A Front End to a Java Based Environment for the Design of Embedded Systems

Daniel Karlsson
ESLAB, IDA
Linköpings universitet
581 83 Linköping, Sweden
danka@ida.liu.se

Petru Eles
ESLAB, IDA
Linköpings universitet
581 83 Linköping, Sweden
petel@ida.liu.se

Zebo Peng
ESLAB, IDA
Linköpings universitet
581 83 Linköping, Sweden
zebpe@ida.liu.se

Abstract. *During the design of embedded systems, at a certain point,the specification has to be transformed into an internal design representation. From that representation it should be possible to further perform system partitioning, mapping and scheduling. This report provides a framework to specify a system in Java and a way to automatically translate that specification into an internal design representation. That design representation is the Conditional Process Graph which captures both the data and the control flow at process level.*

1. INTRODUCTION

Embedded systems are getting more and more commonplace in areas like communication, multimedia, process control and consumer electronics. They usually have to fulfill strict requirements in terms of performance, safety, power consumption and cost efficiency. Simultaneously, time-to-market for new products has to be reduced. New design environments have to support the design of such systems from the early design phases, starting with the system specification. This specification, progressively, has to be refined into the hardware and software components of the embedded system [1].

In order to efficiently perform the successive design tasks, the system has to be modelled at an adequate abstraction level, so that only certain essential features of the system are captured. In our environment we use the *Conditional Process Graph (CPG)* as a design representation during system-level design.

The main advantages with Java as a specification language [2, 3, 4] are object orientation, platform independence and strong typing. Object orientation allows the programmer to encapsulate the code in objects, and to describe new objects by extending existing ones with new features. Furthermore, Java does not have historical legacy problems like C and C++ do. The strong typing prevents many errors from occurring at run-time, since they are caught already by the compiler.

For our purpose, Java has one big shortcoming, thread handling. Although it is easy to create threads and run different parts of the program concurrently, it is difficult to communicate between the different threads. Java threads are represented by an object which inherits the Thread class. The way communication is handled within standard Java is by invoking methods in another thread's object or by using monitors. Monitors are implemented as classes where some methods or pieces of code are *synchronised* (only one thread can execute that code at a time) [5]. Such a mechanism is at too low level and often inadequate for system level specification of distributed systems.

In order to provide a message based mechanism which is adequate for the specification of distributed embedded systems, a process specification and message based communication mechanism built on top of Java is proposed.

Similar problems have been attacked before. For instance in [4], the authors have developed a similar mechanism using Java for specification of reactive systems. They based their Java extension on Esterel's communication model.

[5] discusses the issues of Java's concurrency support in more detail. It also proposes an extension to standard Java where communication can take place both synchronously and asynchronously. The asynchronous communication is done by monitors. The disadvantage with this approach is that it introduces some new syntax to Java which makes it impossible to compile such programs with a standard Java compiler.

A prototyping environment for hardware/software systems based on Java is described in [6]. It discusses how to determine which methods in the specification (which is written in Java) should be implemented in hardware and which methods should be implemented in software. Moreover, it discusses the translation from Java source code into either Java bytecode or VHDL. This approach is a detour with respect to our approach where we want to be able to compile a Java specification directly into the design representation.

[7, 8, 9] focus on methods to model and simulate hardware systems based on Java descriptions. [10] describes a method where a system specified in a general purpose language (Java) is successively refined to a form consistent with a formal model of computation.

In March 1999, the Real-Time for Java Experts Group (RTJEG) has started an effort aimed at developping the real-time specification for Java (RTSJ) [11]. Thus, a platform is provided for developing real-time applications in Java. The mechanisms described in the present paper are situated at a higher level and fit with the lower level mechanism proposed in the RTSJ.

The remainder of this paper is organised as follows. Section 2 introduces the concept of CPG, the design representation used in our design environment. Section 3 presents the Java classes which have been introduced. In section 4, we describe the problem of translating Java into CPGs. Next, section 5 describes an application. Conclusions are presented in section 6.

2. THE INTERNAL DESIGN REPRESENTATION

As an internal design representation, we use a CPG. It is defined as a directed, acyclic, polar graph $\Gamma(V, E_S, E_C)$ [12]. Each node $P_i \in V$ represents one process. E_S and E_C are the sets of simple and conditional edges respectively. $E_S \cap E_C = \emptyset$ and

$E_S \cup E_C = E$, where E is the set of all edges. An edge $e_{ij} \in E$ from P_i to P_j indicates that the output of P_i is the input of P_j. The graph is polar, which means that there are two nodes, called *source* and *sink*, that conventionally represent the first and last processes. These nodes are introduced as dummy processes so that all other nodes in the graph are successors of the source and predecessors of the sink respectively.

