Annales Univ. Sci. Budapest., Sect. Comp. 30 (2009) 175-190

COMMANDING A ROBOT IN A SAFE WAY[1]

Z. Istenes and T. Kozsik

(Budapest, Hungary)

Abstract. Certified Proved-Property-Carrying Code is a technique for efficiently ensuring correctness of mobile code. It relies on a trusted certifying authority that statically verifies correctness proofs and certifies correctness of program components passed as mobile code from a code producer to a code receiver. This paper uses an implementation of CPPCC to enable the safe dynamic reconfiguration or reprogramming of a robot. In this implementation the B-method is used to construct programs together with machine-verifiable proofs of correctness.

1. Introduction

Mobile code technologies make it possible to extend a running application (a "code receiver") with components, e.g. by downloading a piece of code through a network and dynamically linking it to the application. Such technologies are extremely vulnerable against malicious code, as well as against accidentally erroneous or improper code. There are a number of solutions to increase trust in mobile code. A widely used solution is to attach certificates to the mobile code (e.g. to a web-browser plug-in or to a Java applet). Such a certificate may provide a guarantee that the mobile code originates from a certain code provider, hence the risk of executing malicious code can be minimalized. This solution, however, gives no protection against accidentally erroneous or improper mobile code.

[1]Supported by ELTE IKKK (GVOP-3.2.2.-2004-07-0005/3.0) and Stiftung Aktion Österreich–Ungarn (66öu2).

Another popular solution is the use of proof-carrying code [1]. Proof-carrying code (PCC) is a piece of mobile code with attached properties and proofs. The receiver of the proof-carrying code can verify the attached proofs, and – by investigating the attached properties – it can decide whether to use or to refuse the received mobile component. This approach also has some drawbacks. Due to the proofs, the size of the mobile components may become significantly larger, and the verification of the proofs may significantly slow down the code receiver application. It is even possible that the code receiver does not have the necessary resources (memory, network bandwidth, CPU-time, proof-checking software) for verifying the proofs. The successful applications of PCC target simple safety properties of low-level code (e.g. JVM bytecode). Functional correctness needs to be expressed at a higher level of abstraction, at source-code level (e.g. on Java code) and typically requires large proofs.

Certified Proved-Property-Carrying Code [2] (or CPPCC, for short) is a technique that aims at eliminating the drawbacks of the above two approaches by combining certificates and PCC and by splitting the verification process into two phases. Using this technique the producer of a mobile component proves the correctness of the source code of the mobile component (in the presented case study the source code is expressed as a set of B abstract machines and B implementations). A trusted third-party certifier authority checks the proofs – this is the first phase of the verification process – and signs the mobile component. The code receiver application checks the signature and whether the specification of the received mobile code (which is received as a part of the mobile component) corresponds to the requirements of the code receiver – this is the second phase of the verification process.

As illustrated in this paper, the CPPCC approach can be used to dynamically modify the behaviour (the software) of a robot in a safe way. This makes it possible to reconfigure or reprogram the robot at run-time, or to have the robot complete missions expressed as mobile components (see Figure 1). One can develop a robot controlling component using e.g. the B-method, and use the CPPCC framework to transmit this component to the robot as a piece of mobile code. The robot software will execute the received component only if the component satisfies the requirements imposed by the robot software.

This paper is *not* about the verification of a particular software system, but rather about the way the verification could proceed in the case of mobile components. The main contribution of this paper is a case study exploring how the CPPCC approach lets us obtain formal guarantees on the proper collaboration of the components that make up a software system. In this case study it is assumed that the robot is being shared by various users who submit mission programs (mobile components) into the robot controlling system. The mission programs make use of a simple API of primitive robot controlling operations. The robot controlling system refuses the execution of mission programs that violate a cer-

