From Java to mCRL2

Bachelor Thesis

Allan van Hulst
Computer Science

November 23, 2007

Chapter 1

Introduction

The importance of software quality, especially when this software when is used in critical applications, cannot be overestimated. Software correctness is considered an important contributor to software quality [1]. However, numerous difficulties exist in guaranteeing software to be correct [8]. The mCRL2 tool [4] provides a framework for the analysis and visualisation of concurrent systems and protocols. It can therefore be used as a means to model, validate and verify software. This can be done from a pure theoretical point of view but also in industrial applications (see for example [9]).

The basic workflow when using mCRL2 is the following: (part of) a certain system is (conceptually) isolated and a model is created using the mCRL2 specification language [5] (see for example [7] for an overview of this process). The resulting model can now be used for validation and verification using tools in the mCRL2 toolset. The most difficult, time-consuming and error-prone part of this process is the creation of the model. Therefore, it seems worthwhile to explore the possibilities of an automated conversion from the programs that make up the system to the mCRL2 specification language. Such a conversion, in a very restricted context, is the subject of this thesis. The research focus is on: *Exploring the possibilities of a (semi-)automated conversion from a subset of the Java programming language to the mCRL2 specification language.* The automation of a conversion from Java to mCRL2 would provide a distinct advantage over the current workflow. The resulting time-benefit and fault-tolerance would allow people to completely focus on the task of developing a certain system, instead of having to spend time on converting it to mCRL2.

In this research, only software-based systems are considered. A further restriction is made by focussing on a conversion from a subset of the Java programming language to mCRL2-input. This conversion is formalized to a certain extent but a complete proof of its correctness is omitted as this would be outside the scope of this research.

The reader is assumed to be familiar with the Java programming language. A short summary of process algebra, the formal base of the mCRL2 specification language, is provided in chapter 2. Chapter 3 describes the basics of the conversion from Java code to mCRL2 input. Chapter 4 extends this basic conversion by showing how to handle multiple threads and communication between simulated processes. Chapter 5 presents a short case study: a well-known algorithm is implemented in Java and an equivalent mCRL2 model is derived using the conversion steps presented in chapters 3 and 4. Chapter 6 contains a general roadmap section describing the conversion from Java to mCRL2 from a 'howto'-perspective. In chapter 7 the approach of this thesis is discussed and possible future work is suggested. This thesis is illustrated by code examples. For each example the programming language of the code listing is noted in *italics*.

I would like to thank Marko van Eekelen for supervising this thesis and keeping me on track. Jan Friso Groote and Yaroslav S. Usenko, both from the Eindhoven University of Technology, provided many useful tips and directions.

Allan van Hulst
November, 2007

Chapter 2

Process Algebra and mCRL2

This chapter provides a short introduction into process algebra. Process algebra is a formalism that can be used to describe systems in terms of their constituent operations and interactions. This introduction summarizes parts of[3]. Examples of corresponding mCRL2-input are given to illustrate the usage of process algebra as a practical process description tool.

2.1 Basic Process Algebra

Basic process algebra is a small framework that can be used to describe individual processes. The framework is based on the following definitions:

- The non-empty set A contains a set of *atomic* (indivisible) actions of the system. Examples of atomic actions include: reading a byte from input, incrementing an integer etc. The contents of A depends both on the system to be modeled and the chosen abstraction level. Each atomic action can be *executed* and it will *terminate* by definition. The label of this atomic action is used to describe the action as a *process term*.

  Some atomic actions have a specific predefined meaning, such as the action δ representing *deadlock* or stagnation, and the action τ which represents a *silent action* (abstract actions that have no specific identity). Intuitively, the action τ represents zero or more actions that can be ignored.

- A binary operator $+$ for *alternative composition* of processes. Assume process terms t_1 and t_2 represent processes p_1 and p_2 respectively, then the term $t_1 + t_2$ represents a single process that either executes p_1 or p_2. Intuitively, the operator $+$ can be seen as a non-deterministic choice between p_1 and p_2.

- A binary operator $\cdot$ for *sequential composition* of processes. Assume process terms t_1 and t_2 represent processes p_1 and p_2 respectively, then the term $t_1 \cdot t_2$ represents a single process that first executes p_1 followed by p_2 (if p_1 terminates).

Each finite process can be represented by a closed term that is build from the set A of atomic actions using the $+$ and $\cdot$ operators. Such terms are called *basic process terms* and the collection of all basic process terms is called the basic process algebra or BPA.

2.2 Algebra of Communicating Processes

The basic process algebra can be used to describe individual processes. However, most practical systems consist of multiple processes or can be modeled more precisely by assuming they are built out of more than one process. To describe such parallel behaviour, where the separate entities may influence each other's execution, the *algebra of communicating processes* is used. The basic process algebra is therefore extended by the following definitions:

- A binary operator $\parallel$ (called *merge*) which represents the parallel composition of two processes. Assume process terms t_1 and t_2 represent processes p_1 and p_2 respectively, then the term $t_1 \parallel t_2$ represents a single process which executes the individual actions of t_1 en t_2 interleaved. For example, $(a \cdot b) \parallel (c \cdot d)$ might be executed as $abcd, acbd, acdb, cabd, cadb, cdab$ but not as $bacd$ i.e. the inner sequence of process actions is preserved.

- A binary operator $\lfloor\!\lfloor$ (called *left merge*) which behaves just like the merge operator except that it takes the first action from the left argument always. For example, $(a \cdot b) \parallel (c \cdot d)$ might be executed as $abcd, acbd, acdb$ but not as $cabd$.

2.3 The mCRL2 specification language

There exists a simple correspondence between basic constructs in process algebra and mCRL2-input. The mCRL2 specification language consists of statements ended by a semi-colon. Below is a short summary of the mCRL2 specification language and its correspondence with process algebra:

- Atomic actions are declared using the `act` keyword. For example:

  ```
  act a, b, c, d;
  ```

  denoting atomic actions `a`, `b`, `c` and `d`. In mCRL2 atomic actions can be parameterised by data types referred to as *sorts*. Sorts, like `Bool` and `Nat` are explained in more detail later on. The parametrisation of actions makes it possible to define actions like `setswitch:Bool` and `write:Nat`.

The predefined actions δ and τ are denoted in mCRL2 as `delta` and `tau` respectively.

- Processes are declared using the `proc` keyword. For example:

  ```
  proc start = (a+b).c.d;
  ```

  Just like atomic actions, processes can be parameterised using sorts. Multiple data-parameters are allowed, so

  ```
  proc writeNumber(num:Nat,update:Bool) = tau;
  ```

  is a legitimate process definition.

- The operators $+$ and $\cdot$ are converted into their mCRL2-equivalents `+` and `.` respectively. Their semantics is exactly the same as in a process algebra contexts. However, with respect to the parallel operators, there are some differences between mCRL2 and the algebra of communicating processes. The mCRL2 syntax introduces a new operator `|` expressing true parallism, i.e. actions are executed in a real parallel fashion at the same time. The semantics of the process algebraic operator || is slightly modified and now includes both interleaving of individual actions and paralellism, therefore in mCRL2:

  $$\texttt{a||b} = \texttt{a.b + b.a + a|b}$$

- The initial process is denoted by the `init` keyword. For example:

  ```
  init start;
  ```

  Using appropriate parameters if this is required by the initial process.

Comments start with a percentage-sign (`%`) and extent to the end of the line.

Altough they are very similar concepts, it is important to distinguish between process algebra and the mCRL2 specification language. Process algebra serves as a theoretical framework while the mCRL2 specification language is used to implement actual models in process algebra. Because this research focusses on a conversion from Java to mCRL2, the process algebra syntax is not used hereafter and only mCRL2 syntax is used to avoid any confusion.
A few examples are given below to illustrate the syntax and semantics of the mCRL2 specification language. Further examples are given in the rest of this thesis. There is quite some material available on the mCRL2 toolset: research papers, technical guides and tutorials. See for example [6] for a more comprehensive introduction to mCRL2.

The first example introduces the actions `read` and `write` and a single process `P`. It also shows how to define recursive processes:

mCRL2 code

```
act read;
act write;

proc P = read.write.P;

init P;
```

The semantics of this example are very straightforward. A process `P` first performs a read, followed by a write action and calls itself again to repeat these actions indefinitely. The next example shows the choice operator (`+`) and the parallel operator (`||`) as well as parametrised actions. It also shows how parameters can be used in processes.

mCRL2 code

```
act output:Bool;
act answer:Int;

proc TrueOrFalseAndAnswer(b:Bool) =
  (output(true)+output(false))||answer(42);

init TrueOrFalseAndAnswer(true);
```

In this example the action `output` is either called with the value `true` or `false` and the action `answer` is called (in parallel) with the value `42`. Upon calling the initial process (`TrueOrFalseAndAnswer`) the parameter `b` is passed having the value `true`. The next example shows a specific mCRL2 construct called *choice*, denoted by B `->` P `<>` Q. Its semantics is rather straightforward: if B is true, then P is executed, otherwise Q. The `tau` and `delta` constructs are also shown:

mCRL2 code

```
proc A(b:Bool) = b -> delta <> tau;

init A(true);
```

In this example, when process `A` is started with the value `true`, the process enters deadlock (denoted by `delta`). If `A` is called with the value `false`, the `tau` is executed (having no effect at all). Another extensively used concept are the `comm` and `allow` constructs (used almost exclusively inside the `init` construct). The set denoted by `comm` contains actions that should be paired together and replaced by a third action. The intuition behind this is the following: Two actions are executed at the same time (because they are silently replaced by a third action) and these two actions are therefore allowed to communicate by exchanging zero or more parameters. Communicating actions should be explicitly allowed by the `allow` construct. This is shown in the following example:

mCRL2 code

```
act communicate:Nat;
act send:Nat;
act read:Nat;

init allow({communicate},comm({send|read->communicate},
  send(3) || sum m:Nat.read(m)));
```

In this example, the action `communicate` replaces paired `send` and `read` occurences and allows them to communicate a natural number. In the actual process (shown in the last line of the example), the natural number 3 is communicated. This example also shows the mCRL2 `sum` operator. Using this operator, all elements of a particular sort (in this case the natural numbers) can be enumerated and for each matching value (in this case 3) the right hand side of the `sum` operator (in this case `read(m)`) is executed.