An edge $e_{ij} \in E_C$ is a *conditional edge* (thick lines in Fig. 1) and it has an associated condition. Communication on such an edge only takes place if the associated condition is satisfied. We call a node with conditional edges at its output a *disjunction process*. Alternative paths starting from a disjunction process, which correspond to complementary values of a certain condition, are disjoint and they meet in a so called *conjunction process*. Conditions are dynamically computed by disjunction processes and their value is unpredictable at the start of an execution cycle of the CPG. In Fig. 1 circles representing conjunction and disjunction processes are depicted with thick borders. Process p_1 and p_2 are disjunction processes while p_6 and p_9 are conjunction processes. p_1 and p_9 are the source and the sink respectively. We assume that conditions are independent. A boolean expression X_{Pi}, called a guard, can be associated to each node P_i in the graph. It represents the necessary conditions for the respective process to be activated. For some nodes in the graph in Fig. 1, the guards are: $X_{P1} = true$, $X_{P5} = A \wedge \bar{B}$, $X_{P7} = \bar{A}$ and $X_{P9} = true$.

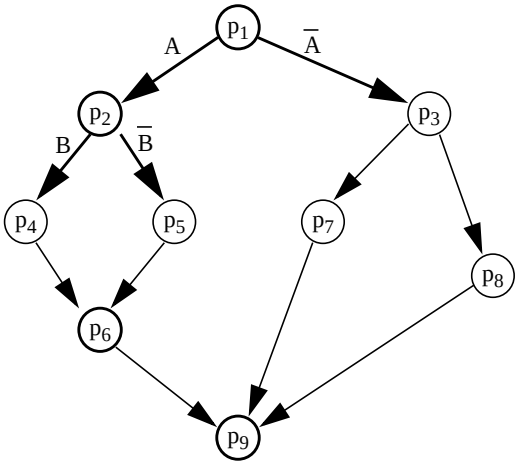


Fig. 1: Example of a CPG with two conditions

A process, which is not a conjunction process, can be activated only after all its inputs have arrived. A conjunction process can be activated after messages coming on one of the alternative paths have arrived. Thus, for example, in the graph in Fig. 1 process p_7 can not be activated before receiving the message from p_3. p_9 is not activated until either a message is received from p6 or from both p_7 and p_8. All processes issue their outputs when they terminate. If we consider the activation time of the source process as a reference, the finishing time of the sink process is the delay of the system at a certain execution.

One essential feature fo CPGs is the ability to capture both data and control dependency among processes. The CPG is sufficiently general as to capture a large class of distributed embedded applications with allowing at the same time to perform the synthesis tasks and timing analysis needed for the implementation of hard real-time systems. In [13] we have discussed the problem of hardware/software partitioning using a graph based representation. Static scheduling of hard real-time distributed systems represented as CPGs has been discussed in [12], while in [14, 15] we presented algorithms for schedulability analysis and performance estimation with priority based scheduling. In the following sections of this paper we concentrate on aspects related to the specification of systems using Java and the translation of such a specification into a CPG representation.

3. SPECIFICATION IN JAVA

In section 1 we have mentioned several advantages of Java as a specification language for embedded systems. Its greatest shortcoming for us is its thread handling and communication mechanism. Fig. 2 illustrates the communication process between threads in standard Java. The thread objects must possess a reference to every other thread they are supposed to communicate with. The sender, T1, invokes a method, *mr*, in the recipient object, T2. mr manipulates a field, containing the message, which can be read by T2. Note that this procedure is highly asynchronous. It could even happen that T2 reads the message field before T1 has put a value in it. This leads to an erroneous state of the system. The procedure is similar when T2 sends back the reply to T1. To avoid this problem you need a flag based synchronisation. As discussed earlier, the standard Java communication mechanism is at a too low level for the purpose of system level specification.

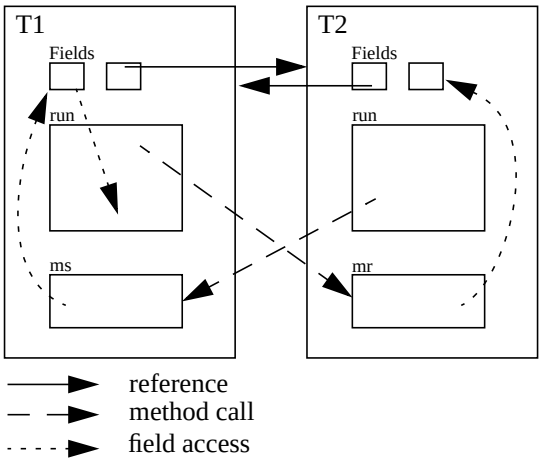


Fig. 2: Thread communication in standard Java

In the rest of this section we present a mechanism for process specification and message based communication in Java as well as the associated class library. In section 4 we discuss the translation of such a specification into CPGs.

A. The Message Passing Centre

According to our methodology the system specification consists of parallel processes. Processes are instantiations of the predefined class `CPGProcess`, which will be discussed in section C. They interact exclusively by message passing using the methods `send` and `receive` which are inherited from the class `CPGProcess`.

The *Message Passing Centre* (MPC) is the control engine for the implementation of the proposed communication mechanism. All messages in the system must be transferred via the MPC. It is strictly forbidden to call a method of another thread object or in any other way to bypass this mechanism.

