Alvis Approach to Hexor Robot Controller Development

Marcin Szpyrka
Department of Automatics
AGH University of Sci. and Techn.
Kraków, Poland
Institute of Physics
Jan Kochanowski University
Kielce, Poland
Email: mszpyrka@agh.edu.pl

Piotr Matyasik
Department of Automatics
AGH University of Sci. and Techn.
Kraków, Poland
Email: ptm@agh.edu.pl

Rafał Mrówka
Department of Automatics
AGH University of Sci. and Techn.
Kraków, Poland
Email: Rafal.Mrowka@agh.edu.pl

***Abstract*—Alvis is a novel modelling language defined especially for the embedded systems design and verification. The language has its origin in CCS and XCCS process algebras, but algebraic equations have been replaced with a Haskell based high level programming language. Moreover, Alvis provides communication diagrams for the visual modelling of an embedded system structure, especially from the control and data-flow point of view. This paper presents an introduction to Alvis based on a model of a controller for the Hexor II mobile robot.**

***Index Terms*—Alvis modelling language, embedded systems, Hexor II Robot, formal modelling**

I. Introduction

The aim of the paper is to present an introduction to *Alvis modelling language* based on a real example. For this purpose, we have chosen a model of a controller for the Hexor II mobile robot that has been already presented in [1], but this time a different modelling language has been used and the model is very close the real implementation of the controller.

The direct ancestors of Alvis are the XCCS [2], [3], [4] and CCS process algebras [5], [6], [7]. CCS (Calculus of Communicating Systems) is one of the most famous process calculi. It provides a tool for the high-level description of interactions, communications, and synchronizations among agents and also algebraic laws to analyse agents properties. One of the main disadvantages of CCS is the lack of a suitable graphical language for modelling concurrent systems. A designer is forced to use the textual (algebraic equations) form of a system description.

XCCS (eXtended CCS) is a graphical extension of the CCS process algebra. XCCS provides a graphical modelling language for the description of interactions among agents, but still preserves algebraic equations to describe the behaviour of individual agents. XCCS is compatible with CCS calculus. It is possible to transform an XCCS model to an equivalent CCS script automatically. Generated CCS scripts are compatible with Edinburgh Concurrency Workbench [8], so the CWB tool can be used for the formal verification of XCCS models.

The paper is supported by the Alvis Project funded from 2009-2010 resources for science as a research project.

The CWB tool is also one of the main disadvantages of XCCS. An application of XCCS and finally CWB for the modelling and verification of a controller for the Hexor II mobile robot [4], [1] exposed essential limitations of the verification tool. CWB is not able to cope with a state space with more than 300 thousands states.

Alvis, as well as XCCS, uses two layers, a graphical and a textual one. The Alvis graphical layer, called *communication diagrams*, is a significant enhancement of XCCS diagrams. Communication diagrams are hierarchical constructions and reflect the structure of the system under consideration more precisely (active and passive agents, two-way ports, etc.) than XCCS diagrams. The XCCS algebraic layer has been replaced with a completely new Alvis code layer. Instead of algebraic equations, Alvis uses a high level programing language based on the Haskell syntax. Moreover, Haskell is used to define data types for parameters and to define functions for data manipulation.

Alvis provides a possibility of the formal verification of models. An Alvis model is transformed into a *labelled transition system* (LTS) encoded using the *Binary Coded Graphs* (BCG) format. Then, the CADP toolbox [9] is used to verify its properties.

The paper is organised as follows. Section II presents some basic information about Hexor mobile robot and the new controller. Section III deals with communication diagrams – the graphical layer of Alvis. The second (code) layer is described in Section IV. A short summary is given in the final section.

II. Hexor microcontroller model

Hexor II (see Fig. 1) is an autonomous 6-legged mobile robot developed by the *Stenzel* company for educational purposes [10]. An Alvis model of the ATmega128 microcontroller [11] is considered in the paper. The tasks of that chip are as follows:

- scanning sensors (sonar, tentacles, infrared),
- generating signals for servos (PWM),
- executing the movement algorithm,

Copyright © 2010 by Department of Microelectronics & Computer Science, Technical University of Lodz

- communicating with the host computer,
- executing higher level algorithm (obstacle avoidance, obstacle search, etc.).

