Journal of Universal Computer Science, vol. 9, no. 11 (2003), 1277-1295
submitted: 2/6/03, accepted: 5/9/03, appeared: 28/11/03

Automatically Generated CSP Specifications

Frantisek Scuglik
Brno University of Technology, Czech Republic
scuglik@fit.vutbr.cz

Miroslav Sveda
Brno University of Technology, Czech Republic
sveda@fit.vutbr.cz

Abstract: Two possibilities of automated CSP (Communicating Sequential Processes) support are introduced in [11] and [10] using either behavioral diagrams or application source code. While in the first approach a tool generates CSP specification from behavioral diagrams, based on UML Composite States diagram, in the second approach an application source code is translated directly into CSP specification using a compiler. This paper reviews tools related to both techniques.
Key Words: CSP, Model, Formal Specification, UML, Translator, Grammar
Category: I.6.4 Model Validation and Analysis

1 Introduction

At present day, widely used hardware and software systems occur in applications where failure is unacceptable. These systems interact with us every day and we often do not realize how much can a failure impact our lives. Airport control systems, bank systems, medical instruments, and other examples provide a small list of most critical applications where an error can cause human deaths. Clearly, the need for reliable systems is critical. As the usage of such systems grows, the need for their correctness is increasing. Moreover, with successful expansion of Internet and embedded systems into cars, airplanes etc., these systems should be even more reliable. Therefore, development of system verification tools represents one of major topics in computer science and engineering, [3].

The first step in system verification consists in specifying which properties should the system fit. For example concurrent system should fit the property that it never reaches deadlock. After defining the system requirements, the second step represents construction of the system's formal model. To be able to verify the model, the specification should include those properties which must the system conform to be reliable. On the other hand, the model should abstract those properties which don't influence the correctness of verified system and unnecessarily complicate the verification process. Although the formal model construction and verification represents time-consuming process, these steps are not exceptional in current practice due to high cost of system failures.

This contribution describes development of a special tool providing a user interface for system developers. The user can utilize the developed tool to specify system's behavior and generate system's formal model denoted in formal language called Communicating Sequential Processes (CSP). The algebra of CSP provides system's formal description in an understandable and useful way. Two ways how to generate CSP specifications are introduced using either behavioral diagrams or application source code describing systems behavior. The behavioral diagrams stem from UML Composite States diagrams, where each diagram graphically describes a subsystem behavior. An n-ary tree is build for each diagram and, by climbing down the tree and recording the visited nodes, a tool generates the related CSP representation. The other way applies a special compiler on the application source code. The compiler's grammar describes acceptable program structures such as condition, loop, function call. Whenever the compiler reaches a terminal symbol in the grammar, the CSP specification of this terminal is generated.

First part of this contribution focuses on the relevant subset of CSP algebra used in the developed tool, second part introduces behavioral diagram notations and the source code representation in CSP. Third part describes the automated translation from behavioral diagrams and application source code to CSP specifications. Last part of this contribution includes a cash dispenser specification case study utilizing the behavioral diagrams.

2 Representation of diagrams / source code

When generating the CSP specification, the developed technique utilizes only small part of the CSP algebra. This section describes not only the utilized CSP syntax, but also behavioral diagrams and basic program structures, and their CSP representation.

2.1 Relevant subset of CSP syntax

Understanding, designing and building concurrent systems represent a major challenge for computer science. The involved complications vary from the sequential programming problems, therefore the concurrent systems requires systematic approaches.

Concurrent systems are all around us. They consist of independent, but communicating components. The familiar examples include:

- the network of bank cash machines
- the Internet
- the telephone system

– the components of a PC

The algebra of CSP provides a possibility for concurrent systems to be modeled in more elementary and abstract way. It is supported by particular software tools which offers system analysis and verification.

CSP describes processes - objects which exist independent on each other, but may communicate. During their lifetime, processes can perform various actions or events. These events represents the visible part of modeled processes. For example, when describing a simple vending machine, two events may be interesting:

1. *coin* - represents insertion of a coin

2. *choc* - represents appearance of a chocolate

The set of events used by the process to represent its behavior is called alphabet or interface. During the process activity the events in the interface may occur once, many times, or not at all. Which events should be included in the interface depends on aspects of process behavior which are interesting. For example, when specifying a lecture and interesting just for the beginning and the end of the lecture then the interface of the process consists of two events - begin and end.

The simplest possible process behavior stands for do nothing written as $STOP$. Whenever the behavior of a system reaches this process then deadlock occurred. Non trivial processes are written by means of prefixing operator which allows events to occur in sequence. So, when P is a process and a an event then $a \rightarrow P$ represents a process which performs the event a and then behaves like process P. Expressions of the type $P \rightarrow Q$ or $a \rightarrow b$ are not allowed. The prefix operator defines only the relation between events and processes.

Except $STOP$ another predefined process exists in CSP - $SKIP$. Like $STOP$, it does nothing but ends correctly. Therefore, the $SKIP$ process indicates the correct termination of a process.

