CSP-based modelling for self-adaptive applications

Szilárd Jaskó, Gyula Simon, Katalin Tarnay, Tibor Dulai, Dániel Muhi

University of Pannonia, Department of Electrical Engineering and Information Systems
jasko.szilard@uni-pen.hu

Keywords: self-adaptive, CSP based model, communication system, sensor network

In this paper, a CSP-based modeling approach is presented for self-adaptive systems, the proposed solution supports self-configuration, self-learning and testing as well. The behavioral elements of the system are described using the CSP language. A simple service is also defined which supports self-adaptation of subsystem components by learning from other system components. The efficiency of the system is demonstrated in two practical applications: an adaptive communication discovery protocol and a sensor networking application are implemented using the proposed approach.

1. Introduction

The complexity of information-based systems is growing continuously, therefore new approaches are needed to tackle the arising problems. Self-adaptive thinking is one of the promising ways to mention, where the system is able to evaluate its own behavior, make decisions to alter this behavior, and perform the necessary reconfigurations in order to improve its performance. Such changes may be required either because of the changing environment and requirements, or because of the lack of sufficient knowledge to solve the problem.

Self-adaptive systems are usually model-based systems. Several model representations can be used to describe the behavior of systems, depending on the type of system and application. For description of systems with heavy communication between its subsystems, an elegant and efficient way is the usage of the Communicating Sequential Processes (CSP), which a process algebra-based mathematical formalism.

Efficient self-adaptive systems can be created if there are efficient tools for the support of learning, testing, adapting, and configuring methods. For reliable systems the behavior and the working flow of the system have to be exact and provable. CSP provides efficient tool sets for specification, verification and testing.

With CSP, not only the communication protocols can be described efficiently, but the behavior of the system as a whole as well. It is particularly true for event-based systems with heavy intercommunication need.

In this paper CSP-based modeling is proposed, which supports self-adaptive, self-configuring, self-learning and even self-testing behavior of complex systems. The proposed system uses a simple and robust learning mechanism: system components can discover new behavioral elements (e.g. communication protocols, algorithms etc.) in their neighbor's knowledge base, and, if necessary, these rules can be learned. The proposed approach is illustrated by two practical applications: a self-adaptive communication protocol and a sensor networking data acquisition system.

The outline of the paper is the following: Related research is briefly summarized in Section 2, and then CSP is reviewed in Section 3. Section 4 introduces the proposed CSP-based self-adaptive system, and its ability to support self-adaptive behavior is described. In Section 5 two practical applications illustrate the potential of the approach, and Section 6 concludes the paper.

2. Background

Managing complexity is the main driving force behind the application of self-adaptive systems. Self-adaptive systems with learning and teaching abilities can handle problems in a novel way: the designer does not need to build in all the required information for every possible case, but rather the system can handle the exceptions in run-time. Such systems are flexible and can adapt themselves to new challenges. The importance of the area is shown by the wide range of research and the high number of publications. Self-adaptive systems were created in many application areas, e.g. distributed services [1], mobile and next generation networks [2,3], self-organizing solutions [4], or even organic robot control architectures [5] and bio-inspired approaches [6]. Self-adaptive protocols also bring new possibilities in protocol design and network organization [7,8].

In the field of sensor networks, in addition to the wide range of routing applications utilizing self-adaptive features (e.g. [9]), several interesting self-adaptive and self-organizing solutions were proposed: a self-organizing system was used to create a low-cost localization system [10]; adaptive self-diagnosis services were proposed to monitor network status and degradation [11]; or in monitoring applications self-configuring nodes prolong network lifetime while providing the required sensing service as well [12].

A good summary on the application of CSP can be found in [13]. An important usage of CSP is the specification and verification of fault-tolerant systems; an elegant verification technique for this problem was proposed in [14]. CSP was used in large industrial projects as well, e.g. the International Space Station project [15], and the testing of the avionic systems of an Airbus aircraft [16].

3. The CSP (Communicating Sequential Processes)

CSP [17-19] is a notation for describing concurrent systems and the interaction patterns between the component processes. It has a wide range of applications from programming languages [20] to verification of safety protocols [21]. A CSP system is built up from independently running sequential processes, which communicate with each other using message passing. The mathematical background is process algebra.

Each process has its own alphabet, which is the set of all the communication events the process might use. CSP defines several operators, by the help of those operators complex process descriptions can be easily constructed. We can express sequential communicational events (→), decision (*P∢p⊁Q* means: if *b* then *P* else *Q*), recursion, different choices (deterministic: □, or non-deterministic: *Π*), and simultaneous behavior of processes. The language has its own built-in basic processes and we can also use inner variables.