Figure 1. Hexor mobile robot

Because of some software platform limitations [12], [13] of the original design, a new Hexor's internal controlling software architecture has been developed. The basic program with one control loop and interrupt routines has been replaced by a real-time embedded operating system. Each subsystem of Hexor robot is managed by its own task (see Fig. 2). One of the main advantages of these modifications is a more clear and easy to understand source code. This simplifies the parent control layer and enhances response time of a robot.

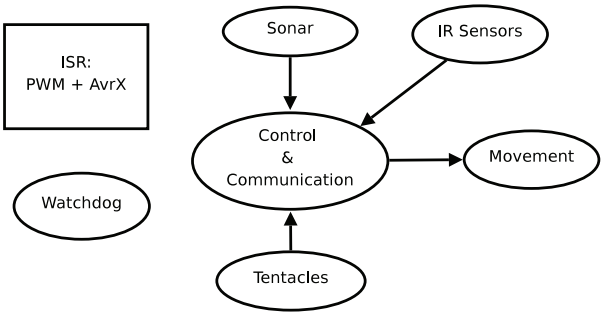

Figure 2. Hexor task model

The complete Hexor system architecture is presented in Fig. 3. It consists of three elements:

- AvrX micro-kernel,
- HexorNG software,
- High level intelligence.

Figure 2 presents a model of tasks and communication in the proposed new system. Ovals represent tasks, arrows represent FIFO's and data flow direction, and the square represents ISR[1]. The tasks are used for the following purposes:

- *Control & Communication* task is the main system task responsible for setting up hardware, interchanging data between other tasks, communication services and basic intelligence. The last feature means simply: *do not run into obstacles*.
- *Sonar* task provides the most recent sonar sensor readings.
- *IR* and *Tentacles* tasks report any changes in sensors state.
- *Movement* task controls servo positions and executes the movement algorithm on demand.
- *Watchdog* task simply resets the hardware watchdog chip every second.
- *ISR* is responsible for generating proper PWM[2] signals for servos. It is also used for context switching and timer execution for AvrX micro kernel.

[1]Interrupt Service Routine

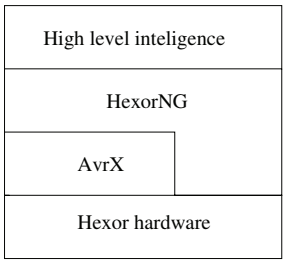

Figure 3. Hexor system architecture

III. Communication Diagrams

The Alvis *graphical layer* is used to define interconnections among agents and takes the form of a *hierarchical communication diagram*. The term *agent* stand for any distinguished part of the model under consideration with its own identity. The graphical layer shows all communication channels among agents. It is composed of a set of pages, i.e. non-hierarchical parts of the diagram. A page may contain active agents, passive agents and connections among them.

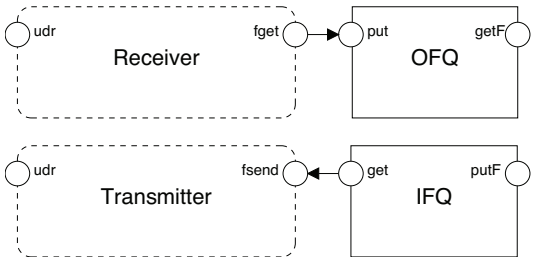

Figure 4. Communication page

There are two kinds of agents in Alvis. *Active agents* (drawn as ovals) perform some activities and are similar to tasks in Ada programing language [14], [15], [16]. Each of them can be treated as a thread of control in a concurrent system. On the other hand, *passive agents* (drawn as rectangles) do not perform any individual activity, and are similar to protected objects (shared variables). Passive agents provide mechanism

[2]Pulse Width Modulation

for the mutual exclusion and data synchronisation. Moreover, Alvis provides mechanisms for the description of interrupt handling routines in the form of agents. An interrupt service is represented by a single active agent called *interrupt agent*. Such an agent is drawn using dashed lines.

The Hexor controller communication diagram contains four pages. *Communication* page is shown in Fig. 4. The page contains four agents. Two of them are simple FIFO queues for storing frames to be send or to be received. Another two agents are interrupt service routines. They provide interrupt driven UART communication. The *Receiver* task is executed every time a byte is ready to be taken from shift register. When it receives a complete frame it is stored in the queue. The *Transmitter* task is executed every time a byte was shifted from UART register. It takes frames from the output queue and sends them byte by byte.