Utilizing predefined processes and the prefix operator only finite processes can be created. But often have to be specified processes that run forever. To achieve this goal recursion is included. For example, specification of a clock using an event $tick$ describes the following process:

$$CLOCK = tick \rightarrow CLOCK$$

The process $CLOCK$ performs the event $tick$ repeatedly.

Specified processes often don't just perform single sequence of events but may have alternative behavior caused by their environment, for example. So, if P and Q are processes and x and y are distinct events, then the process

$$x \rightarrow P \mid y \rightarrow Q$$

performs either the event x and then behaves like process P or performs the event y and then behaves like process Q.

Modeled processes usually don't appear isolated but interfere with other process, for example with the process's environment. Mutual interaction between two or more processes means that these processes performs common events simultaneously. On which events the processes should synchronize specifies the alphabet of events. For example, when describing the vending machine again, the new process representing the customer interacts with the machine. Example 1 describes these interacting processes.

Example 1:

$$\begin{aligned}
MACHINE &= coin \rightarrow (choc \rightarrow MACHINE \mid coffee \rightarrow MACHINE) \\
CUSTOMER &= coin \rightarrow choc \rightarrow SKIP \\
MACHINE\ {}_A\|_A\ & CUSTOMER \\
A &= \{coin, choc, coffee\}
\end{aligned}$$

So far the utilized events were considered regardless of whether their represents inputs or outputs. However, separated notation for input and output may be useful for some cases. For this purpose a special event in the form $c.v$ is defined, where c stands for the communication channel name and v stands for the message value send through the channel. Each channel has a type which simply represents the set of events which can be transmitted among the channel. To support sending and receiving of messages two operators are defined: process $c!v \rightarrow P$ sends a message v among the channel c and then behaves like P, process $c?x : T \rightarrow P(x)$ receives the message x of the type T and then behaves like $P(x)$. Until a message of the specified type appears on the input the receiving process waits.

The complete algebra of CSP provides much more notations but for this contribution purposes the presented subset fit the requirements. [1] describes precisely the complete algebra of Communicating Sequential Processes, [2] presents more simplified version.

2.2 Failures Divergence Refinement - FDR

FDR facilitates verification of many finite system properties and analysis of systems which fail the test. It stems from the Communicating Sequential Processes theory and utilizes refinement theory which provides huge range of correctness requirements including the absence of deadlock and livelock. FDR includes also requirements for general safety and liveliness properties.

FDR provides understandable and usable capabilities and extensive debugging facilities to support system development. Therefore, FDR is suitable for verification of systems with complex behavior. When an error occurs, FDR describes the state that lead to the failure as well as the sequence of events that engaged in this state. At present day FDR can analyze extremely large state-space (for example $7^{2^{1024}}$) within few minutes on common desktop PCs.

FDR was specifically developed for analysis and verification of industrial applications. It was successfully applied on VLSI circuits, embedded systems, etc. Another major group of applications involve using FDR to check communications and communication protocols, specifically to detect security holes by authentication key exchange.

2.3 Behavioral diagrams

Behavioral diagrams stems from UML Composite States diagrams. In the UML specification, each diagram may have initial pseudostates and final states. A transition to the enclosing state represents a transition to the initial pseudostate. A transition to a final state represents the completion of activity in the enclosing region. For this contribution, the initial state represents start of the process, and the final state represents end of the process. The initial state is depicted as a small solid filled circle, the final state is shown as a circle surrounding a small solid filled circle. The object describing an event in the process behavior is represented as a rounded rectangle with event description inside. The transitions representing mutual relations between diagram elements are shown as an arrow optionally with event description. Synchronization point and communication channel are depicted as a rectangle and a named rectangle where the list of events represents synchronization alphabet or the channel's type. Figure 1 depicts all introduced diagram elements.

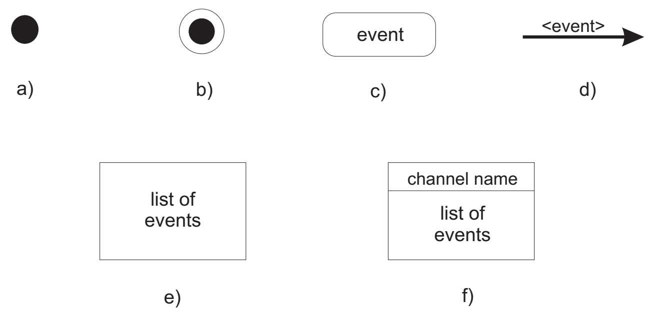

Figure 1: Diagram elements: a) process start, b) process end, c) event, d) mutual relation, e) synchronization, f) communication channel

Process start - this symbol denotes the start of the process behavior. Each specified process behavior begins with this symbol. A name assigned to this symbol specifies name of the process.

Process end - this symbol denotes the end of the current process behavior and specifies which process follows. In case that the current process terminated his behavior successfully and no other processes follow then the name of this symbol will be $SKIP$.

Event - this symbol denotes the event performed by the process.

