# A Declarative Framework for Adaptable Applications in Heterogeneous Environments

P.Inverardi, F.Mancinelli, M.Nesi
Dipartimento di Informatica
Via Vetoio, 1
67010 L'Aquila - ITALY
{inverard,mancinel,monica}@di.univaq.it

## ABSTRACT

In this paper we present an approach for developing adaptable software applications. The problem we are facing is that of a (possibly mobile) user who wants to download and execute an application from a remote server. The user's hosting device can be of different kinds (laptops, personal digital assistants, cellular phones, communicators, etc.) with specific hardware and software capabilities. The problem is to be able to decide whether the user's current device characteristics are compatible with the application requirements in order to prevent execution failures. In the negative case we want to identify the reasons that determined the incompatibility and perform an automatic adaptation of the application, so that it can match the user's device capabilities. We adopt a declarative approach: we provide each device with a declarative description of its characteristics and, possibly, context constraints. Inspired by Proof Carrying Code (PCC), we use first-order logic formulae to model both the behavior of the code, with respect to the properties of interest, and the execution context. The adaptation process is carried out by using theorem proving techniques, in particular, the proof assistant HOL4. The aim is to derive a formal proof which asserts that the behavior of the code can be correctly adapted to the given context. By construction, the proof, if it exists, gives information on how the adaptation has to be done. On the application side, Java2 Micro Edition (J2ME) is the chosen reference application development environment.

## 1. INTRODUCTION

The current panorama of communication infrastructures lets us foresee that several kinds of integrated transmission and communication infrastructures will be available in the near future. In such a scenario there will be a shift from the classical desktop-centric computing paradigm to a more dynamic one. Applications will run, above all, on mobile devices (laptops, personal digital assistants, cellular phones, communicators, etc.) and the communication infrastructure will enable communication among each other and with the surrounding environment. In such a setting applications must be aware of two kinds of heterogeneity. The first one relates to the computational environment offered by the context in which they are running: think, for example, of the bandwidth offered by the communication medium. The second one concerns the device itself: the same communication infrastructure can be accessed by a large variety of devices which are all homogeneous in terms of basic resources and functionalities, but different with respect to quantitative and qualitative characteristics (e.g., memory size, computational power, display capabilities, supported protocols, etc.). Moreover, considering that in the next future computing capabilities will be exploited in many non-conventional domains (e.g. home appliances, wearable computers), we expect this heterogeneity to grow more and more. What today is seen as a discrete set of well-characterized different types of devices, tomorrow will become a virtually infinite range of heterogeneous devices, each one with its own set of capabilities. This means that applications must be aware of this potentially infinite variability in order to prevent execution failures.

Our research addresses the problem of dealing with the second kind of heterogeneity, with a focus on the application side. We aim at developing applications which are *generic* and can be correctly adapted with respect to a dynamically provided context. We chose to attack this problem by using a declarative and deductive approach, that enables the construction of a generic and adaptable application code and its *correct* adaptation. Inspired by Proof Carrying Code (PCC) [7] techniques, we use first-order logic formulae to model the code behavior with respect to the properties of interest, and to characterize the execution context. The adaptation process is carried out by using theorem proving techniques in order to formally prove that the code behaviour can be *correctly* adapted to the given context. Provided that such a proof exists, by construction it gives information on how the adaptation has to be done. We believe that the use of a deductive approach in this context enables us to flexibly and efficiently reason about the adaptation and to cope with the increasing growth of highly heterogeneous contexts. The practical setting in which we are experimenting our ideas is Java2 Micro Edition (J2ME) [10] for the application side, and the proof assistant HOL4 [4] (from now on simply referred to as HOL) for the declarative and reasoning side.

In the following sections we set the context of our work, by briefly introducing the technologies we refer to (namely J2ME, HOL and PCC), and provide a description of our framework by means of a working example. Section 7 discusses related work, and Section 8 summarizes our contribution and outlines future work directions.