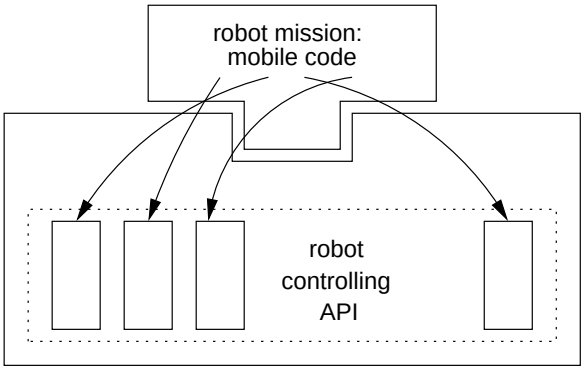

Figure 1. Verified mobile component expressing a robot mission integrates into the robot software

tain set of requirements. The requirements are expressed in terms of the formal specification of the robot control API.

The rest of the paper is organized as follows. In Section 2 the Certified Proved-Property-Carrying Code technology is summarized. Section 3 presents briefly a framework that implements CPPCC. Then Section 4 describes a concrete application of the CPPCC approach, where the correctness of the mobile code is described with respect to, and verified against, the formal specification of an API used by the mobile code. Finally, Section 5 concludes the paper.

2. Certified Proved-Property-Carrying Code

Certified Proved-Property-Carrying Code (CPPCC) is a technology for guaranteeing the correctness of mobile code with a special emphasis on efficiency. This technology was first proposed in [2], and a prototype implementation based on a functional programming language was presented in [3]. This paper applies an implementation of CPPCC which uses Java Virtual Machine bytecode as the target code [4]. The CPPCC technology and this particular implementation is described in more details in [5].

CPPCC combines the certificate-based and the PCC approaches in order to provide highly efficient use of verified mobile program components. In contrast to traditional certificate-based approaches, the code receiver need not trust in the intentions, the skill and the accuracy of every code producer: the code receiver can make a decision on whether or not to accept and utilize the received code

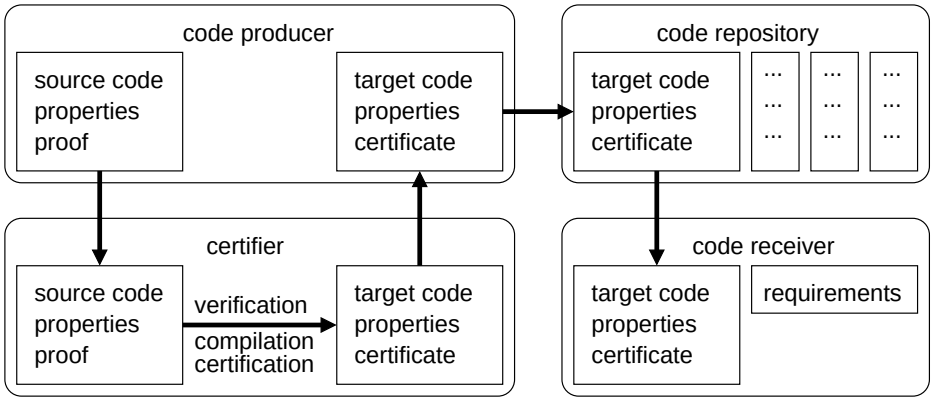

Figure 2. Overview of the Certified Proved-Property-Carrying Code technology

based on the declared properties of the code and on the opinion of a third-party, creditable certificate authority, which verified that the code indeed has the declared properties. Furthermore, in contrast to PCC, the run-time overhead compared to unverified mobile code does not become a burden. The increase in the size of the mobile code and in the time necessary to perform verification is significantly less than in the case of PCC.

As Figure 2 illustrates, participants of a CPPCC system play four different roles. The scenario for producing and receiving safe mobile components is the following.