Let us see CSP expressions for example:

P1 = up → down → P1

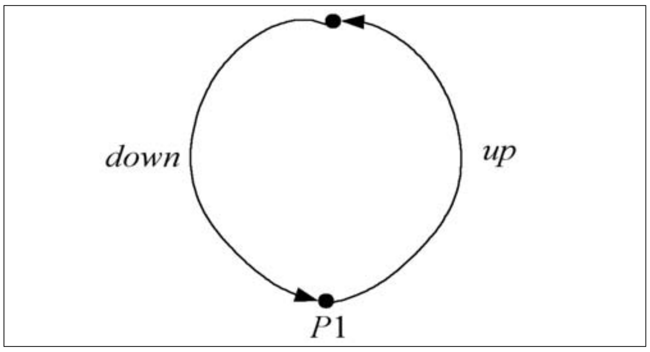


Figure 1. P1 process

We can see that *P1* is a process and it is shown in *Figure 1* *"up* and *down"* are events from the alphabet. The next example is:

COPY = left?x → right!x → COPY

It is a copy-machine and it defines two new things: the channels and an input and output operator. There are *"left"* and *"right"* channels in this example. *"?"* means input and *"!"* means output. So this machine copies *"x"* that comes in the left channel and puts it out in the right channel. Next rule is the external choice:

P □ Q

It is a process which offers the environment of the choice of the first events *P* and of *Q* and behaves accordingly. The combination of the previous examples is:

ExampleState =
c?rq → c!rp → ExampleState □ c?rpEnd → NextState

There is an environmental (external) choice in process *"ExampleState"*, denoted by "□". One of the possible ways: to get an *"rq"* event on channel *"c"* from an other process, consequently, the *"ExampleState"* process sends *"rp"* through channel *"c"* and after that stays in the same state. The second choice is to get an *"rqEnd"* signal on channel *"c"*, which shifts the process to state *"NextState"*. CSP gives us a tool for determining the trace of processes.

Trace of a process is the set of all the possible sequences of events that can happen during the process' life. An element of a trace is written between signs "<>". Trace of a process always contains the empty trace (<>). If we follow the example mentioned above, the traces of the process can be determined:

traces(ExampleState)=
(<>, <c?rqEnd>, <c?rq>, <c?rq, c!rp>,
<c?rq, c!rp, c?rqEnd>, <c?rq, c!rp, c?rq>,
<c?rq, c!rp, c?rq, c!rp, c?rqEnd>,
<c?rq, c!rp, c?rq, c!rp, c?rq>...)

There are various tools for checking CSP implementations. Animators make it possible to write arbitrary process descriptions and to interact with them [22], while refinement checkers explore all of the states of a process [23]. CSP is very useful to debug failures, discover deadlock or livelock, and is an ideal language for helping verification, validation and test processes [24,34,35].

4. CSP-based self-adaptive system

Imagine a system where every component can automatically recognize the other components' communication and working rules. They can learn from each other new methods of operation, and the whole system can adapt to changes in the environment. Of course, it is true that if at least one element of the system knows the right rule(s) that gives a solution for the occurring problem. From this point, the adaptation process is automatic. In this case the system contains self-adaptive, self-configuring, and self-learning elements indeed.

The system builds up from nodes that can communicate with each other as can be seen in *Figure 2*. Every node has a database that stores fields of CSP rules and conditions. The CSP rule gives the communication/working flow of the node. If the condition is true the corresponding CSP rule will be activated, and all other rules will be inactive at the same time. An example will be presented in Section 5.1.

There is a special node in the network that includes the referenced data. That is useful if some nodes try to