2.4 Summary

In this short chapter, an introduction to process algebra was given covering both the basic process algebra and the algebra of communicating processes. The correspondence between the process algebra and the mCRL2 specification language was illustrated by examples. In the next chapter, a simple correspondence between Java and mCRL2 is defined. This correspondence is gradually extended in the following chapters.

Chapter 3

Basic Conversion

This chapter describes a conversion from Java to mCRL2 in its simplest form. A very basic subset of Java is described first. A correspondence between elements in this subset and mCRL2 constructs is defined. At first, this correspondence is only partial as several language elements are ignored. However, throughout the rest of this thesis, these elements are included in the language as well. The reason for using this approach is that this is a very illustrative way to show how the conversion is build up. The description of the Java-subset (called miniJava here) is followed by a simple correspondence between miniJava-constructs and mCRL2 in section 3.2. A conversion for functions which is safe for recursion is the subject of section 3.3. This section is followed by sections describing how various forms of variables such as function parameters can be converted as well.

3.1 Starting point of the conversion: miniJava

This chapter details a basic conversion from a subset of Java (called miniJava) to mCRL2 code. A grammar describing the subset of Java that is used as the source of the conversion is shown below. Some elements in this grammar (like using the `class` keyword) might seem unnecessary in this context but as this is a subset of Java they are not. The print statement is included here as an example of an atomic action.

```
<program>       =>  "public" "class" "Main" "{" <functionlist> "}"
<functionlist>  =>  <function><functionlist> | <null>
<function>      =>  "public" "boolean" <identifier> "(" ")" "{"
                    <statementlist> "}"
<statementlist> =>  <statementlist><statement> | <null>
<statement>     =>  <declaration> | <assignment> | <return> | <if> |
                    <print>
<declaration>   =>  "boolean" <identifier> ";"
<assignment>    =>  <identifier> "=" <expression> ";"
<return>        =>  "return" <identifier> ";"
```

```
<if>            =>  "if" "(" <identifier> ")" "{" <statementlist> "}"
                    <else>
<else>          =>  "else" "{" <statementlist> "}" | <null>
<print>         =>  "System.out.println" "(" <identifier> ")" ";"
<expression>    =>  <primary> | "!" <expression> |
                    <expression> && <expression> |
                    <expression> "||" <expression>
<primary>       =>  <identifier> | <identifier> "(" ")" | "true" |
                    "false"
<identifer>     =>  CHARACTER-SEQUENCE
<null>          =>  ""
```

The grammar above is a very restricted Java grammar. Apart from a few other constructs that will be added in chapter 4, this grammar defines the scope of this research. Boolean is the only variable type in miniJava because it is the simplest Java variable type. If other types were also used, the conversion would be too focused on the variables only and other important concepts would have to be discarded because of the narrow scope of this research. This restricted grammar describes Java programs only containing one class called `Main`. This class may contain any number of functions of return type `boolean`. Functions consist of any number of statements. A statement can be one of the following:

- A declaration of a `boolean` variable.
- An assignment of the resulting value of an expression to a variable.
- A conditional clause `if` only executing a list of statements if the value of a given variable is `true`. An if-clause may be followed optionally by an `else` clause which is executed when the variable in the corresponding if-clause evaluates to `false`.
- A return statement returning the value of a boolean variable.

For the purpose of simplification this is a very small subset of Java. Therefore, programs may be more lengthy when compared to their equivalents in full Java (i.e. the language is less rich in expressibility).

3.2 Converting miniJava statements to process algebra

Intuitively, the basic conversion from miniJava to process algebra to be described here ignores variable values. Later versions will include variable values but as a first step it is reasonable to ignore them. The result of the conversion is a process term describing all possible paths through the code. At first sight, such a conversion might be of little use, but consider the following example:

```
public class Main
```

```
{
  public static boolean f1 ()
  {
    boolean b1 = f2 ();

    if (b1)
    {
      boolean b2 = f3 ();

      f4 ();
    }
    else
    {
      boolean b3 = f5 ();

      f6 ();
    }
  }
}
```

When variable values are ignored, it can still be proven from this example that `f4 ()` is only executed after `f3 ()` is executed and that `f1 ()` will always preceed both `f3 ()` and `f5 ()`. Using such an approach the conversion function *conv* from miniJava to process algebra can be defined as follows, where $\mathtt{B}_n$, $\mathtt{S}_n$, $\mathtt{E}_n$ and $\mathtt{F}_n$ are used as meta-variables for booleans, statement groups, expressions and functions respectively. The *fbody* function denotes the body of a function.

1. A *statement group* refers to one or more statements meant to be executed sequentially (e.g. some part of the body of a function can be considered a statement group, this is not a very formal definition and it might be more suitable to define statements recursively but using statement group proves to be sufficient).

2. Only print statements are considered atomic actions in miniJava. Every print statement is converted to a suitable (i.e. more descriptive) process term. For example, `"System.out.println (b1);"` is used as `printb1` in process terms.

3. Let t_1 and t_2 be process terms representing statement groups $\mathtt{S}_1$ and $\mathtt{S}_2$ respectively. Then $t_1 \cdot t_2$ is the process term representing $\mathtt{S}_1\mathtt{S}_2$. In terms of the conversion function:

$$conv\ [\mathtt{S}_1\mathtt{S}_2] = conv\ [\mathtt{S}_1] \cdot conv\ [\mathtt{S}_2]$$

4. Let t_1 and t_2 be process terms representing statement groups $\mathtt{S}_1$ and $\mathtt{S}_2$ respectively. Then $t_1 + t_2$ is the process term representing $\mathtt{if(..)\{}\mathtt{S}_1\mathtt{\}else\{}\mathtt{S}_2\mathtt{\}}$.

$$conv\ [\mathtt{if(..)\{}\mathtt{S}_1\mathtt{\}else\{}\mathtt{S}_2\mathtt{\}}] = conv\ [\mathtt{S}_1] + conv\ [\mathtt{S}_2]$$

5. The process term describing an assignment is the same as the process term describing the expression on the right hand side in the assignment.

$$conv\ [\texttt{B = E}] = conv\ [\texttt{E}]$$

6. The results of the conversion function on expressions are:

$$conv\ [\texttt{!E}] = conv\ [\texttt{E}]$$

$$conv\ [\texttt{E}_1\ \texttt{\&\&}\ \texttt{E}_2] = conv\ [\texttt{E}_1] \cdot (conv\ [\texttt{E}_2] + \tau)$$

$$conv\ [\texttt{E}_1\ \texttt{||}\ \texttt{E}_2] = conv\ [\texttt{E}_1] \cdot (conv\ [\texttt{E}_2] + \tau)$$

$$conv\ [\texttt{F ()}] = conv\ [\mathit{fbody}\ (\texttt{F})]$$

$$conv\ [\texttt{B}] = \tau$$

7. Because declarations cannot be combined with assignments in miniJava, the process term corresponding to a declaration is the silent action τ.

$$conv\ [\texttt{boolean B;}] = \tau$$

8. The same holds for return statements.

$$conv\ [\texttt{return B;}] = \tau$$

It is important to note that this is is not the final conversion from Java to mCRL2. This is only a first step to a proposed conversion function. The miniJava grammar and the conversion will be modified later on. The steps leading to the final definition of the Java to mCRL2 conversion are shown here to illustrate the way the final result was established.

3.3 The conversion of functions: enabling recursion

The section above dealt with the conversion of miniJava-statements into process algebra constructs. However, miniJava has, like any realistic programming language, a function mechanism which enables recursion. In the previous definition of *conv* for functions, the contents of the function was expanded. Now the conversion of functions is modified to include recursion. The most straightforward conversion from these functions to process algebra would be to convert each function into a (callable) process:

$$conv\ [\texttt{public static boolean}\ F\ \texttt{()}\ \{S\}] = \texttt{proc}\ F\ \texttt{=}\ conv\ [S]$$

Now functions (converted into processes) can be referenced. The most obvious conversion for function-calls now becomes the direct execution of the process. This replaces the conversion by code-expansion as it was defined before.

$$conv\ [F\ \texttt{()}] = F$$

Because functions are now handled as processes, they can be referenced and recursive expressions can be defined. This is shown in the following simple example which can be shown to always print `true`:

Java code

```
public static void
    main (String [] argv) {
  f1 ();
}
public static boolean f1 () {
  System.out.println (true);
  f2 ();
  System.out.println (false);
  return false;
}

public static boolean f2 () {
  System.out.println (true);
  f1 ();
  return false;
}
```

mCRL2 code

```
act print:Bool;

proc f1 = print(true).f2.
          print(false);
proc f2 = print(true).f1;

init f1;
```

Because each Java function is now converted into a separate mCRL2 process, functions are callable and need not be translated into mCRL2 by expanding their contents. This replaces the previous definition of function conversion. This step enables recursion because processes can be referenced.