1. The code producer develops a program component, and formulates and proves the properties of the program component based on its source code (proof systems that can be used for this purpose range from manual to fully automatic). The source code, the properties and the proofs are packed together and the package is sent to the certificate authority.
2. The certificate authority checks that the source code of the received program component has the specified properties by verifying the received proofs. Then it prepares the target code from the source code. Finally, it packs the target code and the properties together, signs the package and sends it back to the code producer.
3. The code producer uploads the signed package to a code repository.
4. The code receiver sends a requests to a code repository. The request contains a specification of what the code receiver needs. The code repository selects a signed package that satisfies the specification and sends this package to the code receiver.
5. The code receiver checks the certificate attached to the received package. If the signer is trusted, the code receiver compares its requirements with the

properties found in the package. If the properties match the requirements, the received code is linked into the code receiver and gets executed. The requirements against the mobile component form an integral part of the code receiver, and are formalized by the developer of the code receiver application.

The first three steps are performed "at static time" with respect to the code receiver, viz. before (and independently of) the execution of the code receiver application. Henceforth these steps have no run-time costs in the code receiver. Furthermore, the correctness proofs supplied for the mobile component are verified in step 2 before the generation of the target code, so the properties describing the correctness of the code can be expressed on the source code, at a sufficiently high level of abstraction.

The properties that the mobile code has to possess in order to be considered correct may be of various kinds. They may be functional (safety/security and progress) properties, but also extra-functional ones (QoS, resource consumption, timing etc.). The range of possible properties is influenced by the formal specification language and the verification tools utilized in any given implementation of CPPCC.

It is worth to mention here that neither CPPCC nor this paper deals with the important problem of having a compiler that is proved correct. It is assumed that correct source code is compiled into correct target code. Therefore the level of safety obtained by applying CPPCC depends on the quality of the compiler used by the certifier. The case study presented in this paper investigates functional correctness; extra-functional and quantitative properties (e.g. resource consumption) are even more vulnerable against malfunctioning compilers and incorrect code generation.

3. CPPCC for JVM bytecode

This paper uses an implementation of the CPPCC architecture for Java Virtual Machine bytecode [4, 5]. The class loading and dynamic linking mechanisms of the JVM provide an environment that is suitable for the transmission of platform independent mobile code. The main parts of this particular CPPCC implementation are a toolset (Project Packer), two middleware components (Certifier Server and Repository Server) and a Java module (CPPCC-API). In this CPPCC framework different programming languages and proof systems are incorporated. For the extensible robot controlling software presented here the B formal method [6] was applied. This setting is illustrated in Figure 3.