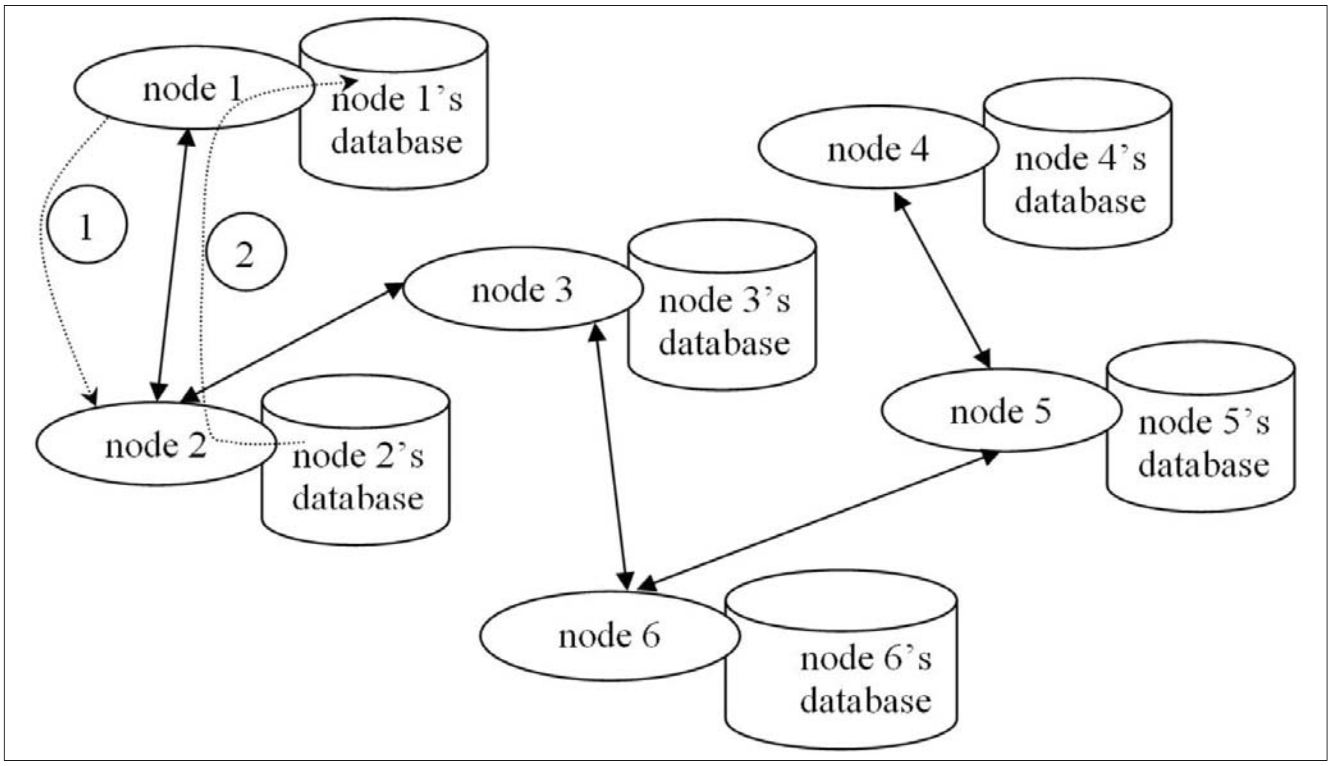


Figure 2. Structure of the system
Nodes can communicate to perform normal operation (solid lines), and also can exchange information to support self-adaptivity (dashed line). In the example Node 1 requests behavioral information from Node 2, and stores the rules in its own data base.

propagate an erroneous working mechanism. In this case, the special node orders the "bad" node(s) to delete the erroneous rule(s) and learn the good one. More details are found about it in Section 4.3.

4.1 Teaching/Learning

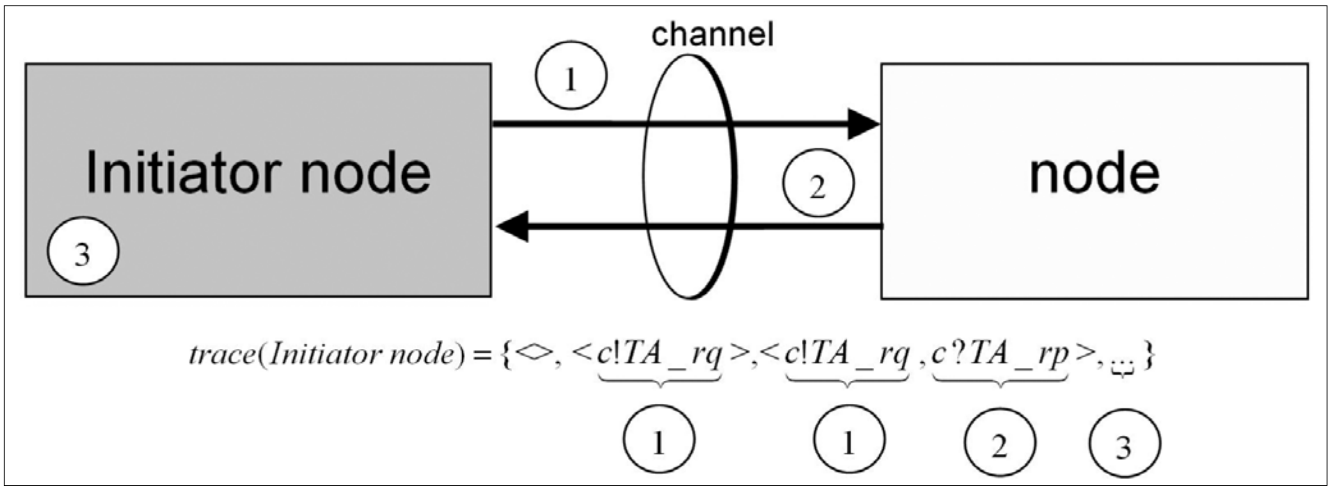
The rules are short and optimized CSP code, describing the communication and the processes in the system, and are stored in the local rule base of the nodes. Every node can check the database of other nodes. If it finds an unknown rule, it can learn it. This flow is illustrated in *Figure 2,* where Node 1 requests the rules of Node 2. Node 2 sends the requested data from its rule base. Node 1 processes the received data and if it finds new conditions and/or rules in the record, it will learn it. In Section 5.1 and 5.2 this process will be illustrated in practice.