3.4 Next step: Adding decisiveness

The conversion as defined above ignores variable values. A process algebra expression in this context was meant to describe all possible paths through the code. However, this non-deterministic interpretation of Java code has little correspondence to reality. It is therefore obvious that a completely deterministic mapping from miniJava to process algebra has to be defined because it would not be very useful in practice otherwise.

Intuitively, the conversion of the if-statement was defined in the previous section as: discard the boolean expression, either the if-part or the else-part might be executed now. But to change this into a deterministic interpretation, the boolean expression is now incorporated into the conversion. The if-part is executed when the boolean expression evaluates to true and the else-part is executed

otherwise. This results in the following conversion:

$$conv\,[\texttt{if (}E\texttt{)\{}S_1\texttt{\}else\{}S_2\texttt{\}}] = conv\,[E]\ \texttt{->}\ conv\,[S_1]\ \texttt{<>}\ conv\,[S_2]$$

In the previous section, the conversion of the boolean expression itself was defined to capture all possible ways the expression could be executed (e.g. in the execution of `A && B`, the execution of `B` might be omitted if `A` evaluates to `false` etc.). However, in the deterministic conversion there is only one correct way these boolean expressions can be interpreted. Therefore, the now following modifications to the conversion function are made, where E and E_n are boolean expression and B stands for a boolean variable.

$$conv\,[\texttt{!}E] = \texttt{!}\,conv\,[E]$$

$$conv\,[E_1\ \texttt{\&\&}\ E_2] = conv\,[E_1]\ \texttt{\&\&}\ conv\,[E_2]$$

$$conv\,[E_1\ \texttt{||}\ E_2] = conv\,[E_1]\ \texttt{||}\ conv\,[E_2]$$

$$conv\,[B] = B$$

The following listing illustrates the application of the new conversion to a simple example. Assume the software in the control unit of a printer contains a function `userFeedBack`. The purpose of this function is to indicate a low level of ink or a small amount of paper using an orange warning LED. When both the level of ink and amount of paper are low, a red warning LED is used while a green LED indicates normal operation.

Java code

```
public static boolean
  userFeedBack (
    boolean inklow,
    boolean paperlow) {

  setGreenLED (false);
  setRedLED (false);
  setOrangeLED (false);

  if (inklow && paperlow) {
    setRedLED (true);
  } else {
    if (inklow) {
      setOrangeLED (true);
    } else {
      if (paperlow) {
        setOrangeLED (true);
      } else {
        setGreenLED (true);
      }
    }
  }

}
```

mCRL2 code

```
act setRedLED:Bool;
    setOrangeLED:Bool;
    setGreenLED:Bool;

proc userFeedBack
    (inklow:Bool,paperlow:Bool) =
 setGreenLED(false).
 setOrangeLED(false).
 setRedLED(false).
 (inklow && paperlow ->
   setRedLed(true)
   <>
   (inklow ->
      setOrangeLED(true)
      <>
      (paperlow ->
         setOrangeLED(true)
         <>
         setGreenLED(true))));

% Example usage: low on ink
init userFeedBack(true,false);
```

This example shows a direct conversion from Java to mCRL2 code. The conversion is in fact very straightforward because it can be achieved by repeated and recursive application of the conversion function *conv* as explained before. One thing not mentioned in the text but shown in the example is the conversion of function parameters. This specific conversion and the modeling of the return-values of functions is the subject of the next section.

To summarize up until this point: the conversion of simple Java constructs has been defined. In a very simple way first: ignore all variable values. The next step was to enable recursion: the conversion function was modified to include recursion by converting each Java function into a separate mCRL2 process. In the following step, the conversion for boolean expressions in if-statements was defined, thereby introducing decisiveness. The now following section explains how to convert parameters and return values in functions.

3.5 Extending the conversion of functions: parameters and return values

Modeling functions as separate processes allows a relatively easy conversion of function parameters. The list of parameters in the Java-function is directly converted into the mCRL2-syntax using the appropriate data types. In this stage, only variables of type `boolean` are allowed so these can be mapped directly to the `Bool` sort in mCRL2 syntax:

$$conv\ [\texttt{public static boolean}\ F\ \texttt{(boolean}\ B_1, ...,\texttt{boolean}\ B_n\texttt{)}\ \{S\}] =$$

$$\texttt{proc}\ F(B_1\texttt{:Bool}, ..., B_n\texttt{:Bool})\ =^1 conv\ [S]$$

However, such a straightforward conversion does not exist when it comes to return values of functions. In both process algebra and mCRL2 there is no such thing as a 'resulting value' defined for a process. The intuitive approach when function F calls function G (and uses its return value) is the following:

1. Convert both the contents of F and G into mCRL2 expressions.
2. Set up a process algebraic communication function that communicates the return value of G back to F.
3. Assure such communication functions are properly set up and 'allowed' to communicate. This can be done by appropriate use of the mCRL2 `allow` and `comm` constructs as shown below.

With respect to (1): the conversion of a function into a callable process and the conversion of individual statements inside a function into their equivalent process algebraic constructs has been described before. For setting up the communication function mentioned in (2), the following actions are created:

```
act return:Bool;
act obtain:Bool;
act transfer:Bool;
```

And the `init` construct is extended with the following code:

```
init allow({transfer},comm({return|obtain->transfer},main));
```

Where `main` refers to the initial process. Returning the value of a function now becomes particularly easy. The action `return` can be used, using the expression that makes up the return value as an argument. The situation at the calling context is more complicated: the function is called using the process name and the return value is obtained using the mCRL2 `sum` operator and the action `obtain`. For example:

[1] Part of syntax

`f || sum r:Bool.obtain(r)`

Which allows the value returned by the process `f` to be used as `r` at the right side of the `||` operator.

The next step is the formalisation of the steps described before by modification of the conversion function *conv*. Besides this, other modifications to the resulting output have to be made including the modification of the `init` construct and the addition of the actions `return`, `obtain` and `transfer`. The precise description and formalisation of these are omitted here because they have been described before. What remains are the instances of the conversion function for returning and obtaining values respectively:

$$conv\ [\texttt{return}\ \ E\texttt{;}] = \texttt{return(}conv\ [E]\texttt{)}$$

$$conv\ [F\ \ \texttt{()}] = F\ \ \texttt{|| sum r:Bool.obtain(r)}$$

Note that the conversion on the side of the called function is remarkably easy as the Java and mCRL2 code are very similar.

The mCRL2 implementation of Java return values, as it is defined here, might seem to involve a lot of overhead in lines of resulting mCRL2 code. Other approaches are very well possible. For instance, not to convert Java functions into mCRL2 processes but to use the possibility of defining mathematical functions in mCRL2. Return values are defined for these kinds of functions but they do not allow any side effects. Processes do allow side effects so the difficulty of converting return values is taken for granted here.

Here is an example of the conversion of Java code including function calls, parameters and return values.

Java code

```
public static void main
    (String [] argv) {
  if (f (false)) {
    System.out.println (true);
  }
}

public static boolean f
    (boolean p) {
  System.out.println (false);
  if (p) {
    System.out.println (true);
  } else {
    System.out.println (false);
  }
  return true;
}
```

mCRL2 code

```
act return:Bool;
act obtain:Bool;
act transfer:Bool;
act print:Bool;

proc main =
  (f(false) ||
    sum r:Bool.obtain(r).r ->
    f(true)) ||
  sum s:Bool.obtain(s).print(s);

proc f(p:Bool) =
  print(false).
    (p -> print(true) <>
     print (false)).
     return(true);

init allow({print,transfer},
       comm(
       {return|obtain->transfer},
       main));
```

As this example shows, the conversion of function parameters and return values adds some significant complexity. To make some of the other examples in this thesis more clear, return values of functions are omitted sometimes. When this is the case, it is explicitly stated.

3.6 Summary

At the start of this chapter, a grammar of a very restricted subset of Java was presented and explained. The chapter continued by showing how a very simple conversion from these grammar elements to corresponding mCRL2 constructs could be defined. This simple conversion was later extended to allow functions to be used recursively (by converting Java functions into mCRL2 processes). In the next step, variable values are no longer ignored and a proper conversion was defined for them too. The last section completed the coverage of functions by defining a conversion for function parameters and return values. The next chapter describes how to convert simple multi-threaded Java-programs into their mCRL2-equivalents.

Chapter 4

Multithreading

In this chapter, the conversion defined in chapter 3 is extended. The Java threading model is explained by examples in section 4.1. It is explained how a very straightforward conversion for threads can be defined by using transformations on both the Java and mCRL2 side. Global boolean variables are used as a communication primitive between threads. The proposed conversion for boolean variables can be used in other cases as well. In the last section, the conversion function is applied on a short example to illustrate how multithreaded Java code is converted into mCRL2.