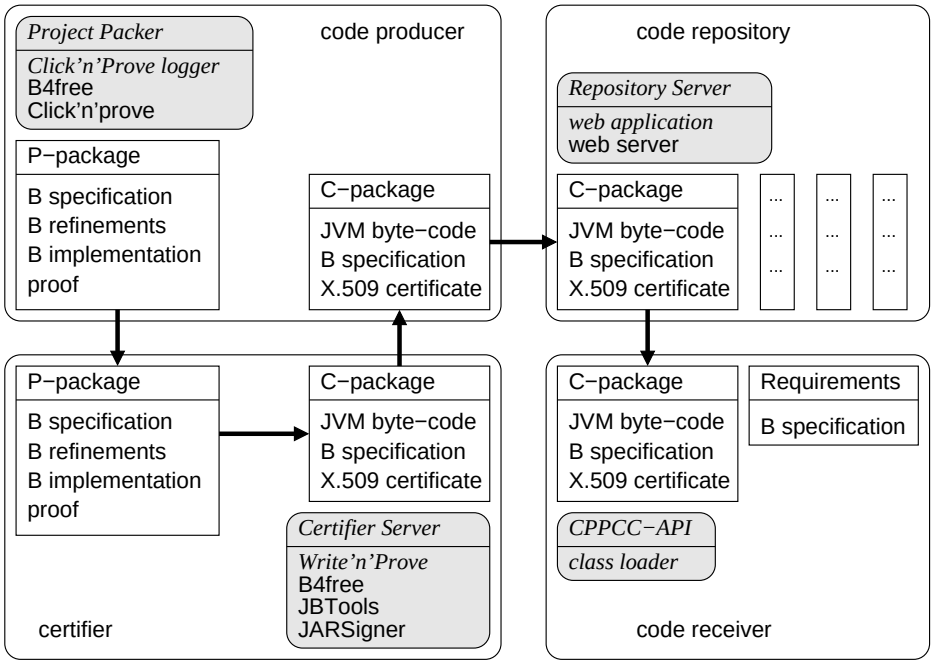

Figure 3. Using the B-method in the JVM-based implementation of CPPCC

- The Project Packer is a toolset for code producers. For program and proof development B4free [7] and Click'n'Prove [8] are used. In [4] an additional tool has been developed that can remember and store proofs (Click'n'Prove logger), and another one to produce "P-packages" (P stands for producer). Such a package contains a B specification, zero or more refinements, an implementation and the textual representation of B4free proofs.
- The Certifier Server accepts P-packages and produces C-packages (C stands for certifier). Write'n'Prove is a tool developed in [4]; it can interact with B4free in order to verify proofs found in a P-package. If the P-package is correct, (a slightly improved version [4] of) jBTools [9] is called which generates Java code from the B implementation module. The Java code is compiled into JVM bytecode, put into a Java Archive (JAR file) and signed with Sun's Java SDK [10]. The C package is the signed JAR file containing the B specification and the JVM bytecode of the mobile component.
- The Repository Server is a web-application serving requests for mobile components coming from code receivers through the HTTP-protocol.
- The CPPCC-API is a JVM class loader that can download JAR files from the Repository Server, verify the X.509 certificate and compare the requirements (a local file) with the properties found in the C package.

The producer of the mobile code applies the B-method to develop the source code of the mobile component together with the proof of correctness. The source code, its properties and the proofs are expressed as B machines, B refinements, B implementation modules and proof steps saved by the Click'n'Prove logger. The certifier checks the integrity of the B project by executing the proofs in Write'n'Prove and B4free. Then the target code (the JVM bytecode) is produced in two steps: Java code is generated from the B implementation modules and this Java code is compiled into JVM bytecode. The target code and the topmost-level B machine are packed together and are signed. The code receiver application contains a B machine describing the requirements imposed against the mobile component. This B machine has to be compared with the B machine found in the mobile component. The way this comparison is made may vary: it may be smart (but resource consuming) or it may be cheap (but simplistic). Currently our implementation uses trivial (textual) comparison.

4. Controlling a robot

The objective of the presented research was to develop a robot shared by multiple entities (people, software applications). These entities intend to assign different tasks (missions) to the robot. These tasks are expressed as mobile code components. In order to guarantee that each task finds the robot in a well-defined state, the tasks should be proven to satisfy certain requirements. Some examples of such requirements are the following.

- After completing the mission the robot is in its original position (if it is a robot, like in the presented case, that can change its position).
- The mission terminates.
- The mission terminates within a given amount of time.
- The resource (memory, time, power) consumption of the mission is within certain bounds.
- The robot does not go too far away from its base.
- The robot avoids dangerous situations (too high speed, too steep slopes and tilts).

One can consider this approach as a requirement to use a shared resource (the robot) in a transactional way. Each mission is assumed to be started from, and required to stop in a consistent state. Compare this with a job execution environment, e.g. with a grid system. In such a system a submitted job might be restricted to use a bounded amount of resources (CPU-time, memory etc.).

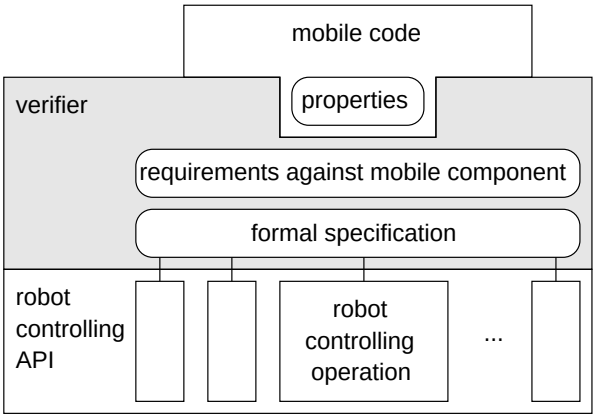

Figure 4. Checking properties of mobile components against requirements and API specification