4.2 Testing

An ideal test tool could recognize every communication device automatically and would be able to determine the protocol of the other party. This is a complicated task; its complexity can be compared to learning a new language from a grammar book and a dictionary.

Computers and mobile devices communicate with each other with the help of artificial languages. These

Figure 3. Background protocol" between initiator node and a node.
1. Ask the traces of the tested machine; 2. Send back the asked information; 3. Generate test cases and start the test


channel
Initiator node
node
1
2
3
trace(Initiator node) = {<>, <c!TA_rq>,<c!TA_rq, c?TA_rp>, ...}
1
1
2
3

protocols have several variations and versions. This fact can cause problems in many cases, because two different versions of the same protocol are sometimes not compatible.

Here, the inter-operability and the cooperation between devices are difficult. An efficient solution would be to be able to recognize the protocols used by different devices, and be able to learn and use them in the subsequent communication. The protocol and the version are recognized by extra information stored in the form of CSP rules. The CSP rules are requested by the initiative node with the help of the background protocol, shown in *Figure 3,* and described in detail in Section 4.3.

A simple and fast verification technique is when the tester node uses simple pattern matching on the received and its own CSP rules. If the rule does not match the authentic rule, the node will drop its own rule and will start the learning period by reconfiguring itself to the right CSP rule. A java-based application working with this principle will be shown in Section 5.1.

An additional, but more complicated possibility is the functional test of the used communication protocol. Based on the dictionary and the grammar book it can be decided whether the speaker is really speaking the right language: the tester has to ask questions and from the replies it can determine whether the speaker correctly speaks the language. Similarly, communication protocols can be verified by CSP rules, which include all the information required for test generation: the communication rules, interfaces and signals [25]. From the CSP model a behavior tree can be built and test cases can be generated, then the tester node can start to analyze the other node with the help of generated data [26-28].

4.3 The service process

In this section the service process will be described, which provides the means of learning and testing for the system components. In the system there always is a special node that has the reference data, being authoritative in conflict situations. Otherwise, every node is equal. A node can request data from another node using the following protocol, described in CSP:

$$\begin{gathered}
\mathit{Node} = \mathit{NormalOperation}\\
\mathit{NormalOperation} = c!\mathit{RD\_rq} \rightarrow \mathit{NormalOperation} \square c?\mathit{RD\_rp} \rightarrow\\
\mathit{ProcessData} \square c!\mathit{TA\_rq} \rightarrow \mathit{NormalOperation} \square c?\mathit{TA\_rp} \rightarrow \mathit{Tester}\\
\mathit{ProcessData} = \mathit{LearnNewComponent}\\
\nless \mathit{RD\_rp\_has\_new} \blacktriangleright \mathit{NormalOperation}\\
\mathit{LearnNewComponent} = \mathit{NormalOperation}\\
\nless \mathit{accomplished} \blacktriangleright \mathit{LearnNewComponent}\\
\mathit{Tester} = c!rq \rightarrow \mathit{Tester} \square c?rp \rightarrow \mathit{Tester} \square c!\mathit{rqend} \rightarrow \mathit{WaitTester}\\
\mathit{WaitTester} = c?\mathit{rpEnd} \rightarrow \mathit{NormalOperation} \square c!\mathit{rqEnd} \rightarrow \mathit{WaitTester}
\end{gathered}$$

where

c = *channel,* ? = *input,* ! = *output, TA = Test Alphabet,*
RD = Request Data, rq = request, rp = response
$rq \in \mathit{TestAlphabets} \cup \mathit{OtherRequest},\ rp \in \mathit{Responses}$

This automaton describes one half of the basic background communication of a node. In state *NormalOperation,* with the help of *RD_rq* (request for Requested Data) signal, a request is sent out to another node asking for its database. The reply arrives in message *RD_rp,* which is processed and the receiver learns the received rules. With the help of the *TA_rq* (request for Test Alphabet) signal, which can also be sent out in *NormalOperation* state, the node asks the other node to send information needed for testing.