4.1 Adapting the Java threading model

From its early design on, the Java programming language was designed to be multi-threaded. It offers a clearly specified and relatively easy way to create multi-threaded programs by using inheritence in the class system. Code that needs to run as a separate thread is encapsulated inside a class that inherits from (i.e. is a subclass of) the `Thread` class. The latter also contains platform-independent code to create, start, stop and otherwise manipulate threads. Below is a short example of multi-threaded Java code.

Java code

```
public class ThreadTest extends Thread {
  public static void main (String [] argv) {
    ThreadTest tt = new ThreadTest ();
    tt.start ();
  }

  public void run () {
    System.out.println ("Code in thread is now running");
  }
```

```
}
```

As the example shows, a class `ThreadTest` is made a subclass of the system class `Thread`. The method `run` is overloaded and redefined. The `run` method is the starting point of the thread. That is, if the new thread is created and started, the code in the `run` method is the first to be executed.

In the previous chapter, a subset of Java called miniJava was defined. It contained only the basic constructs in the Java language and did not include any language features related to multithreading. However, the Java threading system depends on multiple more advanced elements in the Java language such as classes, inheritence and non-boolean functions. One possible solution would be to support these constructs and to define a conversion function for a more broader version of miniJava including classes, inheritence etc. But this solution would lead to a vast amount of overhead and it does not capture the essence of converting multi-threaded Java-programs into equivalent process algebra. Therefore, another solution seems to be more appropriate. Consider the following Java code which starts two threads.

Java code

```
public class Main {
  public static void main (String [] argv) {
    (new Thread(){public void run(){Main.runThreadCode1();}}).start ();
    (new Thread(){public void run(){Main.runThreadCode2();}}).start ();
  }

  public static boolean runThreadCode1 () {
    System.out.println ("Code in thread 1 is now running");
    return true;
  }

  public static boolean runThreadCode2 () {
    System.out.println ("Code in thread 2 is now running");
    return true;
  }
}
```

As the example shows, for each thread, it takes only one line of code to start it. This is shown in the 3rd and 4th line of the example. This specific pattern is included as a new statement in an extended version of miniJava. This approach overcomes the obvious difficulties that occur when classes, memory

allocation etc. need to be converted into mCRL2 code. A possible drawback is that the Java code before the conversion can be applied has to be rewritten to this specific format. However, this should not be a major obstacle since Java conversions are far simpler than conversion from Java to mCRL2. Another advantage of this approach is that miniJava remains a valid subset of Java but now includes multithreading. Note that the function `run` in the anonymous class is interpreted as part of the syntax of the statement that starts the thread. It is not interpreted as a method when converting these statements into mCRL2 code; only public static boolean methods are allowed. Threads are have to be included in the miniJava grammar. Therefore, a number of elements in the grammar have to be changed. The modified grammar now becomes (unchanged elements not shown):

```
<statement>     =>  <declaration> | <assignment> | <return> | <if> |
                    <print> | <threadstart>
<threadstart>   =>  "(new Thread(){public void run(){Main."
                    <identifier> "();}}).start ();"
```

Threads occur often in both Java and concurrent systems. Java threads can be started using only one line of code. To avoid the conversion of very complex constructs (e.g. memory management) this line is taken as if it were part of the Java syntax. Note that the new version of miniJava remains a subset of Java (from a syntactical point of view).

4.2 Converting multi-threaded code into mCRL2

The concept of a thread in miniJava resembles the concept of a process in mCRL2 which seems a good argument to convert miniJava threads into separate processes. Because functions are also converted into processes (as explained in the previous chapter) this is precisely what is required because miniJava threads also require the starting point of a thread to be a function. To convert the `<threadstart>` statement (see miniJava grammar) into mCRL2 code, the conversion function is extended in the following way:

$$conv\,[\texttt{(new Thread()\{public void run()\{Main.}\;\; F \;\texttt{();\}\}))).start()}] = F \;\texttt{||}$$

Where F refers to the function that contains the code that needs to be run in a separate thread. The purpose of the converted mCRL2 code is to make F (converted into a callable process) run parallel to any statements that follow the `<threadstart>` statement. Please note that this statement will never be the last statement in a function. Because only boolean functions are allowed, the last statement in a function will always be a `return` statement. If this was not the case, the conversion into F `||` would not have made much sense because there would be no trailing statement. A simple example of multi-threaded miniJava-code and its mCRL2 equivalent is provided below:

Java code

```
public static void main
    (String [] argv) {
  m ();
}

public static boolean m () {
  (new Thread(){public void
    run(){t1();}}).start ();
  (new Thread(){public void
    run(){t2();}}).start ();
  return true;
}

public static boolean t1 () {
  System.out.println (true);
  System.out.println (true);
  System.out.println (true);
  return true;
}

public static boolean t2 () {
  System.out.println (false);
  System.out.println (false);
  System.out.println (false);
  return false;
}
```

mCRL2 code

```
act print:Bool;
act return:Bool;

proc m = t1 || t2 ||
          return(true);

proc t1 = print(true).
          print(true).
          print(true).
          return(false);

proc t2 = print(false).
          print(false).
          print(false).
          return(false);

init m;
```

For the purpose of clearity, some of the overhead involving returning the value from a function and obtaining the result is omitted in the example above. The complete example is included in section A.1 in the Appendix.

4.3 Communication between threads

In any realistic example, there is always some form of communication between threads. A number of communication primitives exist, for example: memory mappings, semaphores or even more sophisticated constructs. However, any of these can be expressed in terms of shared variables. Programs in the miniJava language are allowed to contain only one class (`Main`) so any shared variable should be accessible from within this class. Therefore, the most obvious extension of the miniJava language would be to allow `public static boolean` variables in the `Main` class and to model them as global variables. These variables

are accessible within every function in `Main` and therefore they are accessible within every thread. The miniJava grammar is extended in the following way (omitting unchanged elements):

```
<declaration>      =>  <function> | <shared>
<shared>           =>  "public" "static" "boolean" <identifier> ";"
```

The grammar is extended with a `<shared>` keyword, introducing a shared variable. To model such shared variables in mCRL2 is far from easy. The most naive and simplistic approach might be to ignore each statement where a shared variable is set and to define two expressions when a shared variable is read (one for reading a `false` value and one for reading a `true` value). It is obvious that such a conversion does not add any value and is equivalent to handle shared variables as having random values. The purpose of this section is to model communication between threads, the variable values are therefore important to achieve a deterministic solution.

Altough boolean variables are introduced here as a means to achieve communication between threads, variables of course have many other uses in Java and the same conversion can be applied when boolean variables are not used for passing values between threads.

The mCRL2 syntax does not allow any global variables (or a similar construct). Shared variables therefore need to be modelled using processes. The idea is to define a process for each shared variable. The purpose of this process is to keep hold of the variable value during the evaluation of the process model. By means of process communication, the shared boolean variable can be set, reset and its value can be obtained. The following mCRL2 example illustrates this (Note: this model is not derived from any corresponding miniJava code but only serves to illustrate a possible mCRL2 approach for modeling boolean variables).

mCRL2 code

```
act setVarside;
act setCallside;
act setOperation;

act resetVarside;
act resetCallside;
act resetOperation;

act getVarside:Bool;
act getCallside:Bool;
act getOperation:Bool;

act print:Bool;
```

```
proc BoolVariable(value:Bool) =
      setVarside.BoolVariable(true) +
      resetVarside.BoolVariable(false) +
      getVarside(value).BoolVariable(value);

proc Thread1 = (sum b:Bool.getCallside(b).print(b)).
               setCallside.
               (sum b:Bool.getCallside(b).print(b));

proc Thread2 = (sum b:Bool.getCallside(b).print(b)).
               resetCallside.
               (sum b:Bool.getCallside(b).print(b));

init  allow ({setOperation, resetOperation, getOperation, print},
        comm({setVarside | setCallside -> setOperation,
              resetVarside | resetCallside -> resetOperation,
              getVarside | getCallside -> getOperation},
          Thread1 || Thread2 || BoolVariable(false)));
```

The *xxx*`Varside` actions are only used within the process that simulates the boolean variable (called `BoolVariable` in the example) while the *xxx*`Callside` actions are meant to be used outside this process. The `set`*XXX*, and `reset`*XXX* actions are coupled in communication functions *xxx*`Operation` using the `comm` operator. The `BoolVariable` process is started parallel to the other processes.