The user should not try to manipulate, directly, the MPC himself. To communicate he should use the methods provided by the class `CPGProcess` and let `CPGProcess` interact with the MPC.

The MPC is a passive object (not a thread) which contains several queues, one for each registered process. How registration is done is described in section B. When process p1 sends a message to process p2, the following operations are performed inside the MPC. A pair consisting of the message and its own identity (the identity is explained later) is put in the queue of process p2. Then all processes are notified of the event and p1 continues with its own execution independent of p2.

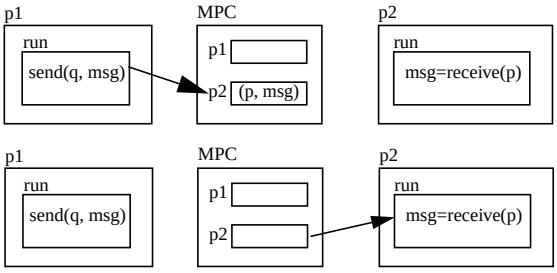


Fig. 3: Message communication through the MPC

When p2 executes the receive statement, its queue in the MPC is searched to see if there is any message from a process p2 expects a message from. If there was not, p2 suspends itself and awaits a notification from a sending process. When all expected messages are received, the receive method finishes and returns the messages. Fig. 3 illustrates this mechanism.

B. The main class

The *main class* is the class in which the method `main` is situated. Hence, it is the class which is given to the JVM at execution. The main method can be put in one of the processes, but it can be put in a class of its own.

```
public static void main(String[] args){
  MPC mpc = new MPC();
  new Process1(mpc, "p1");
  new Process2(mpc, "p2");
  new Process3(mpc, "p3");
  new Process4(mpc, "p4");
  mpc.start();
}
```
Fig. 4: Example of a main method

The main method should create an MPC and instantiate all the processes by giving them a reference to the MPC and an identity. From this point, the processes are only referred to by the given identity. Fig. 4 shows an example of a main method. We can easily distinguish the different parts just described.

C. The CPGProcess class

The `CPGProcess` class is the superclass of all processes. There must be no object which inherits directly or indirectly from the class `Thread`, except for the `CPGProcess` class.

The main purpose of this class is to give the user easier access to (and hide) the methods in the MPC. It provides mainly the following methods: `send`, `receive` and `getDataFromMessageArray` (which is not further discussed in this paper). The send and receive methods only forward the call to the corresponding methods in the MPC.

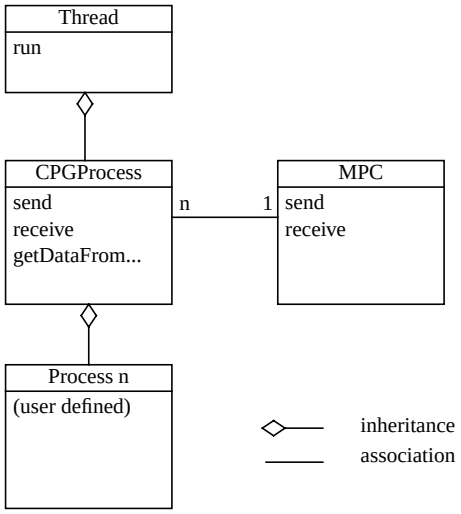


Fig. 5: The class diagram

The send method delivers a message to a destination by passing it to the MPC. It takes two arguments: process identity and message. The process identity is the string value which was assigned to the destination process when it was initialised in the main method. The message could be any value of `Object` type. It is up to the receiver to know how to extract the relevant information from the message. After a message has been sent, the execution immediately continues and the sending process does not wait for another process to accept the message.

The receive statement takes only one argument, which is an expression of process identities. The argument is an expression since a process can wait for messages from several processes at the same time. Two operations can be used in the expression: and (`&&`) and or (`||`). When a message from, for instance, p1 arrives, it can be interpreted as assigning the boolean value *true* to the variable in the expression with the same name as the process identity. Initially, all variables are considered to have the value *false*. The previously mentioned operations, `&&` and `||`, then correspond to their boolean equivalents. As soon as the expression becomes true, the received messages are returned in an array of which each element represents a pair: process identity and the message received from the respective process.

An overview of the class hierarchy is shown in Fig. 5. Examples on how send and receive statements may look like can be found in Fig. 6.

D. Specifying a process

A process is written as its own class extending the `CPGProcess` class. To make it possible to initialise the process in the main method, a constructor method must be provided. It should pass on the process identity and the MPC reference to its parent. Other initialisations could also be included.

The specification of what the process should execute is entered in the process's run method. We should remember that `CPGProcess` inherits from `Thread`, so it is actually `Thread`'s run method which we override. The run method could basically contain any Java code. Certain limitations imposed by our environment are further discussed in the following section.