This information comes in *TA_rp* (this is the active CSP rule); if the process is supported by the machine under test, a transition is generated to state *Tester,* where the generated test suite is running – in the automaton it is represented by *rq-rp* (request–response) message pairs. If testing is over, the initiator node sends an *rqEnd* signal. As a response, the other node sends back an *rpEnd* signal and it takes back the automaton into the initial state.

The other part of communication runs on the responder node. For sake of clearness, this part of the automaton is shown separately, as follows:

$$\begin{gathered}
\mathit{Node} = \mathit{NormalOperation}\\
\mathit{NormalOperation} = c?x \rightarrow \mathit{RequestData} \nless x = \mathit{RD\_rq} \ngtr\\
(\mathit{TestAlphabet} \nless x = \mathit{TA\_rq} \ngtr \mathit{NormalOperation})\\
\mathit{RequestData} = c!\mathit{RD\_rp} \rightarrow \mathit{NormalOperation}\\
\mathit{TestAlphabet} = c!y \rightarrow \mathit{TestState} \nless y = \mathit{TA\_rp} \ngtr \mathit{NormalOperation}\\
\mathit{TestState} = c?rq \rightarrow c!rp \rightarrow \mathit{TestState} \square c?\mathit{rqEnd} \rightarrow \mathit{TestEnd}\\
\mathit{TestEnd} = c!\mathit{rpEnd} \rightarrow \mathit{NormalOperation} \square c?\mathit{rqEnd} \rightarrow \mathit{TestEnd}
\end{gathered}$$

where

c = *channel,* ? = *input,* ! = *output, TA = Test Alphabet,*
RD = Request Data, rq = request, rp = response
$rq \in \mathit{TestAlphabets} \cup \mathit{OtherRequest},\ rp \in \mathit{Responses}$

The responder node starts from the *NormalOperation* initial state. If the signal *RD_rq* (request for Requested Data) arrives via channel *c* as signal *x*, the automaton steps to state *RequestData*, otherwise the state does not change. In state *RequestData* the requested data is send back in message (*RD_rp*). If the signal *TA_rq* (request for Test Alphabet) arrives via channel *c* as signal *x*, the system turns into *TestAlphabet* state, otherwise, the state does not change. In *TestAlphabet* state if the *TA_rp* is sent back via channel *c* as signal *y*, the automaton turns into state *TestState*. Otherwise, it steps back to state *NormalOperation*.

Testing happens in *Test State* communicating with pairs of request and response messages; if it is over, the tested machine gets an *rqEnd* message. It inducts the shift to state *TestEnd*, where it is also possible to get other *rqEnd* messages. After sending an *rpEnd* message the system gets back to its normal operation at the initial state.

5. Applications

5.1 Self-configuring communication system

Self-adaptive communication protocols will appear in the future in many application areas and they will be able to adapt to changes of the environment. In the following example a system is defined in which every node

Communication rule (CSP trace)	Condition	Status
<openChannel?Data>;<openChannel!Data>	*packetLost=0*	*active*
<openChannel?Data,openChannel!Ack>; <openChannel!Data,timerChannel!100,openChannel?Ack>; <openChannel!Data,timerChannel!100,openChannel?Timer, openChannel!Data,timerChannel!100,openChannel?Ack>; <openChannel!Data,timerChannel!100,openChannel?Timer, openChannel!Data,timerChannel!100,openChannel?Timer>	*packetLost>0*	*passive*

Table 1. The database of a communicating node

can test its communication partner and thus use the appropriate protocol. Simple examples include adaptivity to channel quality (in a noisy channel more robust protocol must be used); or channel safety (in a safe channel encryption is pointless; otherwise cryptographic defense of data is inevitable.)

In the demo application every communicating node has a database. This structure was defined in Section 4 and this scenario follows it. One of them is the part of describing communication rules. Here we use CSP traces, because it is equivalent to the standard CSP language, it is compact and easy to interpret (naturally, redundancies are removed from the traces for sake of compactness.) The trace(s) is/are chosen from the set of traces about the following view-points:

– the trace will be optimal as it is available,
– cover the given working/running mechanisms.

The traces give us a further advantage. If the used trace is finite we have good chance that the program of the node is livelock free. Of course, there are many CSP traces that are infinite. In this case, it has to be modified manually to work right. So the trace works like an indicator in the system and shows us if the program code includes some fatal errors. An example is presented in *Table 1.* This example includes an extra element and a status part. It is just an indicator that signals to the user which rule is active.

The simple example database contains two rules. The first (currently active) rule defines the behavior of the system when the communication channel is of good quality *(packetLost=0)*; in this case received messages are not acknowledged. The second rule explains the expected behavior if the message loss rate is unacceptable *(packetLost>0)*; in this case received messages must be acknowledged by the receiver, otherwise the sender node retries the transmission two times, after a backoff time of 100. The channel quality can be measured by any appropriate way (not included in the description); in the example condition the measured *packet Lost* variable is checked.