To convert shared variables in miniJava code into mCRL2 syntax similar to the example above, the conversion function *conv* has to be extended in the following way:

conv [`public static boolean` *Identifier*;] =

```
act setIdentifierVarside;
act setIdentifierCallside;
act setIdentifierOperation;
act resetIdentifierVarside;
act resetIdentifierCallside;
act resetIdentifierOperation;
act getIdentifierVarside:Bool;
act getIdentifierCallside:Bool;
act getIdentifierOperation:Bool;

proc Identifier(value:Bool) =
 setIdentifierVarside.Identifier(true) +
```

reset*Identifier*Varside.*Identifier*(false) +
get*Identifier*Varside(value).*Identifier*(value);

The ***xxx***Operation actions are added to the allow set specified in init. The set specified in comm is united with the following set:

{set*Identifier*Varside|set*Identifier*Callside->set*Identifier*Operation,
reset*Identifier*Varside|reset*Identifier*Callside->reset*Identifier*Operation,
get*Identifier*Varside|get*Identifier*Callside->get*Identifier*Operation}

What remains is the conversion of the assignment: a value can be assigned to a shared boolean variable. The conversion function *conv* is therefore extended in the following way:

$$conv\ [Identifier = \texttt{true}] = \texttt{set}Identifier\texttt{Callside}$$

$$conv\ [Identifier = \texttt{false}] = \texttt{reset}Identifier\texttt{Callside}$$

This completes the definition of the conversion for boolean variables. A process is defined for each boolean variable to keep hold of the value of the variable. Actions are defined to set, reset and read the value of the variable. Using the appropriate get-action, variable values can be read and used inside the code.

4.4 Result of the conversion function on a short example

A larger case study on the miniJava to mCRL2 conversion is the subject of the next chapter. Here is a shorter example of an imaginative printer driver system. Assume there are two threads in the printer driver system. The first thread continuously prints the next page, if there is a next page available. The second thread continuously monitors the paper and ink levels in the printer. If paper and ink are below certain levels, a warning light is shown and the printer is not allowed to print the next page until paper and ink levels are readjusted to normal. The Java code for this example is shown below. Please note that this program can only be used for illustrative purposes because it uses unguarded recursion.

miniJava code

```
public class Main {
  public static boolean inklow;       // true if ink low
  public static boolean paperlow;     // true if paper low
  public static boolean warning;      // true if warning issued

  public static void main (String [] argv) {
```

```
    m ();
  }

  public static boolean m () {
    (new Thread(){public void run(){printer();}}).start ();
    (new Thread(){public void run(){checker();}}).start ();
    return true;
  }

  public static boolean printer () {
    if (!warning) {
      printNextPage ();
    }
    printer ();
    return true;
  }

  public static boolean checker () {
    if (inklow || paperlow) {
      warning = true;
    } else {
      warning = false;
    }
    checker ();
    return true;
  }

  public static boolean printNextPage () {
    // print the next page..

    return true;
  }
}
```

The corresponding mCRL2 code is shown below. As was done in previous examples, unused return values were not converted to help make this example more clear.

mCRL2 code

```
% printPage action
act printPage;

% boolean variable InkLow
```

```
act setInkLowVarside;
act setInkLowCallside;
act setInkLowOperation;

act resetInkLowVarside;
act resetInkLowCallside;
act resetInkLowOperation;

act getInkLowVarside:Bool;
act getInkLowCallside:Bool;
act getInkLowOperation:Bool;

proc InkLow(value:Bool) =
  setInkLowVarside.InkLow(true) +
  resetInkLowVarside.InkLow(false) +
  getInkLowVarside(value).InkLow(value);

% boolean variable PaperLow
act setPaperLowVarside;
act setPaperLowCallside;
act setPaperLowOperation;

act resetPaperLowVarside;
act resetPaperLowCallside;
act resetPaperLowOperation;

act getPaperLowVarside:Bool;
act getPaperLowCallside:Bool;
act getPaperLowOperation:Bool;

proc PaperLow(value:Bool) =
  setPaperLowVarside.PaperLow(true) +
  resetPaperLowVarside.PaperLow(false) +
  getPaperLowVarside(value).PaperLow(value);

% boolean variable Warning
act setWarningVarside;
act setWarningCallside;
act setWarningOperation;

act resetWarningVarside;
act resetWarningCallside;
act resetWarningOperation;

act getWarningVarside:Bool;
act getWarningCallside:Bool;
```

```
act getWarningOperation:Bool;

proc Warning(value:Bool) =
  setWarningVarside.Warning(true) +
  resetWarningVarside.Warning(false) +
  getWarningVarside(value).Warning(value);

% start two threads
proc m = printer || checker;

% printer thread
proc printer = getWarningCallside(warning).
               (warning -> tau <> printNextPage).printer;

% checker thread
proc checker = getInkLowCallside(inklow).
               getPaperLowCallside(paperlow).
               ((inklow || paperlow) ->
                 setWarningCallside <> resetWarningCallside).
               checker;

proc printNextPage = printPage;

% initialization
init allow({setInkLowOperation,resetInkLowOperation,
            getInkLowOperation,setPaperLowOperation,
            resetPaperLowOperation, getPaperLowOperation,
            setWarningOperation, resetWarningOperation,
            getWarningOperation, printPage},
       comm({setInkLowVarside|setInkLowCallside->
               setInkLowOperation,
             resetInkLowVarside|resetInkLowCallside->
               resetInkLowOperation,
             getInkLowVarside|getInkLowCallside->
               getInkLowOperation,
             setPaperLowVarside|setPaperLowCallside->
               setPaperLowOperation,
             resetPaperLowVarside|resetPaperLowCallside->
               resetPaperLowOperation,
             getPaperLowVarside|getPaperLowCallside->
               getPaperLowOperation,
             setWarningVarside|setWarningCallside->
               setWarningOperation,
             resetWarningVarside|resetWarningCallside->
               resetWarningOperation,
             getWarningVarside|getWarningCallside->
```

```
  getWarningOperation},
m || InkLow(false) || PaperLow(false) ||
  Warning(false)));
```

4.5 Summary

In this chapter, the miniJava grammer was extended by adding new constructs for multithreading and shared variables. Using these constructs, it is now possible to convert simple - yet complete - multithreaded Java programs into corresponding mCRL2 code because both multiple threads and communication functions can be handled now. However, more than in the previous chapter it might be necessary to rewrite the Java code before the conversion can be applied. When ready for conversion, each thread is converted into a separate mCRL2 process. For each shared boolean variable, a process is defined including operations to set, reset and get the value of this variable. In the last section of this chapter, a short example was given to illustrate the result of the conversion function.

Chapter 5

Case study

Using the definitions in the previous chapters, the conversion function is now applied on a real example to convert an algorithm implemented in Java to a suitable mCRL2 model. For clearity, the complete process from Java to mCRL2 code is illustrated. The example used in this case study is a well-known mutual exclusion algorithm designed by the Dutch mathematician Dekker. After the application of the conversion function, it is shown how properties in the resulting mCRL2 model can be verified.

5.1 Dekker's algorithm

The algorithm, often referred to as "Dekker's algorithm", can be expressed in pseudo-code as follows:

```
bool f0 := false,f1 := false
int turn := 0

f0 := true                          f1 := true
while f1 {                          while f0 {
  if turn != 0 {                      if turn != 1 {
    f0 := false                         f1 := false
    while turn != 0 { }                 while turn != 1 { }
    f0 := true                          f1 := true
  }                                   }
}                                   }

/* critical section */              /* critical section */

turn := 1                           turn := 0
f0   := false                       f1   := false
```

This algorithm is implemented in Java using one class and two threads to model the processes. The Java code is shown below. Because of the obvious similarities between the pseudo-code and Java, it is assumed that the Java code implements the same algorithm. Extensive testing confirms this. The Java code only uses boolean variables and a critical section is modelled by printing a number of boolean values sequentially. The first thread prints three times the value `true` while the second thread prints three times the value `false`. Due to Dekker's algorithm, these print statements cannot interleave with each other (e.g. `true true true false false false` is correct while `true false true false false true` is an example of wrong output). This property will later be used in mCRL2 to prove that the critical section is valid.

Java code

```
/**
 * Dekker.java
 *
 * Implementation in Java of the algorithm derived by Dutch
 * mathematician Dekker for mutual exclusion.
 *
 * @author Allan van Hulst
 */

public class Dekker
{
  private static boolean f0   = false;
  private static boolean f1   = false;
  private static boolean turn = false;

  public static void main (String [] argv)
  {
    (new Thread(){public void run(){processA();}}).start ();
    (new Thread(){public void run(){processB();}}).start ();
  }

  public static boolean processA ()
  {
    f0 = true;

    while (f1)
    {
      if (turn != false)
      {
```

```
      f0 = false;

      while (turn != false)
      {

      }

      f0 = true;
    }
  }

  // Start of critical section
  System.out.println (true);
  System.out.println (true);
  System.out.println (true);

  turn = true;
  f0 = false;

  return true;
}

public static boolean processB ()
{
  f1 = true;

  while (f0)
  {
    if (turn != true)
    {
      f1 = false;

      while (turn != true)
      {

      }

      f1 = true;
    }
  }

  // Start of critical section
  System.out.println (false);
  System.out.println (false);
  System.out.println (false);
```

```
    turn = false;
    f1 = false;

    return true;
  }
}
```

5.2 Converting Java to miniJava

For the conversion function to be applied on the example, a number of changes have to be made to the code. The conversion function *conv* defined in the previous chapters can only be applied to a subset of Java: miniJava. No conversion for while loops was defined in the previous chapters. So an equivalent miniJava program based only on recursion is derived from the original code. This converted program is shown below.

