electronic purse with personalization functionalities and some additional classes (utilitarian). Some of the API (Java Card) classes have been stubbed for the analysis, but for other we used the Java code.

**Main results**
The next table shows the different kind of errors found during the analysis.

<table>
<thead>
<tr>
<th></th>
<th>Red</th>
<th>Gray</th>
<th>Orange</th>
</tr>
</thead>
<tbody>
<tr>
<td>e-purse</td>
<td>13</td>
<td>330</td>
<td>387</td>
</tr>
<tr>
<td>utilitarian</td>
<td>5</td>
<td>42</td>
<td>23</td>
</tr>
<tr>
<td>API</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>19</strong></td>
<td><strong>374</strong></td>
<td><strong>413</strong></td>
</tr>
</tbody>
</table>

Once a red error is found, the remaining code is considered by verifier as unreachable and each unreachable errors are still considered as errors (they are not deleted). This is the reason for the huge number of gray errors detected. Grey errors really reveal dead code and are really significant when no red error is found; therefore they have to be analyzed after correction of red errors. In the same way, orange errors have only to be analyzed after elimination of red errors. In the forthcoming analyze of the result generated by verifier, we do not consider gray errors but only red ones.

**Analysis of Red errors**
The following table shows the exact meaning of the red errors found in the program.

<table>
<thead>
<tr>
<th></th>
<th>Class</th>
<th>Method</th>
<th>Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AllowedLoyaltyException</td>
<td>ThrowIt</td>
<td>ET</td>
<td>This exception is caught in admAddLoyaltyTable of the Purse class</td>
</tr>
<tr>
<td>2</td>
<td>LoyaltyTableException</td>
<td>ThrowIt</td>
<td>ET</td>
<td>Same as 1</td>
</tr>
<tr>
<td>3</td>
<td>Purse</td>
<td>AdmRegister</td>
<td>ET</td>
<td>Call of Throwit of the ISOException class.</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>AdmSetCreditAuthorizationKey</td>
<td>NTC</td>
<td>Explanation detailed later.</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>AdmSetCreditAuthorizationKey</td>
<td>ET</td>
<td>Related to error 4.</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>AdmSetState</td>
<td>ET</td>
<td>Same as 3.</td>
</tr>
<tr>
<td>7</td>
<td>PurseApplet</td>
<td>AdmRegister</td>
<td>ET</td>
<td>Call to the AdmRegister method that generates the red n°3</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>AdmSetExpirationDate</td>
<td>ET</td>
<td>In the catch clause, there is a call to ThrowIt of the APIsISOException.</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>AdmSetState</td>
<td>ET</td>
<td>Call to the AdmSetState method that generates the red n°4</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>AppCredit</td>
<td>ET</td>
<td>Same as 8</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>AppDebit</td>
<td>ET</td>
<td>Same as 8</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>ExchangeCurrency</td>
<td>ET</td>
<td>Same as 8</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>ThrowIt</td>
<td>ET</td>
<td>Same as 1</td>
</tr>
</tbody>
</table>

We denote ET the *Exception termination* warning and NTC the *Non termination of call*. 
In Java, the exception mechanism is as a regular control flow management technique. The ET warnings are therefore not to be considered as errors. They give a short-cut in the analysis of dead code: if a method always throw an exception it may signifies that part of this method is never executed because of a constant value in a test... After analysis, all the exceptions always thrown by the methods tagged ET, are associated to a handler in its callers.

**Analysis of the red NTC**

Here we analyze precisely the method `admSetCreditAuthorizationKey` in which the NTC is signaled. In this method, two lines are colored: one is signaled red by the tag `@` and the second is signaled gray by `@`.

```java
void admSetCreditAuthorizationKey(byte [] bArray, byte [] apduBuffer){
    verifyAccessCondition(apduBuffer, SYSTEM_SESSION,
            PurseApplet.ADM_SET_CREDIT_AUTHORIZATION_KEY );
    @ decryptKey(bArray, (short) 0, (byte) 16, creditAuthorizationKey);
    @ applet.admSetCreditAuthorizationKeyResp(SMKey);
}
```

Let consider first the line tagged `@`:

If one clicks on the symbol `@`, all the checks computed by verifier for this line are displayed. The only check computed here is an NTC: `non termination of call to decryptKey([BSBLcom/gemplus/pacap/utils/PacapKey)V`

This check is now analyzed:

- In the definition of the static method `decryptKey` no red error is signaled by verifier, and the other calls to `decryptKey` are not NTC.
- Inside the definition of `decryptKey`, an orange error is signaled on the access to its fourth argument (`key`).
- In the considered call to `decryptKey`, the fourth argument is `creditAuthorizationKey`.

If one searches for the declaration of the variable `creditAuthorizationKey`, one finds out that it is a private variable:

```java
private PacapKey creditAuthorizationKey = null;
```

The variable `creditAuthorizationKey` is never assigned a non null value. Inside method `decryptKey`, a method is invoked on this fourth argument `key`. An orange check signals that his parameter may contain a null value for some call. At the very call when this parameter is null, this virtual call will certainly fail. This explains the NTC warning indicated on the call to `decryptKey` and the red color of `@` of this line.

