AUTOMATYKA/ AUTOMATICS • 2012 • Vol. 16 • No. 2

http://dx.doi.org/10.7494/automat.2012.16.2.115

Konrad Kułakowski[*]

Concurrent systems modeling with CCL[**]

1. Introduction

A *Concurrent Communicating List (CCL)* is a *Clojure* language library that we proposed to facilitate the creation of models that can be first verified, then executed as a part of a *Clojure* application [11]. *CCL* models are written using a special lisp-like notation, which can be simultaneously interpreted as an executable piece of code and a formal model susceptible to formal analysis. Thanks to the close integration with *Clojure* [13], a modern, concurrent dialect of *Lisp* language [7] running on *Java Virtual Machine*, the *CCL* has access to the vast array of *Java* based libraries, including standard ones provided together with the *JVM* platform. This makes *CCL* attractive for all the programmers familiar with the *Lisp*-based languages looking for a programming platform offering high functionality, ease of integration and increased reliability. The *CCL* notation follows the process algebra approach [4]. Hence, as in other algebraic notations, the main structural element of *CCL*, the *named list* (*nlist*), may contain operator expressions, primitive calls and a reference to other *nlists*. *Nlist* is a list of expressions (in a *Lisp* sense) designed to be executed in the same order as their appearance in the *nlist*. The primitive expressions are *Clojure* function calls. The operator expressions extend the linear execution scheme with the ability of conditional execution, and of new execution thread creation etc.

The *CCL* library is freely available as a source code and the binary *JVM* classes[1] are ready to be plugged into the 3rd party *Java* application. The *CCL* distribution package contains the *CCL* library API allowing building, executing, and formal analyzing of the *CCL* models. There is also the simulation environment, *CCL Sim* [10] providing the programmer with the ability of controlled execution and step-by-step debugging of a model, and to simulate the model's external environment.

The *CCL library* consists of three parts. The first one, the common part, allows the definition of models, and includes functions and macros allowing the declaration of *CCL* expressions, communication queues, and primitives etc. The second part is responsible for model execution capabilities, and offers *API* for executing the *CCL* model from *Java* and

* AGH University of Science and Technology, Institute of Applied Computer Science

** This research is supported by the AGH University of Science and Technology contract no.: 11.11.120.859.

[1] All the CCL supplementary materials, including additional examples, API description, and the binary version of the library, are available on-line at *http://www.kulakowski.org/ccl*

Clojure. Finally, the third part of the *CCL* software bundle provides customers with the tools for carrying out formal model verification. In particular, it supports *LTS* (*Labeled Transition System*) generation, *LTS* browsing and visualization, deadlock finding, pairwise model bisimulation checking, and looking for the trace distinguishing two models. It is also compatible with the *CADP* toolbox [6] and provides the ability to export the model data in the *CADP/Aldebaran* form. The working environment for *CCL* is any *IDE* supporting *Clojure* software development including the *Clojure REPL* (read-eval-print loop) mode – an interactive *Clojure* shell allowing immediate creation and evaluation of a *Clojure* expression. The *Clojure* REPL mode is adopted by CCL as an interactive command line shell for carrying out formal model analysis and simulation.

2. CCL Library at a glance

The *CCL* syntax follows the *Clojure* and *Lisp* languages. For this reason all the *CCL* expressions are in the form of lists. The *CCL* model is composed of lists denoting processes, communication channels between them, and primitives implemented in the form of *Clojure* functions, which are called by the processes during the execution of a model. The key concept of the notation introduced by the *CCL* library is *nlist* expression denoting the sequence of operations to execute. The *nlist* operations are launched within the *CCL* processes defined with the help of the *concurrent composition* clause. The processes can be anonymous or named. The name of the process can be declared in the concurrent composition clause and follows the process declaration (Tab. 1).

Table 1
CCL – Basic Operations

Construction	Description

`(defn Foo [] body-expr)`
`(reg-as-prim Foo)`
	Defines the *Clojure* function *Foo* and registers it as a *CCL* primitive.

`(def-nlist Boo`
`   (exp1 exp2 … expN))`
	Defines the *nlist Boo* executing its *nlist-body* i.e. the list of subsequent expressions: *exp1, exp2,…,expN*.