The main advantage of this self-adaptive communication scheme is its ability to adapt to any extreme environment without re-planning the whole system. Only a new record, containing the communication rules of the nodes has to be added to the database and the entire system will work according to the changed parameters. Naturally, not only a client/server connection can be controlled this way but the whole networked system can update its communication rules.

Note that a communicating node can never know precisely which communication rule (known or unknown) is used by its partner at the moment. The test functionality described in Section 4.2 can help in this case, because the client is able to check the other party's communication protocol, or it can learn it if that method is unknown yet, as shown in Figure 2.

In this simple example the server (the node with the authentic data in case of conflict) has rules, shown in Table 1. The client starts its operation with its database empty thus initializes the synchronization process of the communication rules. The server sends the communication rule base to the client. The client learns the new records and finishes the synchronization process.

The log of the operation can be seen in *Table 2.* In this example only a server-client communication was used, but it can be naturally extended to a larger network. The demo program, developed in Java, can be downloaded from [29].

5.2. Self-adaptive sensor networks

Sensor networks are special computer networks, containing potentially hundreds or thousands of embedded

Table 2. Server and client node logs

server node	client node
	Start the synchronization process of the communication rules
Send the communication rule(s) to client...	*Ask the server communication rule(s)...done.*
	Learn...done. The synchronization process is finished.

sensor nodes (called motes). Every mote is a small, usually battery-powered device, built around a low-power microcontroller running at a few MIPS with a few kilobytes of RAM, and is equipped with a wireless transmitter and the application-specific sensing capabilities [9]. Self-adaptivity is a widely researched area of sensor networks. Since inter-mote communication is usually done by ad-hoc networking, network elements must discover and adapt to unknown and potentially changing network topology.

Other interesting solutions include adaptive resource allocation [30], clustering [31], or automatic topology control [12], to achieve longer network lifetime. However, self-adaptivity is not provided at the application level. Currently the only way to change the application is the (in-network) reprogramming of the motes, where the whole memory footprint (usually tens of kilobytes) must be downloaded, inducing a serious overload on the network. CSP-based modeling provides an elegant way to include high-level self-adaptive properties in the network. The following proof of concept application was developed for Mica motes [32] running TinyOS operating system.

In the network every mote has a small built-in CSP interpreter that can translate modified CSP traces. Instead of the full CSP description, the equivalent traces were used again to (1) simplify the complexity of the interpreter and (2) decrease the message sizes. In addition to the standard CSP code, few extensions were added for the sake of efficiency: the syntax of the decision operation was changed (syntax: i(condition)(true branch)(false branch)), and the for loop, as an inseparable event was added (syntax: f(control expressions){body}).

The application layer of the motes becomes self-configuring with the help of this solution. Every mote can detect the actual program to be run, and motes can learn new application pieces, if necessary. Thus the operation can be adapted to the actual requirements. The demo application contains a simple sensor (light sensor). In mode 1 motes measure the ambient light and each mote in the network learns the position of the brightest spot. Mode 2 is more complicated: here each node builds a list of the brightest N nodes.

The CSP rules describing the operation in mode 1 are shown in *Table 3*. The measurement process is described by the first rule: the measured data is stored and broadcasted to the network. The second rule describes the diffusion process: if the received measurement data contains brightest data than the stored one, the received data is stored and broadcasted. The third rule defines the network query.

The operation in mode 2 is similarly described in *Table 4*. The measurement and diffusion processes are more complicated – note the multiple-line rules.

The network is operated in mode 1 and if requirements change, the algorithm describing mode 2 can be diffused in the network, or part of it. The application was developed in the TinyOS environment and can be downloaded from [33].

6. Conclusions

This paper presented a self-adaptive framework using CSP-based models to describe behavioral elements in the system. The elements of the system can perform testing operations, and – if required –, can learn new rules from other system components, thus the whole system can adapt to changing requirements. The testing and learning are supported by a simple services process.

The feasibility of the described method was illustrated by two practical applications: an adaptive communication discovery protocol illustrated self-adaptive behavior of the system when the qualities of the communication channel changes. The sensor networking application illustrated self-adaptivity on application level: changing requirements induce changes in the application program.

Acknowledgement

This research was partially supported by the Hungarian Government under contracts NKFP2-00018/2005 and OMFB-00247/2007.