Fig. 6 shows the code of the first process instantiated in Fig. 4. The second statement in the run method calculates the value of a condition. In this case, we generate it as a random value. Depending on this condition, we send a message to either p2 or p3. When either of these messages is sent, we wait for a message from p4 and for a second message which comes from either p2 or

p3. We then store the message received from p4 in the integer variable `mp4`. The code for the other processes is not presented here.

E. Execution

A great advantage of our Java based specification approach, is that it is easy to execute and hence to achieve a simple form of simulation. After our initial validation of the specification further design steps, like architecture selection, mapping and scheduling have to be performed [1,16]. In order to perform these design steps our initial Java specification has to be translated into an internal design representation, which in our case is the CPG.

It is important to mention that our translator not only creates an abstract CPG but also an equivalent executable Java program. This program is semantically equivalent with the input specification, but differs from it in its structure. This allows to perform simulations during the subsequent design steps and thus to validate further refinements of the initial specification.

```
import java.util.Random;
public class Process1 extends CPGProcess {
   public Process1(MPC mpc, String name) {
      super(mpc, name);
   }

   public void run() {
      System.out.println("Calculating " +
         "condition..");
      boolean condition =
         new Random().nextInt(2) == 0;
      System.out.println("Condition is:" +
         condition);
      if (condition) {
         send("p2", new Integer(3));
      } else {
         send("p3", null);
      }
      Message[] m =
         receive("p4&&(p2||p3)");
      int mp4 = ((Integer)
         getDataFromMessageArray(m, "p4"))
         .intValue();
      System.out.println("Received " +
         mp4 + " from process p4.");
   }
}
```

Fig. 6: Example of a process class

4. COMPILING JAVA SPECIFICATIONS INTO CPGS

The translator performs two tasks. One is to produce a CPG and the other one is to produce new Java code which is a direct correspondence of the graph.

For the parser, Sun Microsystems' Java Compiler Compiler [5] has been used. The JavaCC generates a parser as Java classes based on some grammar. In this case, the grammar for Java 1.1 was used. The resulting parser was then modified to build an abstract syntax tree (AST). All of the algorithms of the translator work on the AST and not directly on the source code itself.

The algorithms in the translator have a graph data structure to their help. When the translation is finished, the graph should reflect the resulting CPG. There is information stored in the nodes and edges to guide the algorithm. The nodes contain the following information: the identity of the process it represents, a split index and the code associated with the node. The split index is a number which differentiates nodes belonging to the same process. The code associated with a certain node is initially the same AST as the parsed input specification for that process. An edge has a field for the condition associated to it and a flag called *matched*. The use of the flag *matched* is explained later.

F. The translation algorithms

The translator is composed of three main algorithms: The parser, the main method analyser and the run method analyser.

The main method analyser receives as input the AST of the main method. It checks statement after statement until it finds one which creates a new process. At that point, it adds a new node to the graph and fills it with the information mentioned above. When the whole main method is analysed, the graph hence only consists of as many nodes as there are processes and no edges. After the analysis of the main method in Fig. 4, for example, the graph data structure will contain four nodes and no edges.

The run method analyser looks for communication statements (send or receive) inside the run methods of the processes. If one is encountered, an edge is added between the two nodes involved. That edge is also added to a set, called *recent*. The set *recent* contains all edges which are created during the analysis of the current process. Before analysing a new process, the set is emptied. If there already was an edge between the two nodes, the flag *matched* is set to true. This means that the algorithm has encountered both the receive statement and the send statement for that edge. Both the set *recent* and the flag *matched* are used in the splitting algorithm described in section G.

```
public void run()
{
   receive(”p2&&p3”);
   System.out.println(”start..”);
   a = 5;
   str = ”CPGs are great!”;
   System.out.println(”Sending results”);
   send(”p4”, new Integer(a));
   send(”p5”, str);
}
```

Fig. 7: The run method of a process p1

Fig. 7 presents a simple example of a process, called p1, which is now used to clarify the basic algorithm. Assume that the processes are analysed in the following order (determined by chance): p2, p4, p1, p3, p5. The first statement in p1 is a communication statement. We have to check if there already is an edge. Considering the order in which the processes are analysed, there is an edge from p2 to p1, but not from p3 to p1. This means that the flag *matched* of the edge from p2 to p1 is set to true, and that a new edge is created connecting the processes p3 and p1. The newly created edge is also put in the set *recent*. After the receive statement four non-communication statements follow. They are merely skipped since they do not interact with any other process. The treatment of the send statements is the same as with the receive statement mentioned before. Since the edge from p1 to p4 already exists, its flag *matched* is set to true and a new edge from p1 to p5 is created. The resulting graph of this example is shown in Fig. 8. We note that it is not a complete CPG.

What has been discussed until this point is the basic algorithm. However, the following exceptional situations might occur which stem from the semantics of the CPG.

1. The semantics of a CPG states that a process can not be activated until it has received all its input data and that outputs are

generated when the process terminates. This implies that all processes must begin with a receive statement and end with send statements. This problem is solved with process splitting.