Mutual relation - this symbol denotes relation between particular events and processes. In other words this symbol represents the prefixing operator. The event in the specification, which's presence is optional, serves to describe alternative process behavior. The alternative process behavior can be specified also using common events.

Synchronization - this object serves to describe interaction between processes. Processes join this symbol direct from their start symbols and the list of events inside the object specifies on which events have the joined processes to synchronize. The synchronization symbol may be joined also to another synchronization points. This joining denotes interaction between particular groups of already synchronized processes. Synchronization of process groups may be performed on other set of events than the events in particular groups.

Communication channel - similarly to the synchronization point, the communication channel denotes relation between processes. The list of events described inside the symbol represents the type of the channel, i.e. the set of events which can be sent among the channel. To decide whether the process sends the message or receives the message the mutual relation symbol is used. This notation requires presence of the event in the relation and defines which event will be sent among the channel or in which event will the message be stored.

Now, when intuitively composing the presented symbols together into a diagram the specification of particular processes is constructed. Utilizing these processes and putting them in interaction the specification of the complete system arise. Figure 2 shows the vending machine diagram, figure 3 depicts copying of a bit via a communication channel with related CSP specification.

2.4 Basic program statements

The source code describes behavior of the system. The complete system behavior description can be decomposed into basic program statements. Each statement

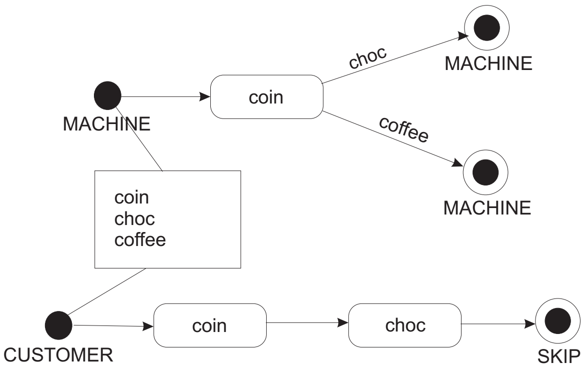

Figure 2: Vending machine diagram

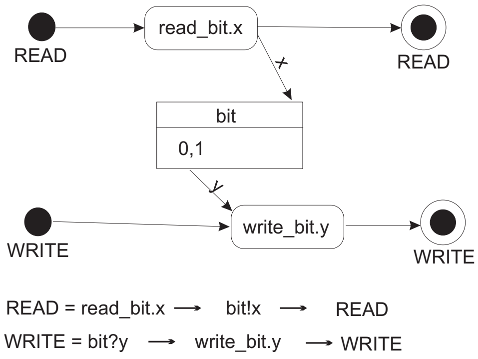

Figure 3: Copy bit specification

is represented in the CSP specification by a rewriting rule described bellow. Recomposition of the CSP specifications of basic program statements represents the resulting system behavior.

Basic program statements can be specified using CSP as follows.

Assignment :

$x = y$;

$ASSIGNMENT = assignvalue \rightarrow SKIP$

- The event *assignvalue* represents assignment of a value to a variable and then the process ends if no next statement follows.

Function call :

$my\_func(params)$;

$START = process\_params \rightarrow MY\_FUNC$

- The event *process_params* represents the processing of the function input parameters and then the function is called. If no more detailed description of the function is required, then $MY\_FUNC = STOP$.

Sequential composition :

$COMPOSITION = endproc1 \rightarrow PROC2$

- The event $endproc1$ indicates the end of process $PROC1$ and then the process $PROC2$ is initiated.

Conditional :

If (cond) proc1(); else proc2();

$CONDITIONAL = cond\_hold \rightarrow PROC1 \mid else \rightarrow PROC2$

- The event *cond_hold* means that the condition is satisfied and the event else means the opposite.

While :

While (cond) proc1();

$WHILE = cond\_hold \rightarrow DO\_BODY \mid else \rightarrow STOP$

$DO\_BODY = body\_done \rightarrow WHILE$

- The event *cond_hold* means that the condition in the while statement holds and the event else means that the condition does not hold. The event *body_done* indicates that the body of the while loop finished and the while condition can be evaluated again.

3 Automated CSP specification

When using the behavioral diagrams or source code, CSP specifications can be generated automatically. In the first case, utilizing behavioral diagrams and their representation in CSP, an n-ary tree represents the diagram. In the second case, the CSP specification is derived from an application source code directly. A compiler uses a grammar, which describes exactly the acceptable program structures, for generating the CSP specification of an application source code. The following subsections discuss those automated translation tools.

3.1 Translation from Behavioral Diagrams

The automated translation from behavioral diagrams to CSP specifications stems from n-ary tree representation. This tree describes exactly the diagram structure and the mutual relations among particular diagram objects. Each node of the tree represents either an event or a process in CSP, so that when browsing the tree in correct order, the tool generates a CSP specification of the diagram as discussed bellow. Another dynamic structure, a dynamic list, stands for the synchronization points and communication channels.