## 2. SETTING THE CONTEXT

We are considering the context in which a (possibly) mobile device requests an application. Since it is not possible, for the requested application, to have *a priori* information about the characteristics of the runtime environment, the application code may execute incorrectly or even fail. In order to avoid failures, the application must be able to adjust its behavior, according to the runtime environment in which it will be deployed. Applications with this ability are called *adaptable* applications. The future home appliances scenario will provide a fertile ground for this kind of applications [1]. Imagine, for example, a microwave oven which is able to download food recipes and use them to cook some food. In such a scenario a recipe would be a mobile code which contains all the instructions to cook the food. Even though many microwave ovens offer the same functionalities, there might be some differences among them. For example, the cooking power could be different from one model to another. A recipe must take care of those differences in order to, for example, correctly adjust the cooking time with respect to the available cooking power. Some recipes might also fail on some ovens because of the lack of a grilling facility, and so on.

In such a context, we consider two different types of adaptable applications:

- *Self-contained*: applications that embody the adaptation logic as a part of the application itself.
- *Tailored*: applications that are the result of an adaptation process, which has been previously applied on a generic version of the application.

Self-contained adaptable applications consist of tangled application and adaptation code. The former provides the implementation of the application functionalities, the latter implements the adaptation logic. Tailored applications, instead, do not contain any adaptation logic, and appear to be programmed specifically for the environment in which they are executed. They are the result of an adaptation process that, starting from a generic version of the application code, produces the final customized application. There are pros and cons in using either kinds of adaptable applications. Self-contained adaptable applications are inherently dynamic in their nature. Since the code which performs the adaptation is embedded in the application itself, it can handle dynamic changes in the environment by reacting to them at runtime. The pay-off is the inevitable overhead imposed by the adaptation code. Thinking of the microwave oven, a self-contained adaptable recipe should contain the logic to handle all the possible (expected) oven characteristics. This overhead could be a problem since an oven is, after all, a limited device. Moreover, the adaptation logic in a self-contained adaptable application does not scale well when considering multiple, possibly interfering, characteristics. In fact, a typical adaptation logic would consist of isolated and locally enabled switch statements which provide alternatives that might be used to handle changes in the environment. Considering those statements globally would require combinatorial conditions to be provided and, consequently, increased code complexity. On the contrary, tailored adaptable programs are the result of an adaptation process which produces a customized application for the target execution environment. Since the process which produces such applications is external to the application itself, it might allow a very effective way of reasoning on how the application should be adapted. However, tailored adapted applications are dynamic only with respect to the deployment environment. They are static with respect to the actual execution, i.e. they cannot adapt to runtime changes in the execution environment. Even though our approach is inherently static, we may think that the generic application may contemplate an adaptation which provides some dynamic adaptation logic, which is able to recognise changed conditions in the execution environment and trigger further adaptations.

Our approach deals with tailored adaptable applications and exploits deductive methods to find the right way to globally adapt an application. The aim is to derive a formal proof that the behavior of the code can be correctly adapted to the given context. By construction, the proof, if it exists, gives information on how the adaptation has to be done.

For experimentation purposes, we focus on a particular class of mobile devices available nowadays, namely mobile phones. Therefore, since the Java2 Micro Edition (J2ME) platform powers many of these devices, we choose it as the reference platform. The adaptation process is carried out in the proof assistant HOL. All these components have been integrated together in a well-defined architecture for a framework which supports the whole approach.

### 2.1 Java2 Micro Edition

Java2 Micro Edition (J2ME) [11] is the Java platform for consumer and embedded devices, such as personal digital assistants (PDAs), mobile phones and a broad range of embedded devices. The J2ME architecture is a layered architecture which comprises the following elements: a combination of a Java Virtual Machine; a minimal set of class libraries called *configurations*; a set of additional APIs, called *profiles*, which are stacked over a configuration, and characterizes a broad range of devices with peculiar characteristics. In our experimentations, we concentrated on the Connected Limited Device Configuration (CLDC) [12] which provides a reduced version of the Java Virtual Machine, called Kilo Virtual Machine (KVM) [10]. Moreover we use the Mobile Information Device Profile (MIDP) [8] which characterizes many mobile devices, such as mobile phones and PDAs.