2. The branches of an if statement can contain a mixture of communication and non-communication statements. Such situations can not be solved with an ordinary split, but have to be taken care of separately.
3. CPGs are not allowed to contain cycles. However, in the input specification it is possible for processes to mutually communicate back and forth. These situations are also dealt with by process splitting.
4. The resulted CPG has to be a polar graph.

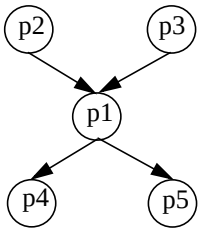


Fig. 8: The graph of the code in Fig. 7

The following sections, G, H and I, present the solutions to the problems identified above.

G. Splitting processes

The semantics of a process in a CPG is that it must not start to execute until it has received all the needed inputs. With this background, it is evident that the first statement in a process must be a receive statement or the process does not have any receive statements. The situation is similar to send statements which must be the last statements. If these conditions do not hold in the input specification, the specification must be transformed in such a way that the conditions hold in the output code. The graph must also be changed in order to reflect these transformations.

Formally, a node split has to take place if either of the following two conditions is satisfied:

1. The previously analysed statement was a send statement and the current one is not.
2. The currently analysed statement is a receive statement and it is not the first statement in the process.

When a split has to be performed the following steps are executed:

1. A new node is inserted into the graph. Its split index is one more than the previously highest split index among the nodes of that process.
2. All edges with the flag *matched* set to false which were connected to the process' node and not contained in the set *recent*, are moved so that they now refer to the new node.
3. An extra edge between the original node and the one created in step 1 must be added to enforce the relative order between them. The edge also serves to transfer the execution environment, the values of the fields and variables.
4. The flag *matched* of the edge created in step 3 is set to true, since we implicitly have encountered both the send and the receive statements.
5. The set *recent* is emptied, since a new process is about to be analysed.

When new edges which correspond to communication with the process being analysed are to be added, they must connect to the new node inserted in step 1 above.

Fig. 9 illustrates how a node is split. The number after the process identity and the underscore character denotes the node's split index. The new node has the same process identity but a higher split index. Edges crossed with a line are matched edges. They are not moved, but all other edges connected to the node are (except for those contained in the set recent).

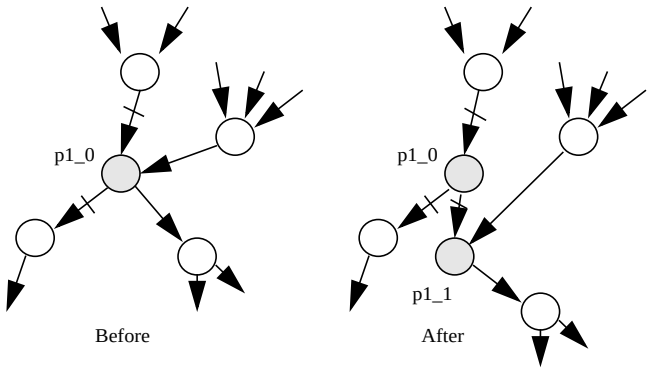


Fig. 9: A node being split

By following this general split algorithm we automatically solve cyclic programs where process 1 sends a message to process 2 which later sends back to process 1. Without node splitting, this would produce a cyclic graph. The example in Fig. 6 illustrates such a situation. The graph resulted after node splitting is performed on the graph in Fig. 10(a) is shown in Fig. 10(b). By performing the split, the statement which sends a message to process p2 becomes the last one in the process p1_0 (or only have other send statements after it). The receive statement ends up in another process (p1_1 in our case) and the cycle is broken. (See also Fig. 14.)

H. Adding source and sink nodes

When all processes are analysed, the graph could still not be a CPG because it is not polar. If there are several source or sink nodes, we must add dummy nodes. Their task is only to send or receive signals (empty messages) to the first and from the last nodes respectively. On the other hand, if there already was only one source or sink node, this procedure is not necessary and consequently not performed.

I. If statement splits

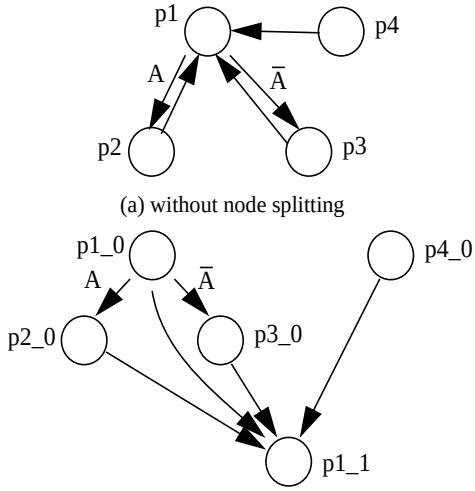
Using the mechanisms described so far, we can only deal with two types of if statements:

1. No communication statement (send or receive) exists in either branch.
2. Both branches contain exclusively send statements.

Using a so called if statement split, however, if statements can be handled without restrictions. In an if statement split, three new nodes are introduced, one for each branch and one for the continuation of the process after the if statement. The last mentioned node is further on referred to as the *collect node*.