If this static call always fails, the following line tagged `@` within the definition of `admSetCreditAuthorizationKey` will never be executed. This explains the gray color of `@` of the next line.

As a conclusion, **JavaCard Verifier has found a fatal run time error** in this application.

The next step of the interpretation of the results given by verifier should be the analysis of gray and orange errors after correction of red errors. The analysis of gray checks has also revealed dead code and unused exception handlers. As for the orange errors, they have not been analyzed for this project. We have shown on some other application that the construction
of the main (infinite loop on the process method) influences greatly the number of orange checks. If the naïve implementation of the main is replaced by some precise one that forces only acceptable sequences of state of the applet, this numbers may be divided by three.

7. Parallelization of the JavaCard PolySpace Verifier

Parallelizing the application described above seems to be rather straightforward: it simply consists in verifying in parallel all the source files that compose the application. If one has a global storage area available to all the nodes of the parallel architecture then it seems simple to start all the verifications in parallel and wait for all these jobs to be over. Nevertheless, as stated earlier, an application to be verified is composed of files that must be handled in a given order to respect dependencies between them. For example, a header file must be treated before the files that depend on it. Respecting this order in parallel makes the parallelization trickier.

Our approach was to use the Athapascan runtime that provides ways to build tasks, express dependencies between them and execute them on a distributed architecture. To do this, a distributed dependencies graph is built. There is a complete independence between how the graph is distributed and where the tasks it describes will be executed. A load balancing protocol is used; its goal is to keep all the processors busy. But the Athapascan environment is aimed at parallel applications not at parallel execution of sequential jobs.

For this project we developed a shell that can be used to describe sequential jobs, typically a command to analyze a file, and dependencies between these jobs. This information is fed to a persistent daemon which is in charge of building the dependencies graph and of scheduling it. Communication between the shell and the daemon use an ORB in order to be able to do the shell part on a system separated from the one that will do the actual execution. The whole infrastructure is a tool called Fork Commander. We also developed a graphical tool to visualize the dependencies graph and modify it. It can also export it in different file formats: makefile, Athapascan shell...

All the tools used in this project are generic, not specific to the Java Verifier project. In particular it can be, and has already been, used to do a parallel make. The Fork Commander suite of tools is freely available and can be downloaded on the APACHE website.

8. Conclusion

The prototype of JavaCard PolySpace Verifier developed during the project has been validated by Gemplus on some applications. It is robust and scales well on the largest application taken as bench-mark by Gemplus. Analyzing the complete case study requires only a couple of hours which is acceptable for validation team, but may be too long for a development team. We only used the sequential prototype, we think that bigger applications can be analyzed using the parallel prototype.

Some improvements should be performed on the prototype by adapting the tool to the use of exceptions. For example, the tool could compare the declared throw clauses that declare the exceptions possibly thrown (directly or indirectly), if an undeclared exception may be thrown then the ET warning should become a red error of type Undeclared Thrown Exception. Another improvement could be to check if the exception is caught, only uncaught exception should lead to red checks of type Uncaught Exception. Such an improvement implies a lot of research developments. The Exception Termination warnings should be only generated on
user requirement as supplementary information, and should be refined to tell exactly what exception is or is not thrown.

In the context of the smart card where code is not observable (no debug means on the platform) any tool that tracks run time errors is very useful. We have shown that PolySpace Verifier satisfied this need. Moreover, on such a constrained device, sparing memory is very important. A precise analysis on gray errors has demonstrated several dead code areas, and also useless exception handlers. As a result the second advantage of using PolySpace Verifier is to optimize the JavaCard source code by eliminating the dead code.

The prototype of JavaCard Verifier developed during this project has been presented at the DAEDALUS European project on abstract interpretation in September 2002. After the end of the project, this initial JavaCard prototype has been partially extended to Java, and the resulting prototype of Java PolySpace Verifier is under evaluation at NASA. Due to a significant pause in the Java programming language propagation, the transformation of the prototypes into a product is still being studied from a commercial point of view. To go from the prototype to a product, a long phase of testing and generalization to Java is still to be performed. Due to the specificity of the Java verifier, the robustness of the Front-End should be guaranteed by a Java bytecode checker to test for the binary compatibility between all the binaries compiled separately. Java development tools generally support automatic recompilation as necessary when source code is available. This insures that the applications constructed using such development tools are coherent if the source code is available. If it is not, there is no way to automatically check for this compatibility. If the existing Java verifier is given incompatible binaries, the analysis will stop at some point for sure, but the user will not get enough information to find the origin of the incompatibility. Such an extension would be a valuable extension of the prototype to get a robust product.

The techniques and tools developed for the parallelization of the Java Verifier are general enough to be applied to the parallelization of other targets. They have shown their applicability, and have already been used to parallelize other makefiles. In the future, we want to use the Athapascan and Fork Commander infrastructure to create a multi-level scheduler that can use information both inside the applications and between them. The idea is to detect idle periods on processors by using the intra-application information and fill these gaps with pending jobs. To do this we needed a batch system that used the Athapascan technology, this is what Fork Commander provides.