3.1.1 The dynamic structures

The behavioral diagram consists of objects and mutual relations among the objects. Each object is in relation with one or more other objects. An n-ary tree provides the behavioral diagram representation in the tool's run-time memory, in which root node of the tree represents process start, while the leaf nodes represent process ends. Nodes in between the root node and the leaf nodes stand for particular events of the behavioral diagram. Each tree has only one root node, but it may have more leaf nodes. Each process is described by a particular tree: for instance when the system consists of three processes then its representation contains three trees.

Each node is an object with particular information stored inside. Pointers to other nodes belong to this information. When performing depth-first search on the tree, the tool generates a CSP representation of the tree. Each node of each tree contains the name of an event or the name of a process. By climbing down all the trees and recording those names, the tool generates the related CSP representation of the complete system.

Because the automated translation tool stores both the diagram and the tree, it is useful to include the diagram information directly in the tree. Concurrently, the positions of the graphical elements, i.e. process start, process end, and events, have to be stored also. Moreover, when an event is assigned to a mutual relation, it necessitates to store that event, too. The best way how to manage this information is to divide the tree into levels, where each even level represents process start, process ends or events, and each odd level represents mutual relations. Figure 4 depicts an example of a divided tree.

Inclusion of the diagram structure directly in the tree increases amount of information stored, because each node contains also the graphical position of itself on the drawing area. Moreover, an object type, such as start, relation, event, should be included.

The synchronization points are represented as dynamic lists. Each element of the list contains the synchronization alphabet and pointers to processes which should synchronize. A dynamic list represents also the communication channel but each element contains only the type of the channel, without pointers. The indication that a process communicates among the channel is stored directly in the n-ary tree's proper node.

3.2 Translation from application source code

The translation from application source code into CSP specification stems from grammar-based compilers. The grammar exactly describes which source code structures the compiler accepts. Applying corresponding syntax analysis, the compiler transforms input source code into compiler's inner variables. Using

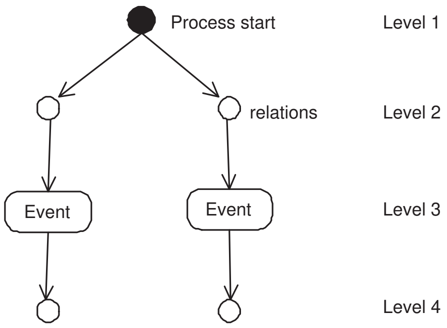

Figure 4: Divided object tree

those variables together with the knowledge of their meaning, the compiler generates the related CSP specification.

3.2.1 The Grammar

Each compiler grammar exactly defines which source code structures the compiler accepts. The grammar recognizes only basic program statements such as function call, conditional, and cycle. Recursive statements are accepted also. To simplify the grammar, some details of syntax are omitted. Standard rewriting rules define the grammar, where symbols written with lowercase letters represent terminals and symbols written with uppercase represent nonterminals.

The grammar:

```
S ->      IF
          | WHILE
          | id IDCONT
IF ->     if (condition) CODE  IFCONT
IFCONT -> else CODE
          | e
CODE ->   S
          | {BODY}
          | ;
IDCONT -> = value;
          | (params);
BODY ->   S  BODY
          | e
WHILE ->  while (condition) CODE
```

The compiler browses the input source code on-the-fly respecting the grammar. When top-down parsing reaches a terminal symbol, the compiler generates the related CSP specification of previously processed source code.

3.2.2 Generating CSP specification

The compiler performs syntax analysis of the source code on its input. Whenever the compiler reaches a terminal symbol in the grammar, the CSP specification of this terminal is generated. The symbols processed before are stored in compiler's inner variables to support generating the CSP specification. Parsing the source code, the compiler pushes each occurrence of a symbol on its stack. When generating the related CSP code, the symbols are popped from the stack.

Generating CSP from terminal symbols:

Note: $PROC1$ stands for the process name popped from the stack, $PROC2$ stands for the name of the process following the current process. This name is pushed on the stack. If no more program statements occur in the input source code, then $PROC2 = SKIP$

$$= value; - PROC1 = assign\_value \rightarrow PROC2$$

$$(params); - PROC1 = process\_params \rightarrow MY\_FUNC$$
$$MY\_FUNC = end\_of\_function \rightarrow PROC2$$

- event $end\_of\_function$ represents the end of a function stored by the id terminal symbol processing

$$\} \,|; - PROC1 = cond\_hold \rightarrow PROC3$$

- $PROC3$ stands for the name of the process representing the $BODY$; it is popped from the stack

$$\}else\ |; else - PROC1 = cond\_hold \rightarrow PROC3 \mid else \rightarrow PROC2$$

- $PROC3$ stands for the name of the process representing the $BODY$; it is popped from the stack
- $PROC2$ stands for the name of the process representing the $BODY$ of the else part; it is pushed on the stack

Other terminal symbols used in the grammar do not generate CSP specification directly. These symbols initiate only pushing process names on the stack and storing related variables into the symbol table.