Fig. 11 shows an example with code and the corresponding graph. The shaded nodes are the ones which are involved in the if statement split. When the if statement is encountered the following steps are performed:

1. A new node is inserted (with a higher split index than that of any other node of the same process) into the graph. The code in the true branch of the if statement is assigned to it and recursively analysed.
2. Step 1 is repeated for the false branch.
3. Finally the collect node is inserted, to which the code after the if statement is assigned.
4. The edges are added between the nodes inserted in the previous steps as shown in Fig. 11.



(a) without node splitting

(b) with node splitting

Fig. 10: The CPGs corresponding to the example in Fig. 6

Note that it could happen that the inserted nodes must be split recursively. For instance, if a receive statement is not the first statement in one of the if branches, the branch node must be split so that the receive statement does come first in its process. The diamond shape of the shaded nodes in Fig. 11 is typical for the if statement split.

```
System.out.println("Statement before if");
if (condition) {
  System.out.println("True branch");
  send("p2", null);
} else {
  System.out.println("False branch");
  send("p3", null);
}
receive("p2||p3");
System.out.println("Statement after if");
```

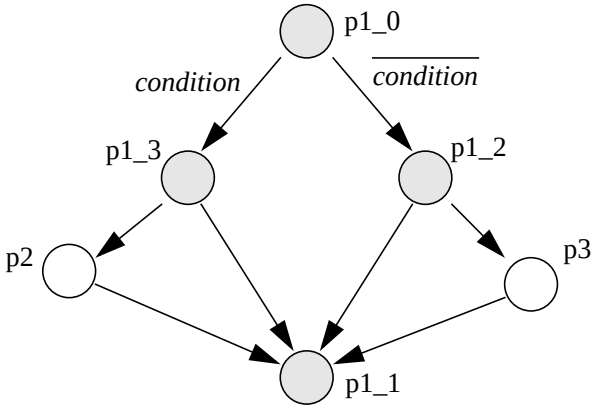


Fig. 11: Example code and the corresponding graph of an ”if split”

J. Discussion on the algorithm

Fig. 12 gives an outline of the algorithm. As we can see, the AST is traversed calling the algorithm recursively. The first branches of the outermost if-else if statements only make the algorithm move deeper down the syntax tree. Finally, we arrive at the statement level where the most interesting things happen. If the code corresponds to a receive statement, the node may have to be split or a new edge is either added or marked as matched.

Even if the Java specification is following the mechanisms as described in section 3, it can happen that, using the translation algorithm discussed above, no generation of a CPG is possible. Take, for example, the two sequences in Fig. 13. They show two loops, one with a statically known number of iterations and one with a statically unknown number of iterations. In a CPG, there is one edge per send. If there is a send statement within the body of a loop, there should hence be several edges corresponding to that send statement, one for each iteration. The strategy is to perform loop unrolling. This is possible for the code in Fig. 13(a) since we can derive the number of iterations statically. Clearly, it is not possible to perform loop unrolling when the number of iterations is unknown at compilation time, like the case in Fig. 13(b). What can be done instead, is to merge the process containing the for statement with the one which is the destination of the send. This means that the two nodes in the CPG are collapsed into a single one. In an extreme case, the whole application can degenerate into one single node by successive merges. An excessive number of merges, however, should be a sign that the application has been structured by the designer in an inappropriate way.

K. Code generation

Beside generation of a CPG, the task of the translator is code generation. The resulting code should directly reflect the graph after possible process splits. By this we get a system specification which can be simulated and is both functionally and semantically equivalent to the CPG representation. This representation/specification is the starting point for further design steps [12].

As mentioned in the previous sections, the execution environments of processes must be transferred in messages between split

nodes. To be able to do that, special message classes have to be created. At the same time, code must be added to transfer the control to the new process and to transfer the execution environment, i.e. variable values, to the new process.

It is also important that we update all references to other processes with their new process identities. The new identity is the process' old identity and its split index, separated by an underscore character. For send and receive statements, these updates can not take place before the corresponding edge is matched, since the final process identity is not determined until then.

Fig. 14 shows an example of how the generated code might look like. It is the code corresponding to node p1_0 in Fig. 10.

```
for (int i=0; i<10; i++) {
   send("proc", msg);
}
```

(a) loop with a known number of iterations

```
for (int i=0; i<n; i++) {
   send("proc", msg);
}
```

(b) loop with an unknown number of iterations

Fig. 13: Loops which could be treated differently

TABLE 1.
Summary of the output signals

Signal	Description
cca	Cruise control is allowed to control the engine torque.
ccont	Cruise control is on but not necessarily active
ccsp	Reference speed
tcc	Request to the engine torque

5. EXAMPLE

In this section a real life example from the automotive area will be discussed, namely an automatic cruise control system [18].

The system consists of several components connected in a network environment. The specification does not cover all of these parts, but only the core parts and regards the other parts only as input providers. The core part is called CCM (Cruise control module).