An agent can communicate with other agents through *ports*. Ports are drawn as circles placed at the edges of the corresponding oval or rectangle. There is no distinction between input and output ports on communication diagrams. Any port can be used as an input or output one. The role a port plays is defined by its connections and the implementation of the corresponding agent.

It should be underlined that Alvis is a case sensitive language. Identifiers for agents must start with an uppercase letter, while identifiers for ports must start with a lowercase letter. Other characters (if any) must be alphabetic character, either uppercase or lowercase, digits, or an underscore.

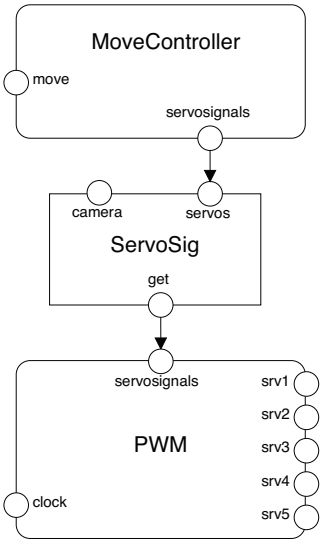

Figure 5. ServoController page

Alvis agents can communicate with each other directly using the *connection mechanism* (communication channels). A connection between two active agents creates a synchronisation point between them. On the other hand, a connection between an active and an passive agents is similar to an asynchronous procedure call. Communication diagrams provide two kinds of connections: one-way (see Fig. 5) and two-way (see Fig. 4) ones. A one-way connection contains an arrowhead that points out the input port, but such a port is treated as an input port only for this particular connection. It can play another role for its other connections.

Figure 5 presents the *ServoController* page. It consists of two major elements. The first is a five channel software PWM implementation that generates signals for servos. The second element is the *MoveController* agent that executes the movement algorithm. It sets desired servo positions according to the selected movement mode (stop, forward, backward, left or right). Both agents are exchanging information via the *ServoSig* passive agent, which has also an input port for setting the camera position.

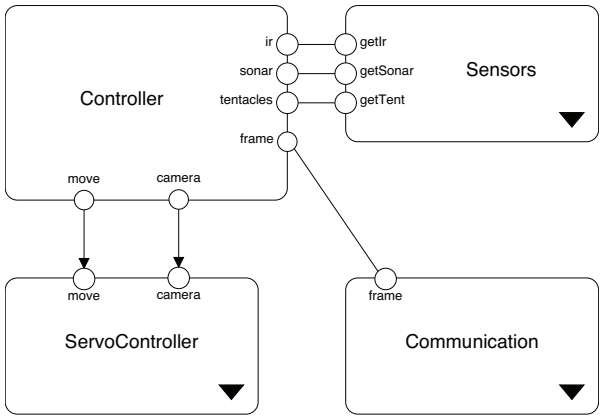

Figure 6. Hexor page

Alvis communication diagrams are hierarchical graphs. Pages are combined using the so-called *substitution mechanism*. An active agent at one level can be replaced by a page on the lower level, which usually gives a more precise and detailed description of the activity represented by the agent. Such an substituted agent is called *hierarchical* one. Hierarchical agents are indicated by black triangles. It should be underlined that all ports of an hierarchical agent must appear on the corresponding subpage as external (unconnected) ones.

The communication diagram for the controller contains two levels. The first (higher) level is shown in Fig. 6. Three agents with black triangles icons stand for three different modules of the system under consideration. Agent *Controller* represents the main control process of the system. The last page of the communication diagram is shown in Fig. 7. The agents represent the robot sensors.

IV. Code layer

To describe the behaviour of individual agents, Alvis uses a high level programing language based on the Haskell [17] syntax called *Alvis Behaviour Description Language* or shortly ABD language [18]. The ABD language has its origin in CCS and XCCS process algebras. However, to make the language

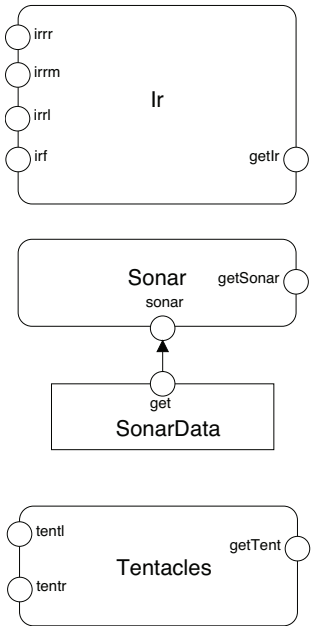

Figure 7. Sensors page

more convenient from the practical (engineering) point of view, algebraic equations and operators have been replaced with statements typical for high level programming languages.