4 Case study

The introduced case study describes cash dispenser's formal specification utilizing behavioral diagrams to specify the system's behavior. The complete specification consists of three parts: the cash dispenser, the bank, and the customer. Each of these parts contains particular processes which communicate with other processes utilizing two communication channels. The first channel serves to send messages from customer to the cash dispenser, the second to exchange information between the cash dispenser and the clearing house. Except communication channels, some processes synchronize on particular events. The synchronization symbol in the diagram depicts this relation between processes and also specifies the synchronization events. This section includes the behavioral diagram specification of the system and the related CSP specification generated by the developed tool.

4.1 The Cash Dispenser

This subsection describes the formal specification of the cash dispenser depicted in figure 5. The specification consists of seven processes. These processes, sequentially composed, represent the cash dispenser's behavior. The following paragraphs describe particular processes.

CASHMACHINE - this is the main process of the cash dispenser. The processes denotes insertion of a credit card, language selection and finally the process receives a message prom the customer representing the PIN related to the inserted card. After receiving the message, the authorization process starts.

AUTHORIZE - this process sends the from customer received PIN number to the bank for authorization. If the authorization fails, the process activate the CLOSE procedure, otherwise the process COMMAND is started.

COMMAND - the cash dispenser offers two choices to the customer: either to ascertain his bank balance or to withdraw money. The events *balance* and *value* synchronized with customer determine which action follows.

BALANCE - this process performs the event *balance* synchronized with the bank and then receives the account balance. After printing out the balance the process activates the procedure CONTINUE.

VALUE - this process receives a message from customer which specifies how much money to withdraw. This value is then forwarded to the bank for authorization. If the account provides enough money, then the cash dispenser spends the requested amount to the customer, otherwise the user haves to input lower request.

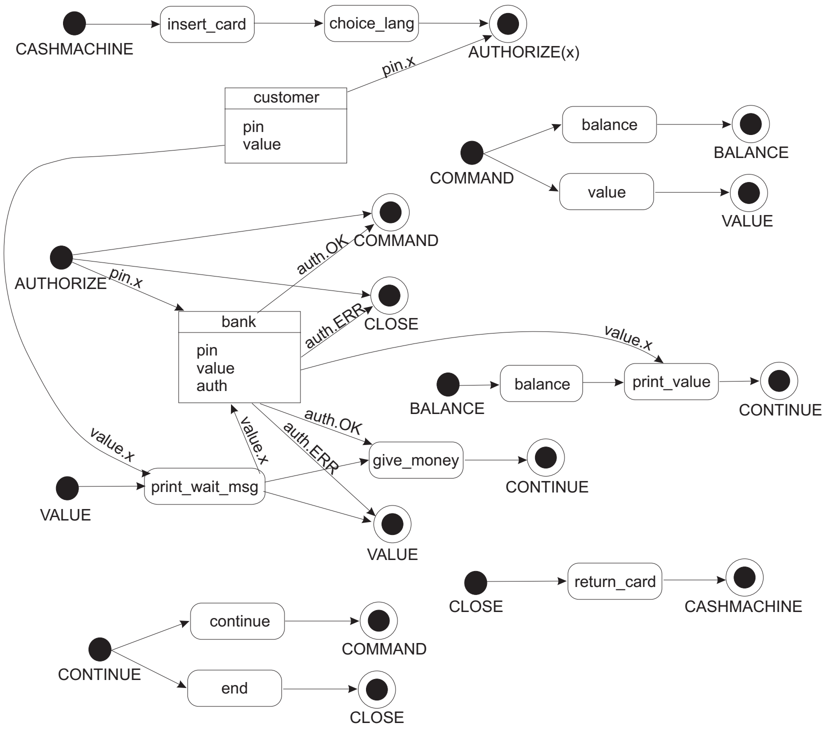

Figure 5: The cash dispenser specification

CONTINUE - after performing the requested command, the customer can either input another command or end his activity with the cash dispenser.

CLOSE - this process represents the returning of the credit card and the cash dispenser is ready to accept another customer.

4.2 The Bank

A single process with alternative behavior describes the formal specification of the bank. The process utilizes the bank communication channel to exchange information with the cash dispenser. The behavior of the process offers three choices. The first choice receives the PIN from the cash dispenser, then checks the PIN number and finaly sends a message to the cash dispenser whether the PIN agree or not. The second choice denotes the account balance detection. The process receives the requested amount of money, check the balance and sends a message whether the account provides enough money or not. The last choice synchronizes with the cash dispenser BALANCE process. The bank process finds