### 2.2 The proof assistant HOL

The proof assistant HOL [4] is based on the LCF methodology for interactive and secure theorem proving by mechanising its logic in the strongly-typed language Moscow ML, a light-weight implementation of Standard ML. The HOL logic can be extended in a consistent way by means of derived rules of definition, such as concrete data type definitions, recursive function definitions and inductive definitions. When using these derived rules, the system performs all the formal inference necessary to define data types, recursive functions and inductive relations in higher order logic, and automatically derives an abstract characterization of data types and relations, including standard theorems such as induction and case analysis theorems.

Moscow ML is used to prove that certain terms are theorems. A theorem $\Gamma \vdash t$ is represented by a finite set $\Gamma$ of terms called *assumptions* and a term $t$ called *conclusion*. Theorems must either be postulated as axioms or derived by formal proof. Besides forward proofs, the HOL system supports goal directed (or backward) proofs. The idea is to start from the desired result (*goal*) and manipulate it using functions called *tactics*, until it is reduced to a goal which is true. When a theorem is proved, it can be stored using several functions.

### 2.3 Proof-Carrying Code

Proof-Carrying Code (PCC) [7] is an approach that aims at checking code behavior against *safety policies*, which define properties and constraints the code should respect. This approach is based on theorem proving techniques and makes use of a Verification Condition Generator (VCGen), which takes as input an annotated version of the code, and outputs a *safety predicate* that models the expected code behavior, using first-order logic formulae. The VCGen is derived from an abstract machine that defines the constrained execution environment, and from a set of *safety rules*, which include both classical logic rules and domain specific rules. The whole approach can be summarized in the following steps: a code producer generates the application code and annotates it; a safety predicate is generated from the application code, using the code consumer VCGen; by using the safety rules, the code producer generates a proof, which validates the safety predicate and asserts that the code respects the code consumer constraints; the proof is sent back to the code consumer, together with the native code, in a bundle called *binary PCC*. The consumer is then able to check it using a proof checker and can, therefore, execute the delivered application code safely.

## 3. THE FRAMEWORK REFERENCE ARCHITECTURE

Our approach focuses on tailored adaptable applications, where the adaptation logic is outside the application, and the final code is the result of an adaptation process. The approach is supported by a framework which enables the development and the distribution of adaptable applications. Figure 1 shows the framework architecture, which is composed of the following components:

- *Development environment*: this is the environment where the developer builds the actual adaptable application. It consists of a set of tools, including a specialized compiler which is able to deal with the generic nature of the application.
- *Mobile code*: this is the object code of the adaptable application. It is produced by the tools in the development environment, and is an annotated code containing all the information useful for the adaptation process.
- *Declarative description*: this is a first-order formula which characterizes the mobile code with respect to some properties of interest. It is extracted starting from the mobile code and provides a declarative description of the application behavior with respect to the properties of interest.