ABD language is used to define:

- data types used in the model under consideration,
- functions for data manipulation
- behaviour of individual agents.

This section presents a few pieces of Alvis code that demonstrate the most important parts of the language.

The code layer of an Alvis model is stored in a textual source file. The file contains two parts – the preamble and implementation. Encoded in pure Haskell, the *preamble* contains definitions of types, constants and functions used to manipulate data in a model. The *implementation* contains definitions of the agents' behaviour and is encoded using native ABD language statements.

Table I
SELECTED BASIC HASKELL TYPES

Type name	Description
Char	Unicode characters
Bool	Values in Boolean logic – *True* and *False*
Int	Fixed-width integer values – The exact range of values represented as *Int* depends on the system's longest *native* integer.
Double	Float-point numbers typically 64 bits wide and uses the system's native floating-point representation.

The ABD language uses the Haskell type system. Selected basic Haskell's types recommended to be used in ABD language are presented in Table I. The most common composite data types in Haskell (and ABD) are *lists* and *tuples*. A *list* is a sequence of elements of the same type, with the elements being enclosed in square brackets and separated by commas, while a *tuple* is a sequence of elements of possibly different types, with the elements being enclosed in parentheses and separated by commas. Moreover, to make the source code more readable, one can introduce a synonym for an existing type (see type *Frame* below) and define the so-called algebraic data types, e.g. an enumeration type (see type *FrameType* Listing 1). For more details see for example [17].

```
data FrameType = IR | TENT | MOVE | CAM | SONAR
type Frame = (FrameType, Int)

char2ftype :: Char -> FrameType
char2ftype '1' = IR
char2ftype '2' = TENT
char2ftype '3' = MOVE
char2ftype '4' = CAM
char2ftype '5' = SONAR
```

Listing 1. Part of the model preamble

Listing 1 presents a part of the model preamble.

The implementation part contains definitions of the agents' behaviour. It contains at least one *agent block* of the following form:

```
agent AgentName
-- declaration of parameters
-- agent body


agent Sonar
-- parameters
  value :: Int
  oldVal :: Int
-- body
  oldVal = 0
  startPoint:
    in sonar value
    jump (value == oldVal) sNoChange
    out getSonar value
    oldVal = value
  sNoChange:
    delay 100
    jump startPoint
```

Listing 2. Agent Sonar implementation

Let us consider the definition of the agent *Sonar* presented in Listing 2. The agent *Sonar* uses two integer parameters *value* and *oldVal* (with the initial value equal to 0). ABD language uses the *recursion* mechanism for looping. Two language concepts are used for this purpose *labels* and the *jump statement*. Labels are identifiers followed by a colon (e.g. `startPoint:` in Listing 2). The jump statement is composed of the *jump* key word and a label name (without a colon). Most Alvis statements may be followed by a *guard*. A *guard* is an additional constraint, which must be fulfilled before the corresponding statement is executed. Guards are logical expressions, written in Haskell, placed inside round brackets after the statement name (see the first jump statements in Listing 2).

The ABD language uses two statements for the communication. The *in* instruction for collecting data and *out* for sending. Each of them takes a port name as its first argument and optionally a parameter name as the second. Parameters are not used for pure synchronisations. The *in* statement assigns the collected value to its parameter, while the *out* statement sends the value of its parameter (or constant). The *Sonar* agent collects data through the port *sonar* and sends data through the port *getSonar*. To postpone the agent for 100 milliseconds the *delay* statement is used.

In the implementation part, the = symbol stands for the assignment operator and is a part of the *exec* statement. The *exec* statement is the default one in the ABD language and therefore, it can be omitted if no guard is used. For example, the *Sonar* agent uses the *exec* statement to assign a new value to the *oldVal* parameter.