If the job exceeds the limits, it will be aborted by a supervisor process. This approach is not feasible here: if the code controlling the robot were aborted, it might be impossible to reset the robot to a consistent, well-defined state. This is the case when a moving robot cannot find its way back to its base by itself, or it cannot proceed any longer if its power resources are exhausted. The only way to ensure that the missions do no harm to the robot, and that they leave the robot available for the forthcoming missions is to prove the correctness of the missions' code. A Certified Proved-Property-Carrying Code framework can support the transmission of safe mission programs to the robot in an automated way, without a human supervisor.

The software that controls the robot is highly modular. It is made up of components (the API containing primitives for robot control and mobile components expressing missions) which know nothing about the inner workings of one another apart from the formal specifications available in the CPPCC scenario. The goal of applying CPPCC in this case is to describe formally how the components interact. The properties of the received mobile components are expressed with respect to those of the primitive API operations. The verifier in Figure 4 checks whether the properties of the mobile component correspond to the requirements of the robot, assuming that the robot control API satisfies its formal specification.

4.1. Technical overview

A case study of the above ideas has been carried out with stock hardware, a simple universal NXT robot built from the LEGO Mindstorms kit [11]. The robot is moved by two motors, and it collects information through an ultrasound distance sensor. It has an ARM7 onboard microprocessor with 256KB of memory.

Different firmwares that make the operation of the motors and the sensors more convenient are available for this kit. There is a Java compiler for one of these firmwares, but it produces platform-dependent binary code and not the usual JVM bytecode. Dynamic loading and linking of components – which are necessary for mobile code technologies – are not supported. Therefore the robot is extended with an additional offboard computing device (a laptop, a palmtop or a mobile phone), which communicates with the NXT through Bluetooth.

The architecture of the robot control software is presented in Figure 5. A Java application running on the offboard computing device plays the role of the code receiver from the CPPCC model. It downloads mission programs from the code repository and executes those that correspond to the imposed requirements. A set of elementary programs – such as moving forward the robot by a unit, turning it left and right, and querying the distance sensor – has been made available for the mission programs through the robot control API. These elementary programs were implemented in a C-like programming language called NXC, compiled with the NBC [12] cross-compiler and pre-installed on the robot. The NXC programs are executed from the mission programs through the Bluetooth communication layer. It is assumed that the Bluetooth communication channel is reliable, safe and secured (e.g. only the offboard computer can access the robot).

The correctness of the mission programs has to be proven against the requirements and with respect to the formal specification of the API; both the requirements and the API specification are provided as B abstract machines. The code receiver Java application uses the CPPCC class-loader to integrate the mission programs. This class-loader is responsible for verifying the certificate attached to, and the properties of the mobile components. In the presented case each mobile component should contain two B machines: one to be compared with the specification of the robot control API, and another one to be compared with the requirements of the robot controlling system.

In this robot controlling case study the repository server is basically a job scheduler that makes the submitted mission programs available for the robot controlling system.