To operate the cruise control, the driver has five buttons, on-off, increase speed, decrease speed, resume speed and cancel buttons. The component which handles these buttons is referred to as the CEM (Central Electronic Module). Signals from other components are, for instance, speed, battery voltage, brake pedal and the quality of certain signals. The system emits four output signals, which are summarised in Table 1.

Starting from a description of the system in natural language, we first have produced an equivalent specification in Java. The initial Java specification of the system corresponds to the process graph in Fig. 15, which consists of 13 processes. Several of the processes in the graph correspond to output signals or are auxiliary processes (prefixed with the signal name) to such processes. Another category of processes are those lined up on the top of the figure. They provide the system with input. This graph, obviously, is not a CPG and, thus, could not be handled by our design environment.

However, starting from the initial Java specification, the translation tool will generate a new Java program and the corresponding CPG shown in Fig. 16. In our case the translator tool has to resolve the cycles around processes cca, ccsp and ccf in Fig. 15. The cycles from ccsp to ccsp_set_speed and ccsp_change_speed and back are solved by an if statement split. According to the translation algorithm, the node ccsp is hence split into nodes ccsp_0 ... ccsp_3 as shown in Fig. 16. The other two cycles (around cca and ccf) are handled with ordinary splits, as discussed in section G. The addition of a source node is also performed. The resulting graph after the translation is shown in Fig. 16. As result of this translation, a corresponding Java program is also generated. The CPG in Fig. 16 is used for system partitioning and scheduling, as shown in [12, 13, 14].

```
Global data: CPG graph, enum status, String processName; Set recent
procedure analyseCode(AST code)
    if code = null
        return;
    else if isClass(code)
        status := start_status;
        makeEmpty(recent);
        analyseCode(getMethod(code, "run"));
    else if isMethod(code)
        analyseCode(getStmts(code));
    else if isStmtList(code)
        analyseCode(getFirstStmt(code));
        analyseCode(getRestStmts(code));
    else if isStatement(code)    statement level
        if isMethodCall(code)
            if isReceiveStmt(code)    receive statement
                if status = send_status or status = work_status;
    previous statement was either a send statement or a non-communication statement
                    CPGNode oldNode := getNode(graph, processName);
                    splitProcess(graph, processName, recent);
                    CPGNode newNode := getNode(graph, processName);
                    fixCode(oldNode, newNode);
                    analyseCode(getCode(newNode));
                    status := end_status;
                else
                    ParseTree tree := parseArg(getFirstArg(code));
                    status := receive_status;
                    for each variable v in tree do
                        if exists edge between the nodes of v and processName
                            setMatched(getEdge(graph, v, processName));
                        else
                            CPGEdge newEdge := addEdge(graph, v,
                                processName);
                            add(recent, newEdge);
            else if isSendStmt(code)
                status := send_status;
                if exists edge between the nodes of processes v and
                        processName
                    setMatched(getEdge(graph, v, processName));
                else
                    CPGEdge newEdge := addEdge(graph, v, processName);
                    add(recent, newEdge);
            else
                status := work_status;
        else
            if status = send_status
                split process the same way as above
        if check other kinds of statements...
```

Fig. 12: Outline of the run method analyser algorithm

```
public void run() {
  receive("source_0");
  System.out.println("Calculating " +
     condition..");
  boolean condition =
     new Random() . nextInt(2);
  System.out.println("Condition is: " +
     condition);
  p1_0p1_1Message _msg =
     new p1_0p1_1Message();
  _msg.__condition = condition;
  if (condition) {
    send("p2_0", new Integer(3));
  } else {
    send("p3_0", null);
  }
  send("p1_1", _msg);
}
```

Fig. 14: The code generated for node p1_0 in Fig. 10

6. CONCLUSIONS

In this paper we have presented a Java frontend to a design environement for embedded systems. In order to make the specification language more suitable to system level specification of distributed embedded systems, we have introduced a message based communication mechanism and the corresponding class library. A method has been presented in order to translate the Java specification into the CPG representation used by a design environment which performs architecture selection, process mapping and system scheduling. In this context the main issues to be solved are: compliance with the CPG communication semantics, communication in if statements, cycles and polarity. Together with the generation of the abstract CPG representation, a transformed Java program is produced, which reflects the CPG structure and supports further simulation during the design process.