The *Ir* and *Tentacles* agents are implemented in a similar way.

```
agent PWM
-- parameters
  servos :: (Int,Int,Int,Int,Int)
  -- left leg, tilt, rigth leg,
  -- camera horizontal, camera vertical
  counter :: Int
-- body
  counter = 0
startPoint:
  jump (counter < 60 ) go
    counter = 0
  go:
  in clock
  in servosignals servos
  alt (counter == 0)
    out srv1 1
    out srv2 1
    out srv3 1
    out srv4 1
    out srv5 1
  alt (counter > 0)
    out (counter == get1st servos) srv1 0
    out (counter == get2nd servos) srv2 0
    out (counter == get3rd servos) srv3 0
    out (counter == get4th servos) srv4 0
    out (counter == get5th servos) srv5 0
  counter = counter + 1
  jump startPoint
```

Listing 3. Agent PWM implementation

A communication through a port can be a pure synchronisation i.e. a communication without sending values of parameters. Such a communication only synchronises two agents. For example, the *PWM* agent (see Listing 3) synchronises with the clock.

In order to allow for the description of agents whose behaviour may follow different alternative paths, the ABD language offers the *alt* statement. The statement is similar to the basic *select* statement from Ada programming language. Let us consider the piece of code from Listing 4. The behaviour of the agent *Controller* is described by a single loop with

```
agent Controller
   f :: Frame
   ft :: FrameType
   d :: Int
-- body
startPoint:
  alt -- frame received from serial link
    in frame f
    ft = fst f
    alt (ft = IR)
      out ir -128
    alt (ft = TENT)
      out tentacles -128
    alt (ft = SONAR)
      out sonar -128
    alt (ft = MOVE)
      d = snd f
      out move d
    alt (ft = CAM)
      d = snd f
      out camera d
  alt -- new sensor values to send
    in ir d
    f = (IR, d)
    out frame f
  alt
    in sonar d
    f = (sonar, d)
    out frame f
  alt
    in tentacles d
    f = (TENT, d)
    out frame f
  jump startPoint
```

Listing 4. Agent Controller implementation

four alternatives (branches). Each branch may have attached a guard. A branch is called *open*, if it does not have a guard attached or its guard evaluates to *true*. Otherwise, a branch is called *closed*. When an *alt* statement is to be executed, all guards are evaluated to determine, which branches are *open*. If more than one branch is open, the choice between them is indeterministic.

```
agent FQ
  fqueue :: [Frame]
  count :: Int
  f :: Frame
-- body
  count = 0
  alt (count > 0)
    f = head fqueue
    fqueue = tail fqueue
    count = count -1
    out get f
  alt
    in put f
    fqueue = fqueue ++ [f]
    count = count + 1
```

Listing 5. Agent FQ implementation

The *alt* statement is also used for description services provided by passive agents. Let us consider the implementation

of the *FQ* agent presented in Listing 5. The agent represents a FIFO queue. The two *alt branches* provide the queue interface. The first one describes collecting the queue head, while the second describes including a new element to the queue.

```
agent Receiver
  ftype :: FrameType
  c1 :: Char
  c2 :: Char
  f :: Frame
  count :: Int
-- body
  count = 1
  in udr c1
  critical
    alt (count == 1)
      ftype = char2ftype c1
      count = count + 1
    alt (count == 2)
      c2 = c1
      count = count + 1
    alt (count == 3) -- complete frame received
      f = (ftype, (char2int c2 c1))
      out fget f
      count = 1
```

Listing 6. Agent Receiver implementation

As it was already said, an interrupt service is represented by an interrupt agent. Example of an implemention such an agent is shown in Listing 6. The *Receiver* agent is triggered every time a byte is ready to be taken from the shift register. When it receives a complete frame it is stored in the queue. The *char2int* function is used to convert two digits into an integer. The *Transmitter* agent works in a similar way. It is triggered every time a byte was shifted from UART register. It takes frames from the output queue and sends them byte by byte.

V. Summary

Selected parts of an Alvis model of a controller for the Hexor II mobile robot have been presented in the paper. The model has been used to present a survey of main features of Alvis modelling language. Alvis has its origin in the CCS and XCCS process algebras, but to make it more convenient from the practical point of view, algebraic equations and operators have been replaced with statements typical for high level programming languages.

Defined for the embedded systems design, Alvis seems to be more accessible for engineers than classical formal methods, but still preserves a possibility of formal verification with CADP (*Construction and Analysis of Distributed Processes*) toolbox [19], [9].