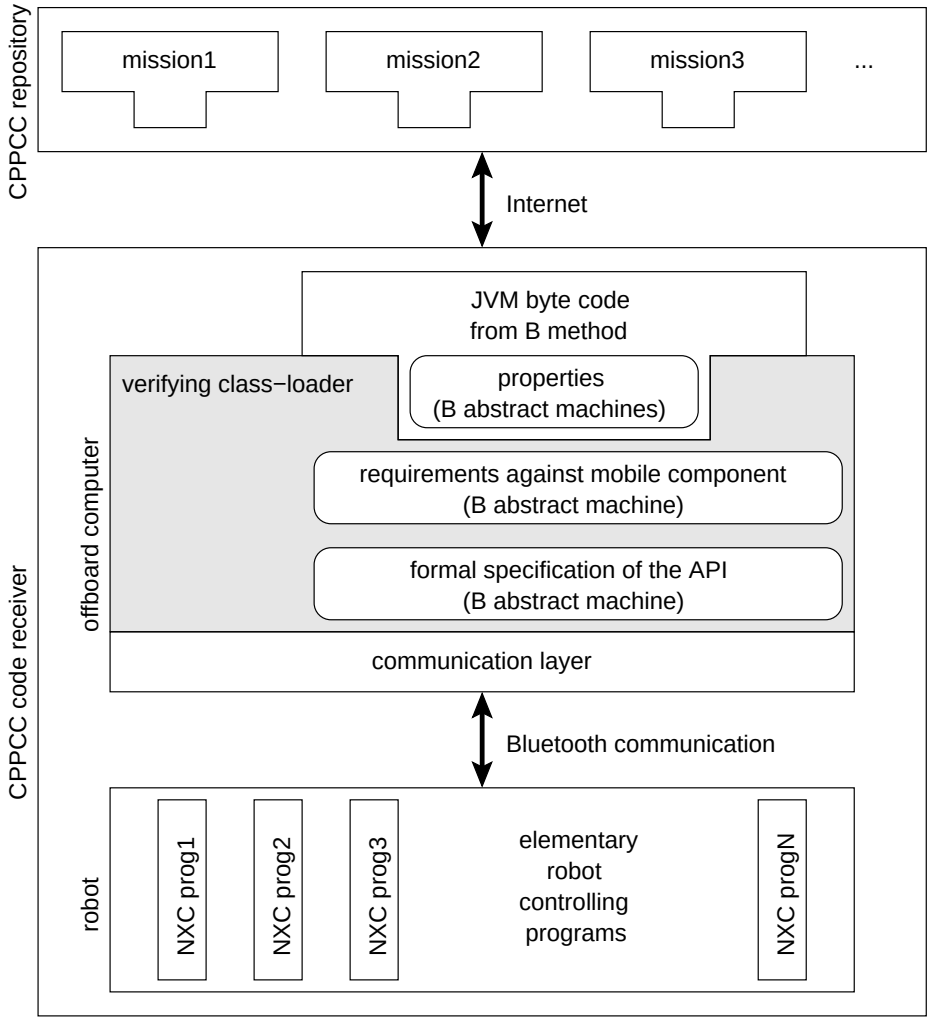

Figure 5. The architecture of the robot controlling software

4.2. The specification of the robot control API

In the experiment presented here the robot can move in a finite grid world. There are obstacles in certain positions in this world, which the robot cannot pass through, but which are sensible with its distance sensor.

The robot has five elementary programs.

- Turn left (by 90 degrees),
- turn right (by 90 degrees),
- turn back (by 180 degrees),
- query if there is an obstacle between the current position and the next position ahead ("*freeahead*"), and
- forward to the next position, if *freeahead* and the next position is not outside of the world.

In the B machine in Figure 6 it is specified that the world is of bounded size and that it is static: during the execution of a mission no new obstacles between grid positions appear. Furthermore, obstacles are symmetric, namely there is an obstacle between A and B exactly when there is an obstacle between B and A (this is expressed by the last four properties of `ENV`). Finally, the starting position `(0,0)` is accessible from every direction.

The NXC programs implementing the `elementary_routines` machine were not developed by using the B-method. However, they are simple enough (consisting one or two basic robot firmware instructions), so it can be verified by testing that they satisfy their specification – at least to the desired extent and precision.

4.3. Requirements against mission programs

As Figure 7 reveals, in this case study mission programs are considered correct if they satisfy the following requirement: at the end of the mission the robot is in its original `(0,0)` position, facing in the original `North` direction. Code producers must implement the mission program as the `route` operation of the `robot_navigation` abstract machine. The `elementary_routines` and `robot_navigation` machines should be made public so that producers of mission programs can base their implementations on these machines.