`(def-nlist (Roo :y)`
`(exp1(:y) … exp2(:y))`
	Defines the *nlist* named *Roo* with the initial parameter :y. The expressions in *Roo's* nlist-body can freely use the parameter :y.

`((:x (Foo))`
`     (Roo (+ :x :y)))`
	Defines the local *nlist* variable :x and initializes it to the value retuned by *Foo*, then starts execution of *Roo* with the input value set to the sum of :x and :y

`(? (cond-1) (nlist-1)`
`   (cond-2) (nlist-2)`
`   … (cond-N) (nlist-N))`
	The *conditional choice* operator allows the definition of the nlist-expression to be executed next depending on their condition expressions, i.e. if *cond-k* is the first true expression on the left then the *nlist-k* expression is to be executed.

`(?? X1 (nlist-1)`
`    X2 (nlist-2)`
`   … XN (nlist-N))`
	Within the random choice statement, *nlists* are picked for further execution randomly. The chance of being selected for *nlist-k* is given as: *Xk/sum(X1,…,XN)*.

`(| Moo :moo Goo :goo)`	As a result of execution of this expression two *CCL* processes have been launched, where the first process labeled: *moo* will execute the *nlist Moo*, whilst the second process: goo will execute the nlist *Goo*.

3. Process synchronization and communication

3.1. Synchronization queues

The fundamental synchronization and inter-process communication mechanism in *CCL* is a blocking queue. Depending on the adopted strategy and the state of the queue the processes attempting to read from or write into the queue can be blocked for a while or returned immediately. The use of the synchronization queue mechanism is possible through the set of queue access methods such as: *q-get*, *q-put*, *q-peek*, *q-try-put*, *q-size* and *q-capacity*. The last two operations do not change the state of the queue. The first of them *q-size* returns the actual number of elements in the queue, whilst the second one brings the answer to the question of the maximum size of the queue. The other four functions seem to be self-explanatory. The *q-get* function inserts the new element into the queue, *q-put* takes it back, *q-peek* returns the value of the first element from the queue but leaves it in place, and finally, *q-try-put* tries to insert a new element into the queue, but it returns *false* when it is impossible (e.g. the queue is full).

```
1.    (def-queue truck :size 5 :rb :wb)
2.
3.    (defn produce-goods [] (do (println "produce") true))
4.    (defn consume-goods [] (do (println "consume") true))
5.    (reg-as-prim send-goods receive-goods)
6.
7.    (def-nlist Producer ((produce-goods)
8.                         (q-put truck "data") Producer))
9.    (def-nlist Consumer ((receive-goods) (q-get truck) Consumer))
10.
11.   (def-nlist ProdCons (| Producer :prod Consumer :cons))
12.
13.   (nlist-model-execution ProdCons)
```

Fig. 1. Producer-Consumer model – an example of inter-process communication in CCL

The general communication scheme between two processes has been illustrated by the well-known *Producer-Consumer* example [1, 5] (see Fig. 1). There is one synchronization queue (Fig. 1., line: 1), two *Clojure* functions *produce-goods* and *consume-goods* (Fig. 1, lines: 3–4) registered as model primitives (Fig. 1, lines: 5) and three *CCL* nlists, *Producer* (Fig. 1, line: 7), *Consumer* (Fig. 1, line: 9) and *ProdCons* (Fig. 1, line: 11), which binds *Producer* and *Consumer* together. *Producer* performs two actions in a loop. First it calls the *produce-goods* primitive resulting in printing the word *"produce"*, and receiving printing success confirmation, and then it inserts the new word *"data" into* the queue. After that, the process *:prod* starts evaluating the *Producer* expression once again. The process *Consumer* is a twin-like *Producer*. The only difference is that *Consumer* calls *q-get* method and takes the element from the queue instead of inserting it. The queue is blocking at both its ends. Thus, depending on the number of elements in the queue, one of these two functions must wait for the other. The whole model can be executed by calling the *CCL API* function *nlist-model-execution* with the *ProdCons* expression as its argument.