To verify or simulate an Alvis model a third layer is introduced. It gathers information about all agents in the model and their states. The *meta-data layer* is generated automatically. Each agent is described using the so-called *agent description block* (ADB) and a list of its parameters values. The meta-data layer keeps the current state of a model and is used for the LTS generation. To verify a model, the generated LTS is encoded in the BCG (*Binary Coded Graphs*) format. Then, the CADP Evaluator is used to check whether the generated LTS satisfies its requirements defined using μ-calculus [20] formulas. The CADP toolbox and BCG format allow us to analyse models up to 30 milion states.

References

[1] M. Szpyrka and P. Matyasik, "Modelling Hexor robot behaviour with value passing XCCS," in *Software Engineering Techniques in Progress (the 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009)*, Z. Huzar, J. Nawrocki, and M. Szpyrka, Eds. Krakow, Poland: AGH University of Science and Technology Press, 2009, pp. 67–78.

[2] ——, "Formal modelling and verification of concurrent systems with XCCS," in *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC 2008)*, Krakow, Poland, July 1-5 2008, pp. 454–458.

[3] K. Balicki and M. Szpyrka, "Formal definition of XCCS modelling language," *Fundamenta Informaticae*, vol. 93, no. 1-3, pp. 1–15, 2009.

[4] P. Matyasik, "Design and analysis of embedded systems with XCCS process algebra," Ph.D. dissertation, AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Kraków, Poland, 2009.

[5] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1989.

[6] C. Fencott, *Formal Methods for Concurrency*. Boston, MA, USA: International Thomson Computer Press, 1995.

[7] L. Aceto, A. Ingófsdóttir, K. Larsen, and J. Srba, *Reactive Systems: Modelling, Specification and Verification*. Cambridge, UK: Cambridge University Press, 2007.

[8] F. Moller and P. Stevens, *Edinburgh Concurrency Workbench user manual*, 1999. [Online]. Available: http://www.dcs.ed.ac.uk/home/cwb

[9] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2006: A toolbox for the construction and analysis of distributed processes," in *Computer Aided Verification (CAV'2007)*, ser. LNCS, vol. 4590. Berlin, Germany: Springer, 2007, pp. 158–163.

[10] *HexorII Robot Manual*, Stenzel, 2006.

[11] *8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash ATmega128 ATmega128L*, Rev. 2467g-avr-09/02 ed., Atmel, 2002.

[12] P. Matyasik and G. J. Nalepa, "Knowledge-based control of reactive systems with multi-layer architecture," in *Proc. of Mixdes 2007, the 14th International Conference Mixed Design of Integrated Circuits and Systems*, Ciechocinek, Poland, June 21-23 2007, pp. 667–672.

[13] P. Matyasik, G. J. Nalepa, and P. Zięcik, "Prolog-based real-time intelligent control of the hexor mobile robot," in *Advances in Artificial Intelligence: Proceedings of the 30th Annual German Conference on AI, KI 2007*, ser. Lecture Notes in Artificial Intelligence, vol. 4667, Osnabruck, Germany, 10–13 September 2007, pp. 485–488.

[14] Ada Europe, *Ada Reference Manual ISO/IEC 8652:2007(E) Ed. 3*, 2007.

[15] J. Barnes, *Programming in Ada 2005*. Addison Wesley, 2006.

[16] A. Burns and A. Wellings, *Concurrent and real-time programming in Ada 2005*. Cambridge University Press, 2007.

[17] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*. Sebastopol, CA, USA: O'Reilly Media, 2008.

[18] M. Szpyrka, P. Matyasik, and R. Mrówka, "Introduction to Alvis internal language syntax," AGH University of Science and Technology, Kraków, Poland, CSL Technical Report 1, 2010, http://cslab.ia.agh.edu.pl/en:csltr.

[19] H. Garavel, F. Lang, and R. Mateescu, "An overview of CADP 2001," *EASST Newsletter*, vol. 4, pp. 13–24, 2002, also available as INRIA Technical Report RT-0254.

[20] E. A. Emerson, "Model checking and the Mu-calculus," in *Descriptive Complexity and Finite Models*, ser. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, N. Immerman and P. G. Kolaitis, Eds. American Mathematical Society, 1997, vol. 31, pp. 185–214.