4.4. A simple mission program

As a proof of concept, a simple mission program has been developed using the B-method. This mission tries to move the robot forward to the northern border of the world: it succeeds if the robot finds no obstacles in the way. After reaching the northern border or finding an obstacle, the robot is turned back and returned to the original position as required.

The source code of the mission program consists of the `elementary_routines` and `robot_navigation` machines, and an implementation module containing the `route` operation of Figure 8.

5. Conclusions

This paper described a technique to ensure that components incorporated dynamically into a system co-operate correctly with the other components. This technique was illustrated with a case study, in which mission programs developed as mobile components were integrated into the controlling software of a robot. The technique also enables the safe dynamic reprogramming or reconfiguration of the robot by upgrading or replacing parts of its software with pieces of correct mobile code. The robot software can automatically and efficiently verify the correctness of the mobile code. To achieve this, an implementation of the Certified Proved-Property-Carrying Code technology is used. This implementation is based on Java Virtual Machine bytecode. Specifications, programs and proofs are devised, for instance, by using the B-method. Proofs are verified, JVM code is generated and signed by a trusted certifying authority. Mobile code is transmitted to the code receiver (to the main robot control software) together with its specification and with a certificate of correctness. The main robot control software executes the mobile component (describing a mission to complete) only if the properties of the component satisfy the requirements of the robot.

In the presented experiments the robot software was programmed in Java and NXC and the mobile code was developed with the B method. In the future an exploration of the possibilities of programming the NXT in Hume [13] and proving the correctness of mobile code written in Hume are planned.

References

[1] **Necula G.C.,** Proof-carrying code, *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York,* ACM, 1997, 106–119.

[2] **Horváth Z. and Kozsik T.,** Safe mobile code – CPPCC, Certified Proved-Property-Carrying Code, *Resource Management for Safe Languages, ECOOP, 2002*, Workshop Reader, eds. G. Czajkowski and J. Vitek, LNCS **2548**, Springer Verlag, 2002, 8-10.

[3] **Daxkobler K., Horváth Z. and Kozsik T.,** A prototype of CPPCC - safe functional mobile code in Clean, *Proceedings of Implementation of Functional Languages, IFL'02, Madrid, Spain,* 2002, 320–329.

[4] **Hoch C. and Tóth L.A.,** *Keretrendszer bizonyítottan helyes mobil kód támogatására,* (A framework supporting correct mobile code), Thesis for

the National Scientific Competition for Students, Hungary, pp. 88, 2006. (A framework supporting correct mobile code) (in Hungarian)

[5] **Istenes Z., Kozsik T., Hoch C. and Tóth L.A.,** Proving the correctness of mobile Java code, *Pure Mathematics and Applications*, **17** (3-4) (2006), 323-342.

[6] **Abrial J.R.,** (ed.), *The B-Book,* Cambridge University Press, 1996.

[7] **ClearSy,** *B4Free*, 2007.
`http://www.b4free.com/`.

[8] **Abrial J.R. and Cansell D.,** *Click'n'Prove*, 2007.
`http://www.loria.fr/ cansell/cnp.html`.

[9] **Voisinet J.C., Tatibouët B. and Hammad A.,** jBTools: An experiental platform for the formal B method, *Principles and practice in Java (PPPJ'02),* 2002, 137-140.

[10] **Sun Microsystems,** *The source for Java developers,* 2007.
`http://java.sun.com/`.

[11] **LEGO Group**, *Mindstorms*, 2007.
`http://mindstorms.lego.com/`.

[12] **Hansen J. and Andersen M.,** *NeXT Byte Codes Manual,* 2007.
`http://bricxcc.sourceforge.net/nbc/`.

[13] **Hammond K., Michaelson G. and Pointon R.,** *The Hume Report, Version 1.1,* 2007.,
`http://www-fp.dcs.st-and.ac.uk/hume/docs/index.shtml`.