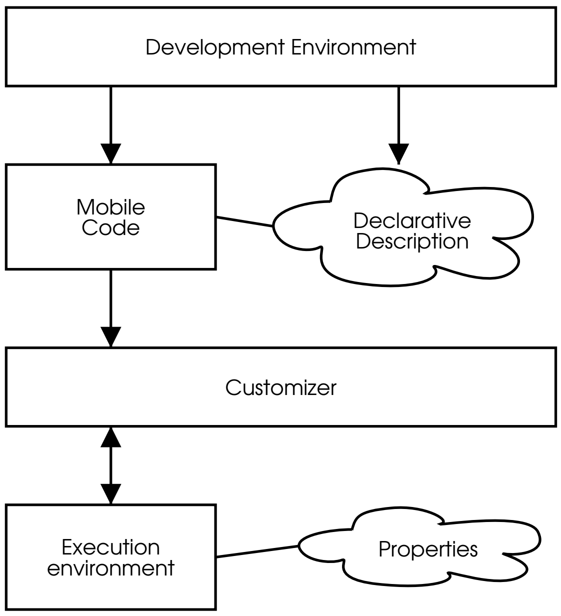


Figure 1: The reference architecture

- *Customizer*: this component performs the tailoring of the mobile code with respect to the execution environment. The customizer embodies all the deductive logic needed to explore the space of possible adaptations and to find a suitable version of the application, if any. The customizer is also able to assemble the final application code, ready to be deployed.
- *Execution environment*: this is the physical device which requests the adaptable application and provides the runtime support.
- *Properties*: this is a declarative description of the properties characterizing the execution environment (in terms of device capabilities). The customizer uses this information, together with the description of the adaptable program, to reason about the adaptation.

The architecture depicted in Figure 1 does not suggest any localization of these components on an actual architecture. Depending on the application domain, the components may be deployed in different ways. For example, a typical client-server deployment scenario would deploy the execution environment and its properties on the client side, while the customizer, the mobile code, its declarative description and the development environment would be installed on the server.

## 4. THE EXPERIMENTATION CONTEXT

We experiment our approach using limited mobile devices, such as mobile phones, where the runtime environment is that of J2ME. The reference architecture is deployed on a client-server architecture as shown in Figure 2.

Our approach spans all the application lifecycle, from development to deployment. We start from a generic application which allows a suitable adaptation according to the execution environment. The way this adaptation might take place is specified by an *adaptation policy*, which is specified through one or more of the following elements:

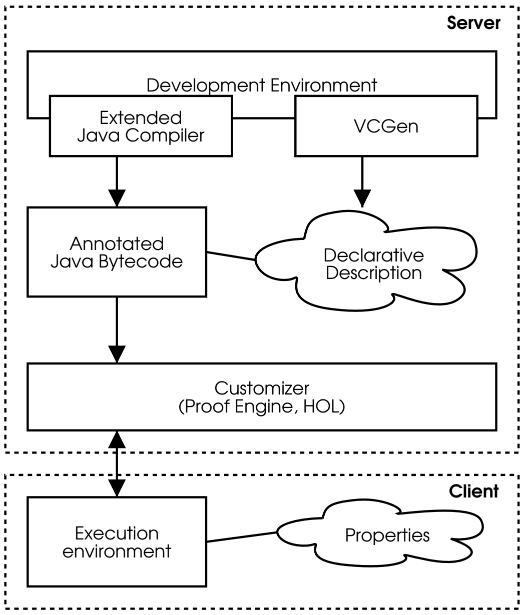


Figure 2: The framework architecture

- *Adaptation points*: well-defined source-level code places where program adaptation may occur.
- *Adaptation alternatives*: for every adaptation point a set of alternatives that provides the actual ways in which the program can be adapted.

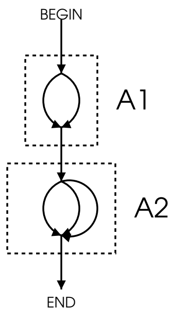


Figure 3: An adaptable program

Figure 3 shows a graphical representation of an adaptable program whose adaptation policy is defined by two adaptation points *A1* and *A2*, with two and three adaptation alternatives, respectively.

The adaptation policy is specified at the source level by extending the source programming language with special syntactical constructs. In our case we have extended Java with an `adapt` statement whose syntax is shown in Figure 4.

Each `adapt` statement defines an adaptation point, and each $c_i$ specifies the piece of code of the adaptation alternative.

```
adapt{c₁}
use{c₂}
...
use{cₙ}
```
Figure 4: The `adapt` construct

The application development will be supported by ad-hoc tools, that compile the extended Java language to an extended Java bytecode, which is relocable and annotated. Relocation information is useful to the customizer for the tailoring phase, when the generic application has to be assembled in an executable one. Annotations are necessary in order to reflect the adaptation policy, down to the bytecode level. Annotations also express properties of the code, like loop invariants, which allow a declarative description of the program to be produced.