Java code

```
public class Main {
  public static boolean f0   =  false;
  public static boolean f1   =  false;
  public static boolean turn =  false;

  public static void main (String [] argv) {
    m ();
  }

  public static boolean m () {
    (new Thread(){public void run(){processA();}}).start ();
    (new Thread(){public void run(){processB();}}).start ();

    return true;
  }

  public static boolean processA () {
    f0 = true;

    outerLoopProcessA ();

    System.out.println (true);
    System.out.println (true);
```

```
  System.out.println (true);

  turn = true;
  f0 = false;

  return true;
}

public static boolean processB () {
  f1 = true;

  outerLoopProcessB ();

  System.out.println (false);
  System.out.println (false);
  System.out.println (false);

  turn = false;
  f1 = false;

  return true;
}

public static boolean outerLoopProcessA ()
{
  if (f1) {
    if (turn != false) {
      f0 = false;
      innerLoopProcessA ();
      f0 = true;
    }
    outerLoopProcessA ();
  }
  return true;
}

public static boolean outerLoopProcessB ()
{
  if (f0) {
    if (turn != true) {
      f1 = false;
      innerLoopProcessB ();
      f1 = true;
    }
    outerLoopProcessB ();
  }
```

```
    return true;
  }

  public static boolean innerLoopProcessA () {
    if (turn != false) {
      innerLoopProcessA ();
    }
    return true;
  }

  public static boolean innerLoopProcessB () {
    if (turn != true) {
      innerLoopProcessB ();
    }
    return true;
  }
}
```

Again it is assumed that this program implements Dekker's algorithm in the correct way. A number of changes is made to prepare the Java code for conversion to mCRL2: A new method `m ()` is introduced which is directly called by `main`, the two while loops are replaced by `outerLoopProcessA` / `outerLoopProcessB` and `innerLoopProcessA` / `innerLoopProcessB` respectively. The name of the class is changed into `Main`.

Rewriting Java into basic constructs is unavoidable when converting Java to mCRL2 using the approach described in this thesis. As described before, this is a two step approach: the first step is to convert Java into miniJava and the second step is to convert miniJava into mCRL2. Only the second step is the subject of this thesis. The miniJava grammar can be used as a guide to convert Java into miniJava. A certain fragment of Java code may have multiple equivalent miniJava translations resulting in different mCRL2 models. However, such models should be equivalent because they all originate from the same Java source and only semantics preserving transformations are applied.

5.3 Converting Dekker's algorithm from Java to mCRL2

The application of the conversion function is now outlined step by step. The Java source code file is traversed from top to bottom and each section is converted into mCRL2-code using the conversion function *conv* as defined in the previous chapters. This conversion includes the following steps:

1. The `Main` class is discarded because the conversion function in its current form is only defined for a single class (which should be called `Main`) but no further conversions at the class-level need to be applied except for the definition of a single action `act print:Bool;`.

2. Next step is to convert boolean variables `f0`, `f1` and `turn` into corresponding mCRL2-code. This is explained here for the boolean variable `f0`. The cases `f1` and `turn` proceed accordingly. First step in the conversion of boolean variables is to set up three times three actions; for the operations set, reset and get respectively:

```
act setf0Varside;
act setf0Callside;
act setf0Operation;

act resetf0Varside;
act resetf0Callside;
act restef0Operation;

act getf0Varside:Bool
act getf0Callside:Bool
act getf0Operation:Bool
```

   The essential part of the mCRL2-realisation of this boolean variable is a process named `f0` defined as follows:

```
proc f0(value:Bool) =
    setf0Varside.f0(true) +
    resetf0Varside.f0(false) +
    getf0Varside(value).f0(value);
```

   The last step in the conversion of this boolean variable is to extend the definition of the mCRL2 `init` construct. In the definition of init:

   `init allow (`*Allow*`, comm (`*Comm*`,` *Start*`))`

   The set *Allow* has to be united with {`setf0Operation, resetf0Operation, getf0Operation`} and the set *Comm* has to be united with {`setf0Varside | setf0Callside -> setf0Operation, resetf0Varside | resetf0Callside -> resetf0Operation, getf0Varside | getf0Callside -> getf0Operation`}. The process *Start* is extended with a parallel statement and the process name of the variable. Assuming no previous variables were converted, this results in: `m || f0`. The very same techniques can be used to convert the boolean variables `f1` and `turn` into mCRL2 code.

3. The conversion of the `main` function is very straightforward because it is only allowed to contain one method call `m ();`. If such a `main` function is present in the program, no further conversions need to be applied.

4. The first boolean function to be converted is the function `m`. It contains two statements in which a thread is started and one return-statement. Therefore, the function `m ()` is converted into the following mCRL2 code:

   ```
   proc m = (processA || processB);
   ```

5. The next functions to be converted are similar for processes A and B. However, they are not exactly the same. The conversion steps are explained in detail for process A, and summarized for process B. The function `processA ()` does not need much explanation. Its conversion is relatively straightforward:

   ```
   proc processA = setf0Callside.outerLoopProcessA.
                   print(true).print(true).print(true).
                   setturnCallside.resetf0Callside;
   ```

   The difficult parts in the conversion of `outerLoopProcessA` are the two if-statements. The general rule for converting these if-constructs where a boolean variable is tested is the following:

   `sum b:Bool.get`*Identifier*`Callside(b).(b ->` *IfPart* `<>` *ElsePart*`)`
   Where *ElsePart* is replaced by `tau` because no such else-part is present in the function `outerLoopProcessA`. The resulting conversion becomes the following:

   ```
   proc outerLoopProcessA = sum b:Bool.getf1Callside(b).
                            (
                              b -> (sum c:Bool.getturnCallside(c).
                                     (c -> resetf0Callside.innerLoopProcessA.
                                           setf0Callside <>
                                      tau).
                                     outerLoopProcessA) <>
                                    tau
                            );
   ```

   where `innerLoopProcessA` is converted in the following way:

   ```
   proc innerLoopProcessA = sum b:Bool.getturnCallside(b).
                            (b -> innerLoopProcessA <> tau);
   ```

   The conversion steps of process B are similar to those of A except for variable names and values.

The last step in the conversion process is to assemble the pieces of mCRL2 code together into suitable process algebra expressions. The resulting mCRL2 code is shown below:

mCRL2 code

```
act print:Bool;

act setf0Varside;
act setf0Callside;
act setf0Operation;

act resetf0Varside;
act resetf0Callside;
act resetf0Operation;

act getf0Varside:Bool;
act getf0Callside:Bool;
act getf0Operation:Bool;

proc f0(value:Bool) =
    setf0Varside.f0(true) +
    resetf0Varside.f0(false) +
    getf0Varside(value).f0(value);

act setf1Varside;
act setf1Callside;
act setf1Operation;

act resetf1Varside;
act resetf1Callside;
act resetf1Operation;

act getf1Varside:Bool;
act getf1Callside:Bool;
act getf1Operation:Bool;

proc f1(value:Bool) =
    setf1Varside.f1(true) +
    resetf1Varside.f1(false) +
    getf1Varside(value).f1(value);

act setturnVarside;
act setturnCallside;
act setturnOperation;
```

```
act resetturnVarside;
act resetturnCallside;
act resetturnOperation;

act getturnVarside:Bool;
act getturnCallside:Bool;
act getturnOperation:Bool;

proc turn(value:Bool) =
    setturnVarside.turn(true) +
    resetturnVarside.turn(false) +
    getturnVarside(value).turn(value);

proc processA = setf0Callside.outerLoopProcessA.
                print(true).print(true).print(true).
                setturnCallside.resetf0Callside;

proc processB = setf1Callside.outerLoopProcessB.
                print(false).print(false).print(false).
                resetturnCallside.resetf1Callside;

proc outerLoopProcessA = (
                           sum b:Bool.getf1Callside(b).
                           (
                             b -> (sum c:Bool.getturnCallside(c).
                                    (c -> resetf0Callside.
                                          innerLoopProcessA.
                                          setf0Callside <>
                                     tau).
                                    outerLoopProcessA) <>
                                   tau
                           )
                         );

proc outerLoopProcessB = (
                           sum b:Bool.getf0Callside(b).
                           (
                             b -> (sum c:Bool.getturnCallside(c).
                                    (c -> tau <> resetf1Callside.
                                                 innerLoopProcessB.
                                                 setf1Callside).
                                    outerLoopProcessB) <>
                                   tau
                           )
                         );
```

```
proc innerLoopProcessA = (sum b:Bool.getturnCallside(b).
                          (b -> innerLoopProcessA <> tau));

proc innerLoopProcessB = (sum b:Bool.getturnCallside(b).
                          (b -> tau <> innerLoopProcessB));

proc m = processA || processB;

init allow ({print,setf0Operation,resetf0Operation,getf0Operation,
             setf1Operation, resetf1Operation, getf1Operation,
             setturnOperation, resetturnOperation, getturnOperation},
       comm({setf0Varside | setf0Callside -> setf0Operation,
             resetf0Varside | resetf0Callside -> resetf0Operation,
             getf0Varside | getf0Callside -> getf0Operation,
             setf1Varside | setf1Callside -> setf1Operation,
             resetf1Varside | resetf1Callside -> resetf1Operation,
             getf1Varside | getf1Callside -> getf1Operation,
             setturnVarside | setturnCallside -> setturnOperation,
             resetturnVarside | resetturnCallside -> resetturnOperation,
             getturnVarside | getturnCallside -> getturnOperation},
             m || f0(false) || f1(false) || turn(false)));
```

5.4 Model checking the resulting mCRL2 code

The subject of this thesis is a proposed conversion from Java to mCRL2. The resulting mCRL2 code is equivalent to the original Java code. However, very often additional constructs are modelled in the mCRL2 code to help check certain properties. These properties are of course not included in the original Java code (they need to be specified at a meta level) so there is no possibility to convert them automatically. It is also very well possible to take the mCRL2 code resulting from the conversion and to check it for certain properties without modification. This is done here to prove that the critical section in Dekker's algorithm is valid. To achieve this, the following steps can be taken:

1. Convert the Java code into mCRL2 as outlined before. It is assumed here that the the resulting mCRL2 code is stored as `Main.mcrl2`.