Although the overall meaning of these six functions is clear, the problem occurs when one end of the queue (or both) is non-blocking, or if the queue length is unusual e.g. 0. In order to handle all of these cases the meaning of the four modifying functions *q-get*, *q-put*, *q-peek* and *q-try-put* has to be reexamined in the context of different queue locking policies and the variable length of the queue. For instance, when the queue's capacity is zero there is no sense in checking what is at the very beginning of the queue by calling *q-peek*. That is because this type of queue is always empty. On the other hand, it makes sense to use the blocking versions of *q-put* and *q-get* together. In such a case, *q-get* waits for *q-put*, and reversely *q-put* waits for *q-get*. For queues with the non-zero length, *q-peek* and *q-try-put* make sense, although the queue blocking policy on their ends is also important. For instance, if the queue is non-blocking on its front, then every *q-get* call is in fact non-blocking, so there is no sense in using *q-peek* since q-get provides the same functionality. All the possible variants of synchronization queues have been summarized in Table 2.

Table 2
Types of synchronization queues

Type Id	**Capa-city**	**Put/Get Read policy**	**Put/Get Write policy**	**Operation semantics – summary**
1	0	Non-block.	Non-block.	Since the size of the queue is 0 all the (non-blocking) operations are ineffective.
2	0	Blocking	Non-block.	The methods *peek* and *try-put* are ineffective, since at the given point of time, the queue is always empty. *Get* always blocks until the corresponding *put* is called. *Put* passes the value directly to the *get* function if it is pending, if not, *put* fails.
3	0	Non-block.	Blocking	*Peek* and *try-put* are ineffective. *Put* blocks until the *get* function is called. *Get* does not wait. If there is no pending *put* operation at the second end of the queue, *get* fails.
4	0	Blocking	Blocking	*Peek* and *try-put* are ineffective. *Get* and *put* wait for each other. This queue models the classical rendez-vous as known from CSP or CCS [6], [7]-ERROR.
5	c > 0	Non-block.	Non-block.	*Peek* and *try-put* are ineffective, since they work the same as *put* and *get*. *Get* fails when the queue is empty, and *put* fails when the queue is full. Both functions return immediately.
6	c > 0	Blocking	Non-block.	*Try-put* is ineffective since *put* is non-blocking. *Get* blocks when the queue is empty and waits for *put*. *Put* fails when the queue is full.
7	c > 0	Non-block.	Blocking	Reversely, *put* blocks when the queue is full. *Get* fails when the queue is empty.
8	c > 0	Blocking	Blocking	*Get* blocks when the queue is empty and waits for *put*, otherwise it takes an element from the queue. *Put* blocks when the queue is full, and waits for *get*, otherwise inserts an element into the queue.

3.2. Internal vs. External communication

The synchronization queues provide both an inter-process communication service, and the synchronization mechanism between them. In *CCL*, similar functions are implemented by a primitive call mechanism. The *CCL* primitive (*Clojure* function registered as parts of a model) can retrieve values from the model (attributes of the function call) and can also return the result back into the model. Of course, its call can also be blocking or non-blocking. Since the body of a *CCL* primitive implements algorithms that are not a part of the model, the primitive call serves as a communication mechanism between the model and the external world. Properties of this implicit communication channel are not specified formally, although it is usually known whether a particular function call is blocking or non-blocking.

Very often synchronization queues play the role of communication links between different parts of the model. In such a case, it is worth considering the division of the main model into sub-models. Thanks to such division the project becomes clearer and more manageable, and, importantly from a practical point of view, each of its parts can be analyzed separately. Thus, the analyzed problems become smaller and their formal verification requires fewer computer resources than working with the entire model. In *CCL,* splitting the model into sub-models is implemented by wrapping queue operations in a primitive call. The synchronization queue ceases to be a part of the model and begins to be a part of the model external environment. The operation of moving the synchronization queue outside the model is called *queue's externalization*, or simply *externalization*. As a result of externalization one massive model becomes a set of loosely coupled sub-models implicitly connected by the externalized queues. The selection of communication queues that need to be externalized is up to the designer and depends on the logic of the model.