Once the annotated bytecode has been produced, it is processed in order to obtain its declarative description. Inspired by Proof Carrying Code, we chose to use a Floyd style Verification Condition Generator (VCGen) for this step [3]. Starting from the operational semantics of the Java Kilo Virtual Machine and from a formalization of the device capabilities, we specified the VCGen which takes the bytecode as input and produces a first-order logic formula as output. This formula is called *adaptation predicate* and represents an abstraction of the application behavior with respect to the properties of interest. Typical properties which can be considered in this setting are: power consumption, display capabilities (screen size, number of colors), CPU power, supported protocols and so on.

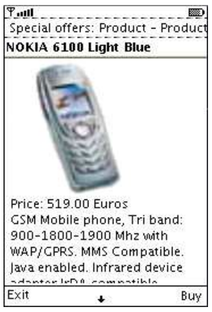
The capabilities characterizing the execution environment are then used as *preconditions* for the adaptation predicate. They are formalized as a set of predicates plus a set of simplification axioms. By using theorem proving techniques, we try to derive a proof of the following formula:

$$\textit{Capabilities} \Rightarrow \textit{Adaptation Predicate}$$

A proof system, which comprises classical first-order rules and property specific rules, is used in order to find a proof for the previous formula. The adaptation predicate validity proof, if it exists, gives two kinds of information: on the one hand it states that the program can correctly run on the target device, on the other hand it gives information on how the program should be adapted. The adaptation predicate, in fact, embeds all the information about the adaptation policy by means of special indexed *OR* clauses, corresponding to the adaptation points. Proving the formula *Capabilities* ⇒ *Adaptation Predicate* implies selecting correct alternatives for each adaptation point. In the next section, all the outlined steps will be described in the scope of the example.

## 5. THE FRAMEWORK AT WORK

To illustrate our approach we built a small and simple application for mobile devices called *Mobile Product Information* (MPI). This application can be used in large malls, where wireless networking is exploited. People with mobile devices may receive on their terminals information about the product they are currently looking at. The information can be of any kind: from textual descriptions to detailed images and animations, and so on. Advanced functionalities, such as buying by clicking, can also be supported by the application. In Figure 5(a) we show the "ideal" application output for somebody who is interested in buying a specific mobile phone. This output is "ideal" because different J2ME compatible devices might run this same application in different and undesirable ways (Figure 5(b) and 5(c)). With the current technology, in order to avoid these misbehaviors, we must have different versions of the application for each compatible device. Instead, in our approach we produce a generic code to be customized at the deployment time, that is when the user gets in the mall with her/his own device.

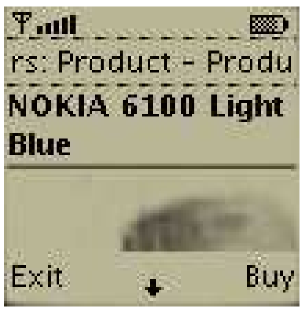

Special offers: Product - Product

NOKIA 6100 Light Blue

Price: 519.00 Euros
GSM Mobile phone, Tri band:
900-1800-1900 Mhz with
WAP/GPRS. MMS Compatible.
Java enabled. Infrared device

Exit ↓ Buy

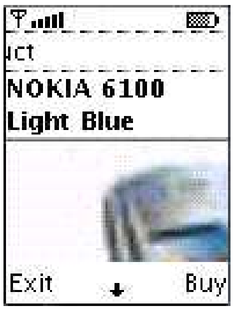
(a)


rs: Product - Produ

NOKIA 6100 Light Blue

Exit ↓ Buy

(b)


ct

NOKIA 6100 Light Blue

Exit ↓ Buy

(c)

Figure 5: Ideal and actual execution on different devices

An excerpt of the main code for the MPI application is shown in Figure 6. There are two adaptation points at line 2 and line 14. The first one defines three alternatives, each one concerning a different bitmap image of the product to be displayed. The second adaptation point defines two alternatives. The first one concerns a scrolling ticker, the second one is an empty alternative which states that it is correct not to display any ticker. The fixed part of the program (lines 19-21) consists of the textual description of the product.