Z. Istenes and T. Kozsik
Faculty of Informatics
Eötvös Loránd University
Pázmány Péter sét. 1/C.
H-1117 Budapest, Hungary
{istenes,kto}@inf.elte.hu

elementary_routines.mch

```
MACHINE
   elementary_routines
SETS
   DIRECTIONS = {North, East, South, West}
VARIABLES
   x, y, dir
ABSTRACT_CONSTANTS
   ENV
CONSTANTS
   world
PROPERTIES
   world : NATURAL & world = 5 &
   ENV : (-world..world) * (-world..world) * DIRECTIONS -->  0..1 &
   ENV(1,0,West)=1 & ENV(0,1,South)=1 & ENV(-1,0,East)=1 & ENV(0,-1,North)=1 &
   !(i,j).(i:-world..world & j:-world..world-1 & ENV(i,j,North)=1 => ENV(i,j+1,South)=1) &
   !(i,j).(i:-world..world & j:-world+1..world & ENV(i,j,South)=1 => ENV(i,j-1,North)=1) &
   !(i,j).(i:-world..world-1 & j:-world..world & ENV(i,j,East)=1  => ENV(i+1,j,West)=1)  &
   !(i,j).(i:-world+1..world & j:-world..world & ENV(i,j,West)=1  => ENV(i-1,j,East)=1)
INVARIANT
   x: INTEGER & x : -world..world &
   y: INTEGER & y : -world..world &
   dir : DIRECTIONS
INITIALISATION
   x := 0 || y := 0 || dir := North
OPERATIONS
   TurnBack  = CASE dir OF
                  EITHER North THEN dir := South
                  OR     West  THEN dir := East
                  OR     South THEN dir := North
                  ELSE                 dir := West
                  END
               END;
   TurnRight = ...
   TurnLeft  = ...
   d <-- FreeAhead =
              d := ENV(x,y,dir)
   Forward =
              IF ENV(x,y,dir) = 1 THEN
                CASE dir OF
                   EITHER North THEN IF y <  world THEN y:=y+1 END
                   OR     East  THEN IF x <  world THEN x:=x+1 END
                   OR     South THEN IF y > -world THEN y:=y-1 END
                   ELSE                 IF x > -world THEN x:=x-1 END
                   END
                END
              END
END
```

Figure 6. The specification of the elementary programs of the robot

robot_navigation.mch

```
MACHINE
   robot_navigation
INCLUDES
   elementary_routines
SETS
   STATES = {Moving, Stopped};
   RETURN = {Success, Failure}
VARIABLES
   state
INVARIANT
   state : STATES & ( state=Stopped => x=0 & y=0 & dir = North )
INITIALISATION
   state:= Stopped
OPERATIONS
   r <-- route =
      PRE
         x=0 & y=0 & dir = North
      THEN
         state:=Stopped || r :: RETURN
      END
END
```

Figure 7. The requirements imposed against the mission programs

mission1.imp

```
r <-- route =
VAR
   n, freeAhead
IN
   state := Moving;
   n := 0;
   freeAhead <-- FreeAhead;
   WHILE (freeAhead = 1) & n < world
   DO
      Forward;
      n := n+1;
      freeAhead <-- FreeAhead
   INVARIANT
      dir = North & x = 0 & n = y &
      y >= 0 & y <= world & y : NATURAL & y:(-world)..world &
      !i.(i:INTEGER & i>=0 & i<n => ENV(0,i,dir) = 1) &
      freeAhead = ENV(0,n,dir)
   VARIANT
      world - n
   END;
   IF n=world
      THEN r := Success
      ELSE r := Failure
   END;
   TurnBack;
   WHILE n > 0
   DO
      Forward;
      n := n - 1
   INVARIANT
      dir = South & x = 0 & n = y &
      y >= 0 & y <= world & y : NATURAL & y:(-world)..world &
      !i.(i:INTEGER & i>=0 & i<n => ENV(0,i,North) = 1) &
      !i.(i:INTEGER & i>=0 & i<n => ENV(0,i+1,South) = 1)
   VARIANT
      n
   END;
   TurnBack;
   state:=Stopped
END
```

Figure 8. A fragment of the implementation of a mission program