In the context of the *Producer-Consumer* example, the queue externalization means that both ends of the *truck* synchronization queue need to be wrapped into the additional *Clojure* functions *send-goods* and *receive-goods* (Fig. 2, lines: 3–4), which substitute the methods *q-put* and *q-get* (Fig. 2, lines: 13 and 15). Of course, both new functions have to be declared as model primitives (Fig. 2, line: 5). The externalization mechanism introduces into the queue declaration the additional keyword *:ext* (Fig. 2, line: 1). All the queues flagged as *:ext* are external and their state does not affect the state of the entire model. As two separate models, *Producer* and *Consumer* can be executed separately (Fig. 2, lines: 16–17) although the previous syntactic form using the *ProdCons* expression as defined in (Fig. 1, line 11) is also admissible.

The *Externalization* process is reversible. The queue previously externalized can be incorporated back into the model. The reverse process will be called *internalization*.

To carry out *internalization,* the new communication channel (or channels) needs to be identified and the model on the other side of the communication channel (or channels) needs to be reconstructed. This is easy if the *internalization* only restores the state before earlier *externalization*. If not, the model on the other side of the communication queue needs to be built from scratch. In such a case the mechanism of internalization is involved in the expansion of the model boundaries.

```
(def-queue truck :size 5 :rb :wb :ext)

(defn-queue-wrapper send-goods truck :put)
(defn-queue-wrapper receive-goods truck :get)
(reg-as-prim send-goods receive-goods)

(defn produce-goods [] (do (println "produce") true))
(defn consume-goods [] (do (println "consume") true))

(reg-as-prim produce-goods consume-goods)

(def-nlist Producer ((produce-goods)
                      (send-goods "data") Producer))
(def-nlist Consumer ((receive-goods)
                      (consume-goods) Consumer))
(nlist-model-execution Producer)
(nlist-model-execution Consumer)
```

Fig. 2. *Producer* and *Consumer* sub-models as an example of *externalization*

4. Dataflow-centric CCL model lifecycle

The *internalization* and *externalization* mechanisms fit well into the top-down structural design approach proposed by *Yourdon* [15]. According to the *DFDs (Data Flow Diagrams)* based technique the first stage of modeling relies on preparing the *context diagram* showing all the interactions between the main process and terminators. In the *dataflow-centric CCL* approach, the *context diagram* should capture all the important interactions between the main model and its environment.

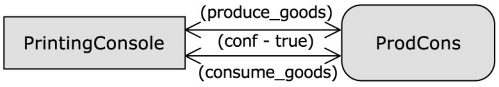


Fig. 3. *Context* diagram for *ProdCons* model[2]

Since this external communication is handled by primitive calls, the list of registered primitives determines the list of data flows between the main model and the external entities. In the case of the *ProdCons* example (Fig. 1) the context diagram (Fig. 3) consists of one external entity (*terminator*) *PrintingConsole* and *ProdCons* as the main model. Both

[2] The graphical notation used in the article is consistent with the *DFD* diagrammatic style proposed at http://yourdon.com/strucanalysis/wiki

diagram elements are linked by the *produce_goods* and *consume_goods* function calls returning confirmation that printing action was successful, which are, in fact, the message flows between *ProdCons* and *PrintingConsole*. After defining the context diagram for the *ProdCons* main model, its sub-models have to be identified. Initially, they are defined as internal *ProdCons* processes. At the moment they are not intended to be modeled and analyzed separately (Fig. 4). The explicit connection between *Producer* and *Consumer* implemented by a synchronization queue truck is specified as an arrow with the black arrowhead (Fig. 4).

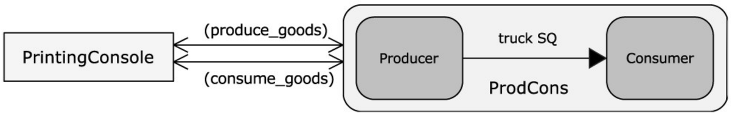


Fig. 4. *Context diagram* with *ProdCons's* sub-processes specified

Then, if there is a need for separate analysis of *Producer* and *Consumer,* the designer *externalizes* the *truck* synchronization queue, and removes the explicit data connection between both processes from the main model. In such a case *Producer* and *Consumer* can be represented as separate processes on the next level diagram *"Diagram 0"* (Fig. 5).