2. Linearize the mCRL2 code using the `mcrl22lps` tool from the mCRL2 toolset. This can be done in the following way:

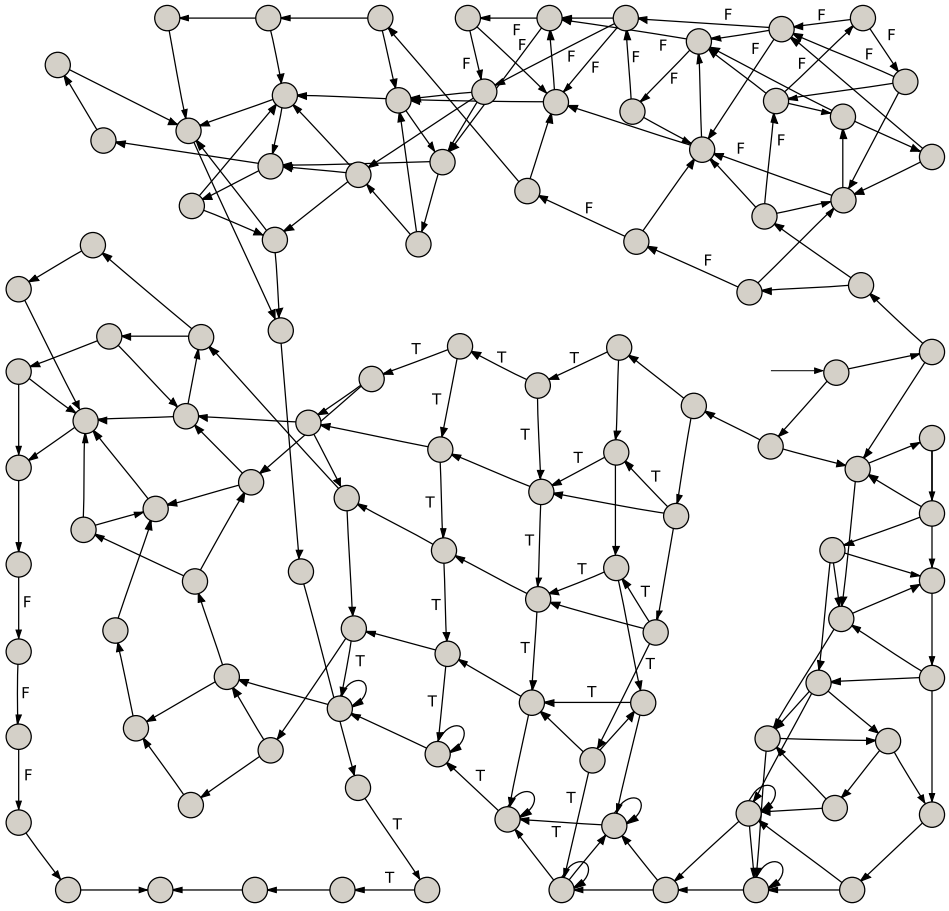
   ```
   %> mcrl22lps Main.mcrl2 > Main.lps
   ```

3. The output of the previous tool can be used to generate a state space using the `lps2lts` tool. This can be done by executing the following command:

   ```
   %> lps2lts Main.lps Main.aut
   ```

   The resulting `.aut` file contains the complete state space of the mCRL2 model.

4. In most cases the state model can be optimized. Optimization is based on trace equivalence, bisimulation equivalences or one of the other equivalence relations defined over process algebra expressions. Because this is not a very complex model, no optimization is required. The resulting state model is shown below (`print(true)` actions are indicated by `T` and `print(false)` actions by `F`, other actions are not shown here):



5. By careful inspection of the state diagram, it is already clear that no interleaving of `print(true)` and `print(false)` can ever occur. However, this is of course not a very useful way to prove such properties as the state diagrams of a realistic model is often several orders of magnitude larger.

6. As a reminder for the original proof of the validity of the critical section: process A executed the action `print(true)` three times while process B exectued the action `print(false)` three times. If there is any interleaving of these print actions (e.g. `print(false)` followed by `print(true)` followed by `print(false)`) this would indicate that the critical section is invalid. This can be checked in the following way. The first step is to generate the traces to each print action:

   ```
   %> lps2lts --trace=1000000 --action=print Main.lps
   ```

   The `--trace` option ensures that all traces are stored (in separate files). Using the `tracepp` the binary trace files can be converted into a more readable format (every action in the trace is printed on a new line). After removal of these newlines, the command line tool `grep` can be used to search for invalid traces:

   ```
   %> grep ".*print(false).*print(true).*print(false).*" output
   ```

   and

   ```
   %> grep ".*print(true).*print(false).*print(true).*" output
   ```

   both of which show no results. This completes the proof of a valid critical section. An algorithm was written in Java, this algorithm was converted into mCRL2 using the conversion function defined in this thesis. The resulting mCRL2 code was linearized and relevant traces were extracted from the model. These traces were checked for invalid sequences of action. No such invalid traces were found, so the model is proved to implement a valid critical section.

5.5 Summary

In this chapter a case study was presented to show how the conversion function can be applied on a real-world example. A certain algorithm was implemented in Java. This algorithm was converted into a subset of Java and it was explained how this can be done and why such a step is neccesary. The conversion was applied on the rewritten Java code to obtain an mCRL2 model. A property was defined for the algorithm and this property was checked against mCRL2 model using tools in the mCRL2 toolset. No modifications to the model had to be made after converting it from Java. The next chapter presents a general

roadmap section for converting Java programs into mCRL2 code.

Chapter 6

Java to mCRL2: A roadmap

This section is part of a bachelor's thesis describing a possible conversion from Java code to the mCRL2 specification language. In this roadmap section, a short summary of steps is given to convert Java code into mCRL2. A detailed description of a (partially) formalized conversion can be found in the thesis itself.

Step 1: Converting Java into a subset of Java

The first step is to rewrite the Java code in such a way that it conforms to the following grammar:

```
<program>          =>  "public" "class" "Main" "{" <declarationlist>
                       "}"
<declarationlist>  =>  <declaration><declarationlist> | <null>
<declaration>      =>  <function> | <shared>
<function>         =>  "public" "static" "boolean" <identifier> "("
                       <parameterlist> ")" "{" <statementlist> "}" |
                       "public" "static" "void" "main" "(" "String"
                       "[" "]" "argv" ")" "{" "m" "(" ")" ";" "}"
<shared>           =>  "public" "static" "boolean" <identifier> ";"
<statementlist>    =>  <statementlist><statement> | <null>
<statement>        =>  <assignment> | <return> | <if> | <print> |
                       <threadstart> | <functioncall>
<assignment>       =>  <identifier> "=" <expression> ";"
<return>           =>  "return" <identifier> ";"
<if>               =>  "if" "(" <identifier> ")" "{" <statementlist>
                       "}" <else>
<else>             =>  "else" "{" <statementlist> "}" | <null>
<print>            =>  "System.out.println" "(" <identifier> ")" ";"
<threadstart>      =>  "(new Thread(){public void run(){Main."
```

```
                        <identifier> "();}}).start ();"
<expression>       =>   <primary> | "!" <expression> |
                        <expression> && <expression> |
                        <expression> "||" <expression>
<primary>          =>   <identifier> | <functioncall> | "true" |
                        "false"
<functioncall>     =>   <identifier> "(" <parameterlist> ")"
<parameterlist>    =>   <parameter><parameterlistend> | <null>
<parameterlistend> =>   "," <parameter><parameterlistend> | <null>
<parameter>        =>   "boolean" <identifier>
<identifer>        =>   CHARACTER-SEQUENCE
<null>             =>   ""
```

This grammar is in fact a very restricted subset of Java, mainly because the only allowed variable type is `boolean`, the code is limited to one class, which should be called `Main` and no looping constructs are allowed. If the Java code can be rewritten in such a way that it conforms to this grammar (preserving semantics) the next step (conversion) can be applied. Programs that cannot be rewritten are beyond the scope of this research.

A few other steps are needed to prepare the code for conversion. The first step is to create a new `public static boolean` function called `m ()` containing the contents of the previous `main` function (where the new `main` function contains only a call to `m ()`). Another important step is to rewrite any thread starting functionality in such a way that it fits into one statement. The `<threadstart>` rule in the grammar shows how to do this (where `<identifier>` is a `public static boolean` function that should be called when the thread starts its execution). Other restrictions imposed by this grammar may seem very stringent but it is definitely not within the scope of this research to define a broader set of supported Java constructs.