Applying the VCGen to the previous code results in the following adaptation predicate:

$$
\begin{array}{ll}
OR_1( & Visible(RECT(0,0,180,180)) \wedge \\
 & Available(POWER(HIGH)), \\
 & Visible(RECT(0,0,64,64)) \wedge \\
 & Available(POWER(HIGH)), \\
 & Visible(RECT(0,0,64,64)) \wedge \\
 & Available(POWER(MEDIUM))) \\
\wedge & \\
OR_2( & Available(POWER(HIGH)), \\
 & true) \\
\wedge & \\
 & Available(POWER(LOW))
\end{array}
$$

*RECT* is a function which defines the area, say a rectangle, needed to display something. *Visible* is a predicate which is true if its argument can be entirely shown on the display. Similarly *POWER* is a function which expresses the power consumption of the operations in the program. *Available* is a predicate which is true if the amount of available power level (LOW, MEDIUM and HIGH) in the execution environment satisfies the power indicated in its argument.

Each adaptation point is reflected in the adaptation predicate by using an indexed *OR* predicate. The first *OR* refers to the first adaptation point and to its three alternatives, that have different display and power requirements. The second *OR* refers to the second adaptation point whose first alternative (the ticker) requires a high power level, while the second alternative is empty and is formalized with *true*. The last *Available* predicate refers to the textual description which does not have any display requirements (the J2ME display API is able to automatically layout the text inside the display boundaries) and has a low power consumption.

Whenever a client requires the application by providing its capabilities, this adaptation predicate is used to prove whether or not there exists a correct configuration matching the client's characteristics. In the next section we show how the proof and configuration steps are carried out in HOL.

## 6. PROVING THE ADAPTATION PREDICATE IN HOL

The first step in the mechanization of the proof system is the definition of the types for rectangles, power levels and power. Using the definition mechanism for data types in HOL, the types are as follows:

```
Rectangle = RECT of num => num => num => num
Level = LOW | MEDIUM | HIGH
Power = POWER of Level
```

A binary predicate `pLess: Level -> Level -> bool` is defined to assert the following ordering on levels:

```
|- pLess LOW MEDIUM /\
   pLess LOW HIGH /\
   pLess MEDIUM HIGH
```

The screen property is formalized in HOL by defining the predicate `Visible` on rectangles in an inductive way. The rules are as follows:

```
Vbase = |- Visible (RECT 0 0 0 0)
Vind = |- !w x y z.
      (?x' y' z' w'.
         Visible (RECT x' y' z' w') /\
         x' <= x /\ z <= z' /\ y' <= y /\ w <= w')
       ==>
       Visible (RECT x y z w)
```

The base rule asserts that the screen origin is visible, and the induction rule states that a rectangle is visible whenever it is included in some visible rectangle.

The predicate `Available: Power -> bool` is formalized as a function that has the following property:

```
|- !x y.
    Available (POWER x) /\ pLess y x ==>
    Available (POWER y)
```

This asserts that, given any available power level $x$, any smaller (with respect to the level ordering) power level $y$

```
Form form = new Form("NOKIA␣6100␣Light␣Blue");
ADAPT {
  Image image = Image.createImage("/images/large.png");
  form.append(image);
}
USE {
  Image image = Image.createImage("/images/small.png");
  form.append(image);
}
USE {
 Image image = Image.createImage("/images/bw.png");
 form.append(image);
}
ADAPT {
Ticker ticker = new Ticker("Special␣offers:␣...");
form.setTicker(ticker);
}
USE {}
form.append("\nPrice:␣519.00␣Euros\n");
form.append("\n————————————\n");
form.append("\n...");
```

Figure 6: MPI generic Java code

is also available. By instantiating on the level values and using the definition of the predicate `pLess`, the following theorem `power_deriv_thm` is obtained that is useful when proving properties involving the predicate `Available`:

```
|- (Available (POWER MEDIUM) ==>
    Available (POWER LOW)) /\
   (Available (POWER HIGH) ==>
    Available (POWER MEDIUM)) /\
   (Available (POWER HIGH) ==>
    Available (POWER LOW))
```