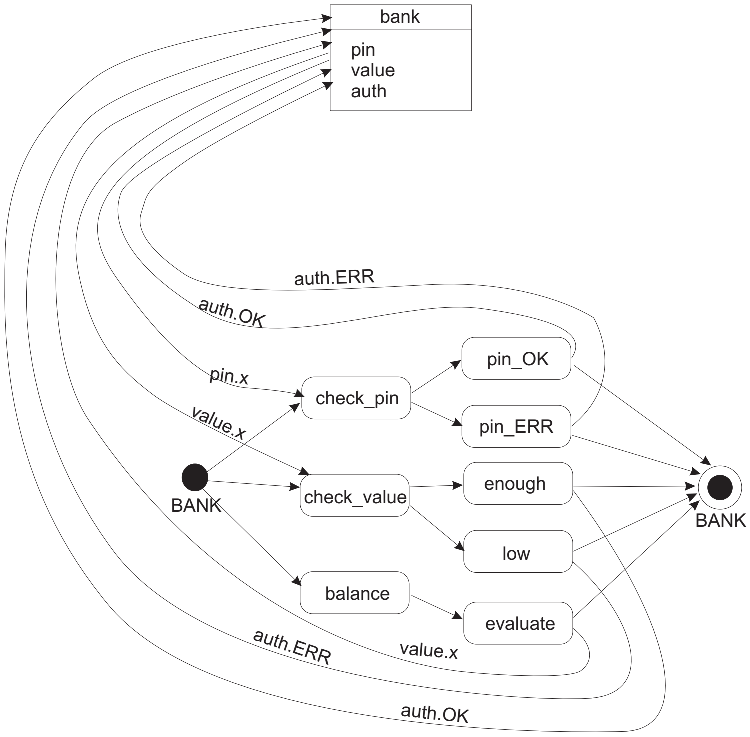

Figure 6: The bank specification

out the balance and then sends the value to the cash dispenser. After finishing, all choices returns back to the BANK process and can perform other requests. Figure 6 depicts the specification of the BANK process.

4.3 The Customer

The customer procedure consists of four processes and represents a sample behavior of a customer operating the cash dispenser. The customer procedure communicates with the cash dispenser utilizing a communication channel and synchronizes with other processes on particular events. The customer's processes are discussed bellow. Figure 7 depicts the customer procedure.

CUSTOMER - this process represents the main process of the customer procedure. The customer inserts his credit card, chooses language and enters his PIN.

CUSTOMER_COMM - this process decides which command will be performed. Either the account balance will be printed, or cash will be withdrawed,

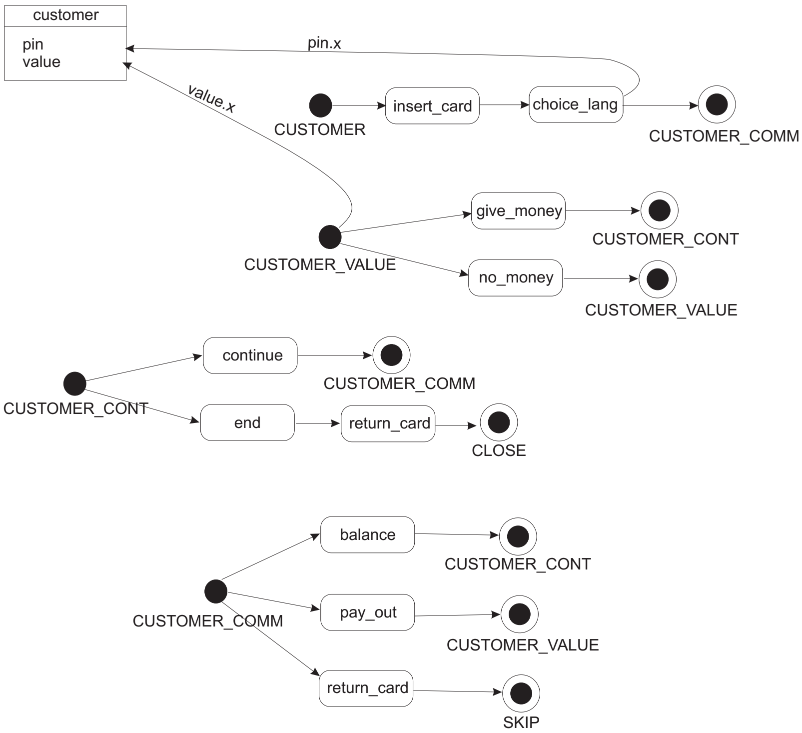

Figure 7: The customer specification

or he ends his operation and takes his card.

CUSTOMER_VALUE - the customer enters the requested value. If the cash dispenser gives money to the customer then the customer can choose the next behavior, otherwise he putted in too high amount and have to repeat the input with lower request.

CUSTOMER_CONT - the customer can either continue operating the cash dispenser and input new commands or take his card and finish his behavior.