1.	*M,S[]=M[],c!S[];*
2.	*c?G[],i(G[0]>S[0])(S[]=G[],c!S[])();*
3.	*c?A,c!S[]*

Table 3. CSP rules in mode 1

Table 4. CSP rules in mode 2

1.	*M,f(i=0,i<x,i++){i(M[0]>S[i][0]) (f(j=i+1,j<x,j++){S[j][]=S[j-1][]},S[i][]=M[])()},c!S[][];*
2.	*c?G[][],l=0,f(i=0,i<(x+0),i++){i(G[k][0]>S[i][0]) (f(j=i+1,j<(x+0),j++){S[j][]=S[j-1][]}, S[i][]=G[k][],k++,l=1)(i(G[k][0]=S[i][0])(k++)())},i(l=1)(c!S[][])();*
3.	*c?A,c!S[][]*
	,where c=channel, M=measured data, S=stored data, G=got data, f = for, i = if, A=AskData, !=out, ?=in, []=array

Authors

SZILÁRD JASKÓ received his MSc Degree in Information Technology Engineering in 2002 at the University of Veszprém. Since then, he has been a PhD student at the same university. Since 2002, he has taught at the Department of Information Systems of the University of Pannonia. He is interested in protocol modeling and testing, self-adaptive technologies, mobile telecommunication and formal languages.

GYULA SIMON received his M.Sc. and Ph.D. in electrical engineering from the Budapest University of Technology, Hungary, in 1991 and 1998, respectively. Currently he is an Associate Professor at the University of Pannonia. His main research interest includes adaptive signal processing and sensor networks.

KATALIN TARNAY is the Professor of Mobile Communication and Telecommunications Software at the University of Pannonia. Her book "Protocol specification and testing" (1989) was published by Kluwer Academic Publisher in New York. She was the coeditor of the book "Testing communicating systems" (1999). Her current research field is protocol modeling and network management.

TIBOR DULAI received his MSc Degree in Information Technology Engineering in 2002 at the University of Veszprém. Since then, he has been a PhD student at the same university. Since 2002, he has taught at the Department of Information Systems of the University of Pannonia. His major research areas include protocol modeling, location based technologies, mobile telecommunication and formal languages.

DÁNIEL MUHI received his MSc Degree in Information Technology Engineering in 2002 at the University of Veszprém. Since then, he has been a PhD student at the same university. Since 2002, he has taught at the Department of Information Systems of the University of Pannonia. He is interested in protocol modeling, formal languages, e-learning systems.

References

[1] Conrad, M., Hof, H.J.:
A generic, self-organizing, and distributed bootstrap service for peer-to-peer networks.
In: IWSOS 2007 – Self-Organizing Systems.
Lecture Notes in Computer Science LNCS 4725,
Springer Berlin/Heidelberg (2007), pp.59–72.

[2] Djenouri, D., Badache, N.:
Cross-layer approach to detect data packet droppers in mobile ad-hoc networks.
In: IWSOS 2006 – Self-Organizing Systems.
Lecture Notes in Computer Science LNCS 4124,
Springer Berlin/Heidelberg (2006), pp.163–176.

[3] Walter, U.:
Autonomous optimization of next generation networks.
In: IWSOS 2007 – Self-Organizing Systems.
Lecture Notes in Computer Science LNCS 4725,
Springer Berlin/Heidelberg (2007), pp.161–175.

[4] Jelasity, M., Babaoglu, O., Laddaga, R.,
Nagpal, R., Zambonelli, F., Sirer, E.G.,
Chaouchi, H., Smirnov, M.I.:
Interdisciplinary research:
Roles for self-organization.
IEEE Intelligent Systems 21(2) (2006), pp.50–58.

[5] Mösch, F., Litza, M., Auf, A.E.S., Maehle, E.,
Großpietsch, K.E., Brockmann, W.:
Orca – towards an organic robotic control architecture.
In: IWSOS 2006: Self-Organizing Systems.
Lecture Notes in Computer Science LNCS 4124,
(2006), pp.251–253.

[6] Pietzowski, A., Satzger, B., Trumler, W., Ungerer, T.:
A bio-inspired approach for self-protecting
an organic middleware with artificial antibodies.
In: IWSOS 2006: Self-Organizing Systems.
Lecture Notes in Computer Science LNCS 4124,
Springer Berlin/Heidelberg (2006), pp.202–215.

[7] Harangozó, Z., Tarnay, K.:
FDTs in self-adaptive protocol specification.
In: Self- Adaptive Software: Applications.
Lecture Notes in Computer Science LNCS 2614,
Springer Berlin/Heidelberg (2002), pp.105–117.

[8] Ferscha, A.G.C.:
Self-adaptive logical processes:
The probabilistic distributed simulation protocol.
In: Proceedings of the 27th Annual Simulation
Symposium, IEEE Computer Society Press,
LaJolla (1994).