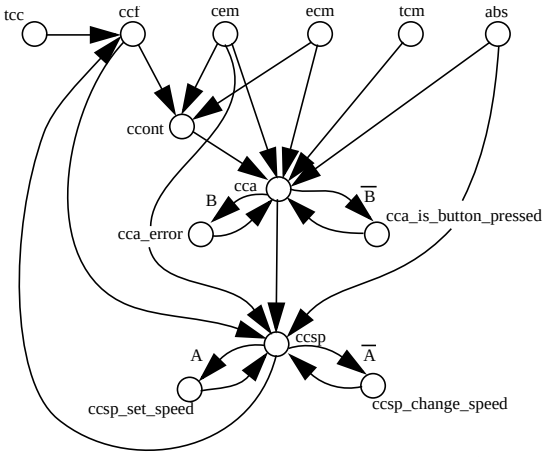


Fig. 15: The process graph corresponding to the CCM specification

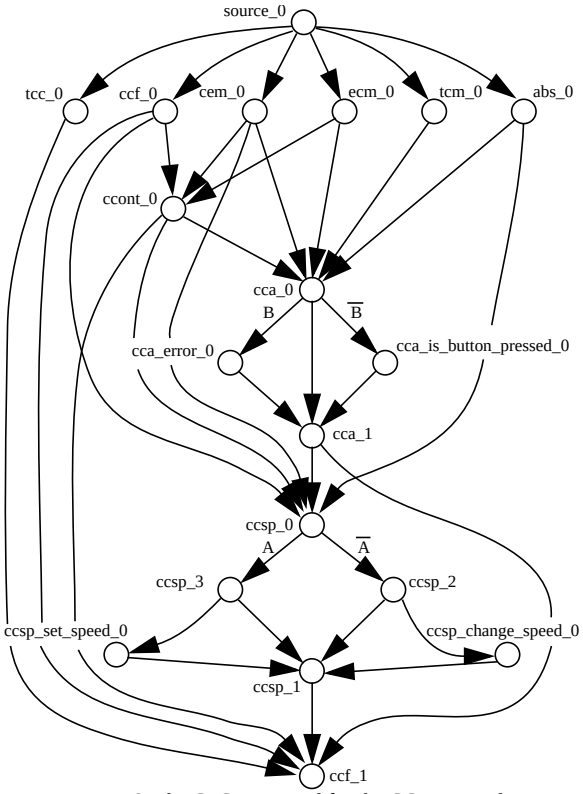


Fig. 16: The CPG generated for the CCM example

REFERENCES

[1] D. Gajski, and F. Vahid, Specification and Design of Embedded Hardware-Software Systems, In *IEEE Design & Test of Computers*, vol. 121, pages 53-67, 1995

[2] K. Arnold, and J. Gosling, The Java programming language, Addison-Wesley, Reading, Massachusetts, 1996

[3] R Helaihel, and K Olukotun. Java as a Specification Language for Hardware-Software Systems, In *IEEE/ACM Int. Conf. On Computer Aided Design*, 1997

[4] C Passerone, C. Sansoè, L. Lavagno, R. McGeer, J. Martin, R. Passerone, and A. Sangiovanni-Vincentelli, Modelling Reactive Systems in Java, In *ACM Transactions on Design Automation of Electronic Systems*, Vol 3, No 4, pages 515-523

[5] L.E. Nugroho, and A. Sajeev, Java4P: Java with High-Level Concurrency Constructs, In *IEEE Int. Symp. On Parallel Architectures, Algorithms, and Networks*, 1999

[6] J. Fleischmann, K. Buchenrieder, and R. Kress, A Hardware/ Software Prototyping Environment for Dynamically Reconfigurable Embedded Systems, In *IEEE Int. Workshop on HW/SW Codesign*, pages 105-109, 1998

[7] T. Kuhn, W. Rosenstiel, and U. Kebschull, Object Oriented Hardware Modeling and Simulation Based on Java, In *Int. Workshop on IP based Synthesis and System Design*, 1998

[8] T. Kuhn, and W. Rosenstiel, Java Based Modeling and Simulation of Digital Systems on Register Transfer Level, In *Workshop on System Automation*, 1998.

[9] T. Kuhn, W. Rosenstiel, and U. Kebschull, Description and Simulation of Hardware/Software Systems with Java, In *ACM Design Automation Conference*, pages 790-793, 1999

[10] J. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, and A. Newton, Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement, In *ACM Design Automation Conference*, pages 70-75, 1998

[11] G. Bollella, and J. Gosling, The Real-Time Specification for Java, Computer, In *IEEE Journal*, pages 47-54, June 2000

[12] P. Eles, A. Doboli, P. Pop, and Z. Peng, Scheduling with Bus Access Optimization for Distributed Embedded Systems, In *IEEE Transactions on VLSI Systems*, 2000.

[13] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search, *Journal on Design Automation for Embedded Systems*, vol. 2, 5-32, 1997.

[14] P. Pop, P. Eles, and Z. Peng, Bus Access Optimization for Distributed Embedded Systems Based on Schedulability Analysis, In IEEE *Design, Automation & Test In Europe Conference*, pages 567-574, 2000

[15] P. Pop, P. Eles, and Z. Peng, Performance Estimation for Embedded Systems with Data and Control Dependencies, *In IEEE Int. Workshop on HW/SW Codesign*, pages 62-66, 2000

[16] P. Eles, K. Kuchcinski, and Z. Peng, System Synthesis with VHDL, Kluwer Academic Publishers, 1998

[17] http://www.metamata.com/JavaCC/ , March 2001

[18] A Lindbom, Function Requirement Description, Engine, Cruise control, Volvo Technological Development, 1998