Step 2: Converting to mCRL2

The next step is to convert the Java constructs into their mCRL2 equivalents. How this can be done is shown in the following table:

Language construct	Conversion strategy

`public class` *Main* { *Variables Functions* }	Set up three actions used for handling return values:
	
	`act return:Bool;`
	`act obtain:Bool;`
	`act transfer:Bool;`
	
	Set up the basic init statement (the sets *Allow* and *Comm* will very probably be modified later)
	
	`init allow (` { *Allow* }`, comm (` { *Comm* }`, m ))`
	
	And convert *Variables* and *Functions* according to the definitions elsewhere in this table.

`public static boolean` *Variable*`;`	Define nine actions (where *Variable* is replaced by the variable name):
	
	`act set`*Variable*`Callside;`
	`act reset`*Variable*`Callside;`
	`act get`*Variable*`Callside:Bool;`
	
	`act set`*Variable*`Varside;`
	`act reset`*Variable*`Varside;`
	`act get`*Variable*`Varside:Bool;`
	
	`act set`*Variable*`Operation;`
	`act reset`*Variable*`Operation;`
	`act get`*Variable*`Operation:Bool;`
	
	And define a new process with the same name as the *Variable*:
	
	`proc` *Variable*`(value:Bool) =`
	`set`*Variable*`Varside.`
	`Warning(true) +`
	`reset`*Variable*`Varside.`
	`Warning(false) +`
	`get`*Variable*`Varside(value).`
	`Warning(value);`
	
	Again replacing *Variable* by the variable name.

`public static void` *main* `(String [] argv) { m (); }`	No conversion for the main function is neccesary because the `init` construct itself already calls the `m ()` function.
`public static boolean` *Function* `(` *ParameterList* `) {` *Statements*`}`	Convert each function into a separate process. This process has the same name as the corresponding *Function* and the parameters can be defined using the standard boolean data type (sort) in mCRL2. The next step is to convert *Statements* to complete the resulting process equation: `proc` *Function* `(`*Parameter1*`:Bool,` *Parameter2*`:Bool .. ) = .. ;`
Variable `=` *Expression*`;`	`set`*Variable*`Callside(`*Expression*`)`

`return` *Expression*`;`	Direct usage of the `return` action: `return(`*Expression*`)`
`if (`*Expression*`) {` *ThenPart*`} else {` *ElsePart* `}`	*Expression*`->`*ThenPart*`<>`*ElsePart*
`( new Thread (){ public void run (){ Main.`*Function* `(); } } ).start ();`	This statement starts a new thread. If *Process* is the current process (under translation) the resulting process becomes: `((`*Process*`) ||` *Function*`)`
Function `();`	A function call can be translated directly into a call of the corresponding process. The return value has to be obtained: *Function* `|| sum r:Bool.obtain(r)`
`!`*Expression* *Expression* `&&` *Expression* *Expression* `||` *Expression*	Fortunately, the syntax for expression in mCRL2 resembles the Java expression style: `!`*Expression* *Expression* `&&` *Expression* *Expression* `||` *Expression*
Variable (function parameter usage)	*Variable*
Variable (global variable usage)	`sum b:Bool.get`*Variable*`Callside(b)`

Chapter 7

Conclusion

7.1 The next step in this research

The introduction starts by stating a very challenging problem in software development: assessing the quality of software. Altough software correctness is not the only variable in software quality (it is perfectly possible to imagine a completely correct but very unusable program), it is certainly of the greatest importance. Incorrect software will definetely not be of the highest quality.

Correctness can be tested by verifying whether certain properties hold. In the previous chapters a very specific technique was described as a means to achieve this correctness: converting a subset of Java into mCRL2 code and testing it using the mCRL2 toolset. However, the approach of this research is not complete yet. Quite a few steps have to be taken to make this setup usable in testing:

- Implementing the conversion function *conv* as an executable program. The conversion in the case study (see chapter 5) was done by hand. This is not a very realistic situation because any program of reasonable size might be orders of magnitude larger than the example programs used in this research. Furthermore, doing the conversion by hand introduces the possiblity of error. Given enough time, it should not be very complicated to implement the conversion function as a Java to mCRL2 compiler. Such a tool should contain the following components:
  - A Java parser: various Java parsers are available in many programming languages so this part of the tool should not be very hard to realize.
  - A Java simplifier: converting Java to miniJava preserving the original Java semantics. Such a tool can take the parse tree generated in the previous step as input.
  - An implementation of the conversion function, deriving suitable mCRL2 constructs from parts of the parse tree.

  - An mCRL2 writer to format the output in such a way that it is readable by the mCRL2 toolset.

  Of course the second and third component will be the greatest amount of work.

- Fixing several detailed problems of the miniJava to mCRL2 conversion, for instance, there are several minor problems involving braces and the mCRL2 sum operator. For an automated conversion, it is important that these issues are solved. However, this should be rather straightforward.

- Extending the conversion to include mCRL2 data types. In this thesis, the conversion function was only defined for the data type `boolean` but mCRL2 also contains other basic data types as well as user defined data types. One can imagine a conversion from Java to mCRL2 where such data types are preserved (like the current conversion translates boolean variables).

- Applying the conversion on more examples. The conversion as it is defined in this thesis is only applied on relatively small examples. If larger fragments or even complete Java programs are converted into mCRL2 (and tested using the mCRL2 toolset) more results will come up. Based on new results, the conversion function can be modified and re-applied to improve it.

- A possible last step might be to prove the conversion to be correct. Such a prove might be quite a lot of work but is definitely not impossible. An operational semantics of mCRL2 has already been defined [5] and several descriptions of Java at a semantic level are also available (see for example [2]). A proof might be constructed based on equivalence of operational semantics or in a similar way.

7.2 Is this the right way to go?

The approach of converting software code into a formal specification that can be checked is not a new approach. It has been used in the past with varying results, ranging from average to very good. The key element of this research was not to define the formal specification itself but to show how such a conversion might be done (semi-)automatically (in a very strict sense). To verify whether such an approach is useful, the following questions will have to be answered:

1. To what extent is it possible to automatically convert Java code into mCRL2 code?

2. Is the generated mCRL2 model a good representation of the actual software?

3. Will this approach ever be used in practice?

With respect to question 1, quite a lot of research will have to be done to answer this question completely. The two-step model suggested in this thesis (first convert the Java code into miniJava or a similar subset, followed by a conversion into mCRL2 code) will have to be extended. It should be very clear which subset of Java will have to be used and how to convert existing Java code into such a subset of the language. It should also be well-defined how external dependencies have to be handled. For instance, when using libraries written in another programming language. The only way to formulate a complete and convincing answer to question 1 is to define and implement the conversion function as best as possible and to evaluate such an implementation in practice.

There are two rather different answers possible with respect to question 2. The first answer is from a very theoretical point of view: if the Java to mCRL2 conversion function can be proven to be correct, then the generated mCRL2 model can be considered a 'good' representation of the actual software. But from another point of view, a very compact or a very readable model might be considered a 'good' representation. If defects in the mCRL2 model can be mapped back to the original source code, the model can be considered 'good'. This is again something that will have to be tested in practice.

A positive answer to question 3 depends on the outcome of questions 1 and 2. In general: creating a model based on software code and verifying such a model will definitely be used in practice. Java is used more frequently in real-time and embedded applications. Such software is usually more critical and therefore needs to be tested more thoroughly. mCRL2 has proved its usefulness for finding defects in software in the past so it seems worthwhile to attempt to check Java programs using mCRL2. If the conversion function proposed in this research will be completed and implemented it might be a very useful tool in finding defects in Java code. But there is a long way to go, the conversion function is not yet complete and there is no implementation to test with. Nevertheless, this research might be a first step.

Bibliography

[1] J. Bowen. Formal methods in safety-critical standards. In *Proc. Software Engineering Standards Symposium, 1993.*, pages 168–177. IEEE, 1993.

[2] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. *Formal Syntax and Semantics of Java*, pages 157–200, 1999.

[3] W. Fokkink. *Introduction to Process Algebra*. Springer, 1999.

[4] J.F. Groote, A. Mathijssen, B. Ploeger, M. Reniers, M. van Weerdenburg, and J. van der Wulp. Process algebra and mcrl2. In *IPA Basic Course on Formal Methods 2006*, pages 1–26, 1997.

[5] J.F. Groote, A. Mathijssen, and M. Reniers. The formal specification language mcrl2. In *Methods for Modelling Software Systems, Dagstuhl Seminar Proceedings 06351*, 2007.

[6] A. Mathijssen. Mcrl2 primer. Technical report, Technische Universiteit Eindhoven, 2007.

[7] A. Mathijssen and A.J. Pretorius. Specification, Analysis and Verification of an Automated Parking Garage. Technical report, Technical Report 05/25, Eindhoven University of Technology, ISSN 0926-4515, 2005.

[8] M. Schenke. Development of correct real-time systems by refinement. Technical report, Universitaet Oldenburg, 1997.

[9] Marko van Eekelen, Stefan ten Hoedt, René Schreurs, and Yaroslav S. Usenko. Analysis of a session-layer protocol in mcrl2. verification of a real-life industrial implementation. In *Proc. 12th Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007)*, Lecture Notes Computer Science. Springer, 2007. To appear.