4.4 The complete system

When composing these three subsystems into one complete system with communications and synchronizations then the behavioral diagram specifies the complete system. Figure 8 shows the behavioral diagram of the complete system. As noticeable from the diagram, when specifying more processes the diagram becomes complicated. Therefore it is useful in further development to include

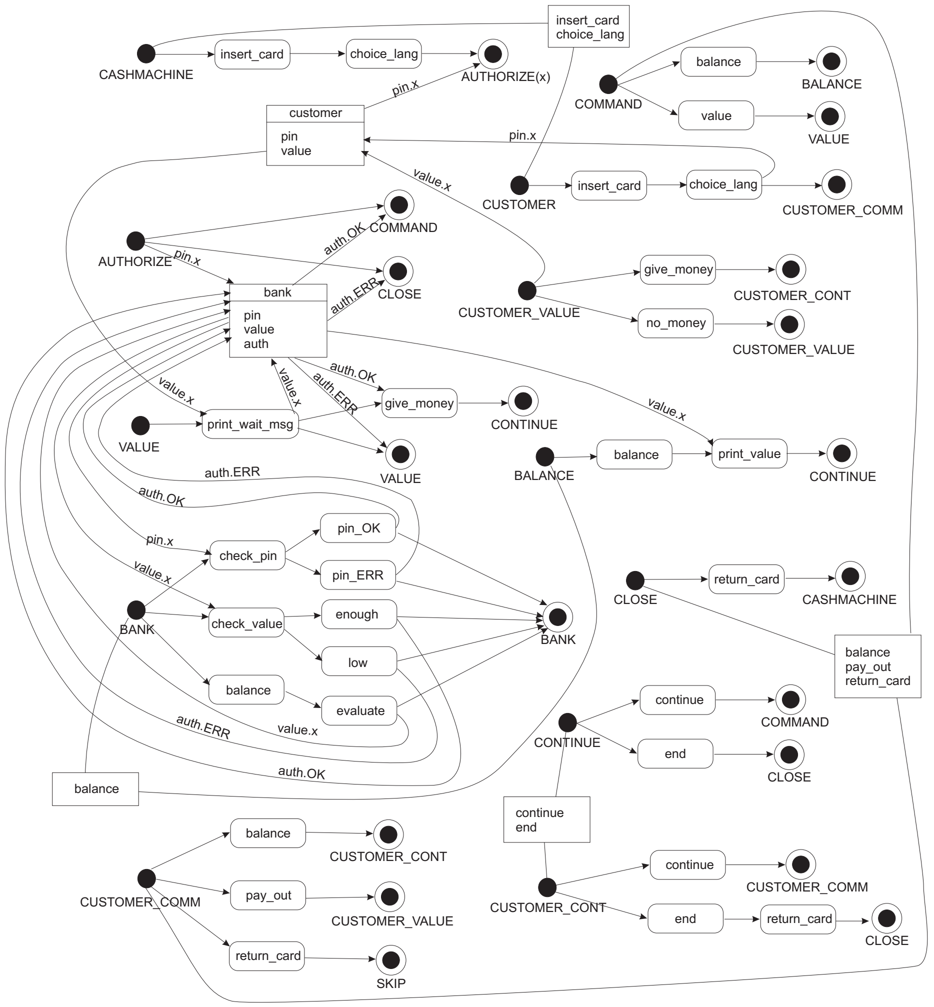

Figure 8: The complete specification

identities in the behavioral diagrams. So can be specified single communication channel on more places, for example.

After specifying the system utilizing the behavioral diagrams the tool generates the related CSP specification:

$$
\begin{aligned}
CASMACHINE &= insert\_card \rightarrow chioce\_lang \rightarrow customer?pin.x \\
&\quad \rightarrow AUTHORIZE(x) \\
AUTHORIZE &= bank!pin.x \rightarrow
\end{aligned}
$$

$$
\begin{aligned}
& (bank?auth.OK \to COMMAND \\
& \mid bank?auth.ERR \to CLOSE) \\
CLOSE &= return\_card \to CASHMACHINE \\
COMMAND &= balance \to BALANCE \mid pay\_out \to VALUE \\
VALUE &= customer?value.x \to print\_wait\_msg \to \\
& \to bank!value.x \to \\
& (bank?auth.OK \to give\_money \\
& \to CONTINUE \\
& \mid bank?auth.ERR \to VALUE) \\
CONTINUE &= continue \to COMMAND \mid end \to CLOSE \\
BALANCE &= balance \to bank?value.x \to print\_value \to \\
& \to CONTINUE \\
BANK &= bank?pin.x \to check\_pin \to \\
& (pinok \to bank!auth.OK \to BANK \\
& \mid pinerr \to bank!auth.ERR \to BANK) \\
& \mid bank?value.x \to check\_value \to \\
& (enough \to bank!auth.OK \to BANK \\
& \mid low \to bank!auth.ERR \to BANK) \\
& \mid balance \to evaluate \to bank!value.x \\
& \to BANK \\
CUSTOMER &= insert\_card \to choice\_lang \to customer!pin.x \\
& \to CUSTOMER\_COMM \\
CUSTOMER\_COMM &= balance \to CUSTOMER\_CONT \\
& \mid pay\_out \to CUSTOMER\_VALUE \\
& \mid return\_card \to SKIP \\
CUSTOMER\_CONT &= continue \to CUSTOMER\_COMM \\
& \mid end \to return\_card \to SKIP \\
CUSTOMER\_VALUE &= customer!value.x \to (\\
& give\_money \to CUSTOMER\_CONT \\
& \mid no\_money \to CUSTOMER\_VALUE) \\
CASHMACHINE\ {}_A\|_A & \ CUSTOMER \\
CUSTOMER\_CONT\ {}_B\|_B & \ CONTINUE \\
BALANCE\ {}_C\|_C & \ BANK \\
CUSTOMER\_COMM\ {}_D\|_D & \ COMMAND_D \ \|_D\ CLOSE
\end{aligned}
$$

$$
\begin{aligned}
A &= \{insert\_card, choice\_lang\} \\
B &= \{continue, end\} \\
C &= \{balance\} \\
D &= \{balance, pay\_out, return\_card\}
\end{aligned}
$$