Finally, the indexed predicate $OR$ in the adaptation predicate generated by the VCGen is simply defined as a recursive function on lists of predicates with the extra argument of the adaptation index, that will be useful to derive the adapted code:

```
OR_def = |- (!n. OR n [] = F) /\
            !n f l. OR n (f::l) = f \/ OR n l
```

Proving adaptation predicates containing the predicate `Visible` usually involves induction proofs. A simple theorem tactic can be applied when reasoning about `Visible`:

```
fun rtacF thm =
   RULE_TAC Vind
   THEN ASM_EXISTS_TAC thm
   THEN DECIDE_TAC;
```

where `ASM_EXISTS_TAC` is a theorem tactic that makes use of the supplied theorem, typically an assumption (in this case, the precondition concerning the screen capability of the execution environment), to reduce existentially quantified goals. The following tactic is able to prove adaptation predicates involving only the property `Visible`:

```
fun vtac thm =
   (ARW_TAC [OR_def] THEN
    REPEAT ((DISJ1_TAC THEN rtacF thm)
            ORELSE DISJ2_TAC)
    THEN rtacF thm)
   ORELSE FAIL_TAC "no alternatives possible";
```

where `ARW_TAC` is a simplification tactic for standard arithmetics. The tactic `vtac` fails whenever there is no combination of alternatives that is true.

Adaptation predicates involving both `Visible` and `Available` can be proved by means of an extended version of the tactic `vtac` that suitably enriches the assumptions about the availability of power:

```
fun pvtac thm = IMP_RES_TAC power_deriv_thm
                THEN vtac thm;
```

Thus, given the following goal formalizing in HOL the adaptation predicate for the MPI application, with the preconditions of a visible screen of dimensions 100×100 and of the availability of a high power level,

```
- set_goal([--`Visible(RECT 0 0 100 100)`--,
            --`Available(POWER HIGH)`--],
           --`OR 1
              [Visible(RECT 0 0 180 180) /\
               Available(POWER HIGH);
               Visible(RECT 0 0 64 64) /\
               Available(POWER HIGH);
               Visible(RECT 0 0 64 64) /\
               Available(POWER MEDIUM)]
              /\
              OR 2 [Available(POWER HIGH); T]
              /\
              Available(POWER LOW)`--);
```

and by applying `pvtac` with the assumption on the screen visibility to solve the `Visible` predicates,

```
- e (pvtac
     (ASSUME (--`Visible(RECT 0 0 100 100)`--)));
```

the adaptation predicate is proved:

```
OK..
> val it =
    Initial goal proved.
```

If the available power level is low, then there is no combination of alternatives that is true, as the first $OR$ is false, and the application of `pvtac` fails.

From the proof of the adaptation predicate is then easy to extract the information needed to perform the actual application tailoring. The tactics shown above for proving adaptation predicates, with respect to the constraints on the screen visibility and power availability of the execution environment, simply check if there exists at least one combination of alternatives of the various adaptation points that is true. For the configuration step, based on these tactics, we have implemented a function that, given an adaptation predicate, returns the set of all combinations of alternatives that are true. This set is the basis for developing various configuration strategies that can dynamically reflect the client's preferences. For example, a client, although characterized by the same preconditions, in different moments can choose different configuration strategies.

## 7. RELATED WORK

Our framework architecture shares many elements with PCC [7]. However, our approach differs from it in the following aspects. First of all, PCC focuses on security aspects of code mobility while we concentrate on adaptation issues. In our approach, the proof is not used to guarantee, on the client side, that the application is correct with respect to some safety policy but is used, on the server side, to assemble a correct application with respect to the client's capabilities. Checking the conformance of the received application according to the proof is beyond the aims of our approach. PCC does not put any constraints on the proof system, its main concern is correctness since the whole approach is based on the assumption that a proof can be built. Our setting is more demanding since we use the proof system to establish the correctness of the adaptation process. PCC is entirely based on the machine code while we apply our process on Java bytecode.