[9] Estrin, D., Govindan, R., Heidemann, J., Kumar, S.:
Next century challenges:
scalable coordination in sensor networks.
In: MobiCom '99: Proceedings of the 5th annual
ACM/IEEE international conference on Mobile
computing and networking, New York, NY, USA,
ACM (1999), pp.263–270.

[10] Bachrach, J., Nagpal, R., Salib, M., Shrobe, H.E.:
Experimental results for and theoretical analysis of
a self-organizing global coordinate system
for ad hoc sensor networks.
Telecommunication Systems 26(2-4) (2004),
pp.213–233.

[11] Li, H., Price, M.C., Stott, J., Marshall, I.W.:
The development of a wireless sensor network
sensing node utilising adaptive self-diagnostics.
In: IWSOS 2007: Self-Organizing Systems.
Lecture Notes in Computer Science LNCS 4725,
Springer Berlin/Heidelberg (2007), pp.30–43.

[12] Cerpa, A.E., Estrin, D.:
ASCENT:
Adaptive self-configuring sensor networks topologies.
IEEE Transactions on Mobile Computing,
3(3) (2004), pp.272–285.

[13] Peleska, J.:
Applied formal methods –
from csp to executable hybrid specifications.
In: Communicating Sequential Processes.
Lecture Notes in Computer Science LNCS 3525,
Springer Berlin/Heidelberg (2005), pp.293–320.

[14] He, J., Hoare, C.A.R.:
Algebraic specification and proof of
a distributed recovery algorithm.
Distributed Computing 2(1) (1987), pp.1–12.

[15] Urban, G., Kolinowitz, H.J., Peleska, J.:
A survivable avionics system
for space applications.
In: FTCS (1998), pp.372–381.

[16] Schneider, S.:
Concurrent and Real-time Systems the CSP Approach.
Wiley and Sons Ltd. (2000).

[17] Hoare, C.:
Communicating Sequential Processes.
Prentice Hall, NJ (1985).

[18] ROSCOE, A.:
The Theory and Practice of Concurrency.
Prentice Hall (1997).

[19] Abramsky, S.:
Interaction categories and
communicating sequential processes.
Prentice Hall (1994).

[20] INMOS: occam Programming Manual.
Prentice Hall (1984).

[21] Schneider, S., Delicata, R.:
Verifying security protocols:
An application of CSP.
In: SOFSEM'99: Theory and Practice of Informatics.
Lecture Notes in Computer Science LNCS 3225,
(2005), pp.243–263.

[22] Natanson, L.D., Samson, W.B.:
An animator for CSP implemented in HOPE.
In: Proceedings of the BCS-FACS Workshop on
Specification and Verification of Concurrent Systems,
London, UK, Springer-Verlag (1990), pp.575–594.

[23] Formal Systems (Europe) Ltd.:
Failures-Divergence Refinement:
FDR2 User Manual, (1997).

[24] Bin, E., Emek, R., Shurek, G., Ziv, A.:
Using a constraint satisfaction formulation and solution
techniques for random test program generation.
IBM Sytems Journal 41(3) (2002), pp.386–402.

[25] Jifeng, H., Hoare, C.A.R.:
SDL- and MSC-Based Specification and
Automated Test Case Generation for INAP.
Telecommunication Systems 20, pp.265–290.

[26] Engels, A., Feijs, L.M.G., Mauw, S.:
Test generation for intelligent networks
using model checking.
In: TACAS (1997), pp.384–398.

[27] Dulai, T., Jaskó, Sz., Muhi, D., Tarnay, K.:
CSP model of self-adaptive test system.
IWSAS, Washington D.C. (2003).

[28] Jaskó, Sz., Dulai, T., Muhi, D., Tarnay, K.:
Process-based model for test system.
ISDA, Budapest (2004).

[29] http://uni-pen.hu/docs/misc/javabaseddemo.zip

[30] Mainland, G., Parkes, D.C., Welsh, M.:
Decentralized, adaptive resource allocation
for sensor networks.
In: NSDI'05: Proceedings of the 2nd conference
on Symposium on Networked Systems Design &
Implementation, Berkeley, CA, USA,
USENIX Association (2005), pp.315–328.

[31] Jin, G., Nittel, S.:
UDC: a self-adaptive uneven clustering protocol
for dynamic sensor networks.
Int. J. Sen. Netw. 2(1/2) (2007), pp.25–33.

[32] Hill, J.L., Culler, D.E.:
Mica: A wireless platform
for deeply embedded networks.
IEEE Micro 22(6) (2002), pp.12–24.

[33] http://uni-pen.hu/docs/misc/tinyosbaseddemo.zip

[34] http://www.mcrl2.org/mcrl2/wiki/index.php/Home

[35] http://www.comp.nus.edu.sg/~pat/