5 Related work

Development of a front-end interface for generating formal models represents the major part of the work. The work of Muan Yong Ng and Michael Butler [5], and similarly, the work of Charles Crichton, Jim Davies, and Alessandra Cavarra [9] are closely related to this contribution. In both publications the authors represent UML statechart diagrams using CSP, but the synchronizations of processes and communication channels are represented utilizing class diagrams. That approach strictly uses UML diagrams but the information about process concurrency is remote and, therefore, the process interaction is not evident. Moreover, the semantics of class diagrams have to be modified for this purpose.

Other approach was introduces by Christie Bolton and Jim Davies in [6] presenting formal behavioral semantics for activity graphs. The paper illustrates, using a simple example, how this semantics may be used to verify the final class model description and its consistency.

6 Conclusions

First step in system verification consist in specifying requirements on the system, second step represents creation of system's formal model. To be suitable for verification, the model has to fit system requirements that must be satisfied for system correctness. On the other hand, the model should abstract those properties which don't influence the system correctness to simplify the verification. Although the verification process represents a time-consuming procedure, it is applied in a great number of current systems because finding and eliminating of consequent errors is much more expensive.

This contribution focuses on automated support for formal specifications using the CSP algebra. Two approaches reviewed offer automated processing. The first approach starts from behavioral diagrams, the second one from application source code. In the first case, the developed tool represents a process in the diagram as an n-ary tree. Depth-first search on this tree generates the CSP specification for this diagram. In the second case, the developed compiler translates an application source code into its CSP specification using a grammar exactly defining acceptable program structures. Processing syntax analysis of the source code, the compiler creates inner variables forming symbol table and stack. Using those variables and rewriting rules of the grammar, the compiler generates resulting CSP specifications. The first touch experience with the prototype implementation of those translators seems promissing. The limitations of the compiler can be reduced by re-designing the grammar.

When we extend the CSP algebra by state variables, the expressibility of this formalism may grow, so that more precise specifications can be generated. On the other hand, such extension requires to design more compound grammar and a new implementation of the compiler.

Acknowledgement

The research has been supported in part by Grand Agency of the Czech Republic in frame of the grant GACR 102/02/1032: Embedded Control Systems and their Inter- Communication, and by the Research Intention No. JC MSM 262200022.

References

1. Hoare C.A.R.: Communicating sequential processes, Prentice-Hall 1985, ISBN 0-13-153271-8
2. Schneider, Gay: Concurrent and real time systems, http://www.cs.rhbnc.ac.uk/books/concurrency/course/index.html, 2001
3. Clarke, E.M.,jr., Grumberg, O., Peled, D.A.: Model checking, The MIT Press, London, 2000, ISBN 0-262-03270-8
4. K. Havelund, N. Shankar, Experiments in Theorem Proving and Model Checking, Formal Methods Europe FME '96, Springer-Verlag, Oxford, UK. March, 1996, Pages 662681
5. Muan Yong Ng, Michael J. Butler: Tool Support for Visualizing CSP in UML. ICFEM 2002: 287-298
6. Christie Bolton, Jim Davies, Activity Graphs and Processes, In W. Grieskamp, T. Santen and W. Stoddart, editors, Proceedings of IFM 2000. Springer, 2000
7. Johan Lilius, Iván Porres Paltor, The semantics of UML state machines, Technical Report 273, Turku Centre for Computer Science TUCS, Turku, Finland, June 1999
8. M. Yanguo Liu, I. TRAORE, PVS Proof-Patterns for UML-Based Verification, IEEE ECBS Conference, Workshop on Formal Specification of Computer-Based Systems (FSBCS), April 8-11 2002, Lund, Swedden, pp 9-19
9. Charles Crichton, Jim Davies, Alessandra Cavarra, A Pattern for Concurrency in UML, Oxford University Computing Laboratory, England, December 2001
10. Scuglik, F.: Formal specification and verification of already composed systems, VUT Brno 2002, ISBN 80-214-2116-9
11. Scuglik Frantisek: Diagram Based Formal Specification using CSP, In: Proceedings of the 9th Conference and Competition STUDENT EEICT 2003, Brno, CZ, FEKT BUT, 2003, p. 629-633
12. Scuglik Frantisek: Comparing CSP representation and First order logic, In: Proceedings of the 9th Electronic Devices and Systems Conference EDS'02, Brno, CZ, BUT, 2002, p. 341-344