A different approach to adaptability of software applications is presented in [6, 5]. These papers propose an approach for Dynamic Software Update which uses verifiable native code, such as PCC or TAL, to deliver correct patches which could be applied at runtime to software with availability requirements. The approach requires the code to be written so that it can be dynamically updated. It deals with dynamic linking in order to patch executable code on the fly and with code verification in order to assure that the received patch is correct with respect to some safety properties. This approach combines PCC-like techniques with dynamic linking, thus the adaptation is carried out at runtime. The focus there is on correct dynamic linking rather than on correct tailoring of generic applications.

Aspect Oriented Programming (AOP) techniques may be considered with respect to adaptation issues. In particular approaches like PROSE [9] enables dynamic adaptation of Java programs, specified using aspect oriented mechanisms. While with our approach we may surely benefit from AOP for specifying adaptation alternatives and dynamic AOP to carry out the adaptation process, we go further by providing a formal framework in which reason about program properties and correct adaptation with respect to these properties.

It is also interesting to mention a new research project called Resource Aware Programming (http://www.cs.rice.edu/~taha/RAP/) that focuses on functional programming. In particular, its authors' aim is "to strengthen traditional multi-stage type systems using (mainly) foundational techniques from type theory and functional reactive programming (FRP) to create a paradigm of resource-aware multi-stage programming".

## 8. CONCLUSIONS

In this paper we have presented a declarative framework to tailor applications for mobile devices. The approach makes use of theorem proving techniques to support the formal reasoning on the adaptation process and of J2ME as the application development environment. A basic assumption underlying our approach is that the device heterogeneity will increasingly grow in the next future, thus introducing unknown requirements on the application programmer side. We believe that more and more often application programmers will produce code in absence of well-established standards for the characteristics of the target device. This situation demands new ways of producing and deploying application software. Our belief is that a declarative approach like the one we propose can be very effective. In our approach the application programmer is only concerned about "local" adaptation constraints. The whole burden of the global adaptation logic is managed outside the application and can dynamically fit the client's needs.

A theorem prover like HOL provides the possibility of programming tools for the analysis and verification of properties in a very flexible and modular manner [2]. Its higher order capabilities are necessary in order to allow the definition of expressive configuration policies based on dependencies and/or interferences among properties.

Future work concerns extending the VCGen beyond the core semantics of KVM, refining the proof system for a larger set of properties, providing more functionalities to support the proof and the configuration processes, and realizing a complete set of tools for the development of adaptable applications.

## 9. REFERENCES

[1] N. Amano and T. Watanabe. A Software Model for Flexible and Safe Adaptation of Mobile Code Programs. In *Proceedings of IWPSE*, pages 57–61. ACM Press, 2002.

[2] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The PROSPER Toolkit. In *Proceedings of TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 78–92. Springer-Verlag, 2002.

[3] R. W. Floyd. Assigning Meaning to Programs. In *American Mathematics Society Symposia in Applied Mathematics*, volume 19, 1967.

[4] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[5] M. Hicks, S. Weirich, and K. Crary. Safe and Flexible Dynamic Linking of Native Code. In *Proceedings of TIC 2000*, volume 2071 of *Lecture Notes in Computer Science*, pages 147–176. Springer-Verlag, 2001.

[6] M. W. Hicks, J. T. Moore, and S. Nettles. Dynamic Software Updating. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23, 2001.

[7] G. C. Necula and P. Lee. Proof-Carrying Code. Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, Sept. 1996.

[8] J. E. V. Peursem. *Mobile Information Device Profile*. SUN Microsystems, 2002.

[9] PROSE, On Dynamic AOP. http://prose.ethz.ch/Wiki.jsp?page=OnDynamicAOP.

[10] SUN Microsystems. *Java2 Platform Micro Edition Technology for Creating Mobile Devices*. SUN Microsystems, 2000.

[11] SUN Microsystems. *Java2 Platform, Micro Edition*. SUN Microsystems, 2003.

[12] A. Taivalsaari. *Connected Limited Device Specification (Version 1.1)*. SUN Microsystems, 2003.