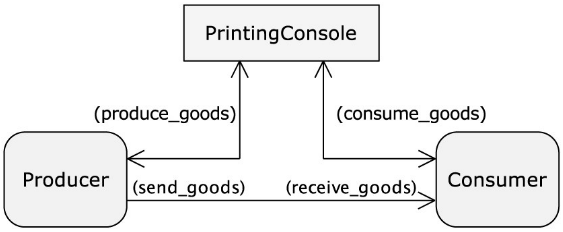


Fig. 5. *Diagram „0"* for *ProdCons* model

Of course the procedure of refinement and externalization can be reiterated. The next iterations may move the design process to the next levels of the *DFD* diagram hierarchy.

The procedure presented above can be generalized in the form of the cyclic *Dataflow-centric CCL lifecycle process* (Fig. 6). A single cycle consists of four stages:

- *Model refinement* – selected models are refined, sub-processes are defined. Data flows and synchronization dependencies between sub-models are identified.
- *Queue selection* – some of the newly identified data flows between sub-processes for further externalizations are selected.
- *Externalization* – selected queues are externalized.
- *Sub-model selection* – some of the new sub-models are selected for further refinement.

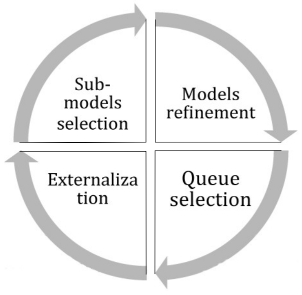


Fig. 6. *Dataflow-centric CCL lifecycle process*

The *Dataflow-centric CCL lifecycle process* as described above leads to a *DFD* diagrams hierarchy modeling dataflow and inter-process communication within the system as shown on Figure 7.

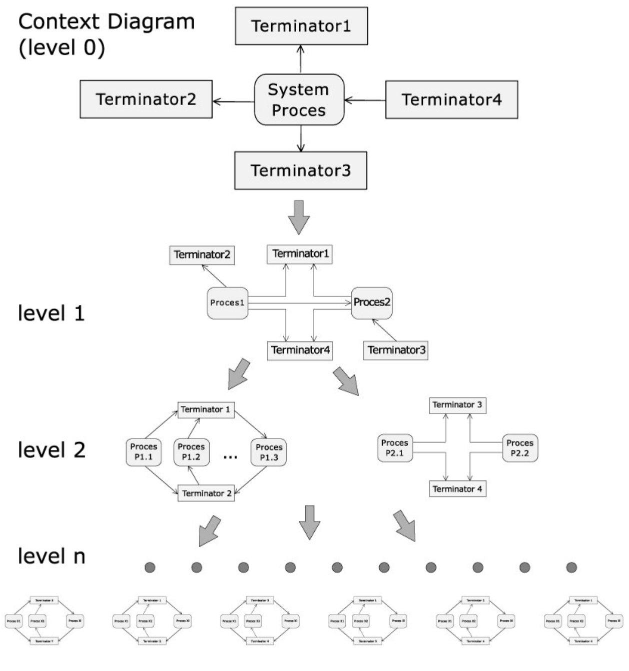


Fig. 7. *Dataflow-centric CCL lifecycle process - the hierarchy of DFD diagrams*

5. Notes on formal verification of the system

The *externalization* mechanism presented above brings a few interesting benefits. The first one is mostly structural. The two different *CCL* processes referring explicitly to the same synchronization queue are strongly linked with each other. It is assumed that the changes to one process can significantly affect the second process. This implies that both processes have to be modeled and tested together.

On the other hand, if the synchronization queue between two processes is externalized, then one process uses the other process as it would use any other part of the system assuming that whatever happens the other process will work correctly. In other words, it can be assumed (at least at the low level *DFD* hierarchy as shown on Fig. 7) that both processes do not need to be tested as if they were a single entity. Such an approach makes room for carrying *unit tests* covering one or more processes explicitly connected through synchronization queues. The other benefit of using *externalization* is connected with limiting the size of models, hence limiting the number of states specifying models behavior. Since one of the biggest obstacles for widespread use of formalisms is the *state explosion problem* [2]. E*xternalization* opens the possibility of effectively using formal methods for local checking of various temporal properties, such as deadlock freedom, safety, liveness etc. (Fig. 8b).

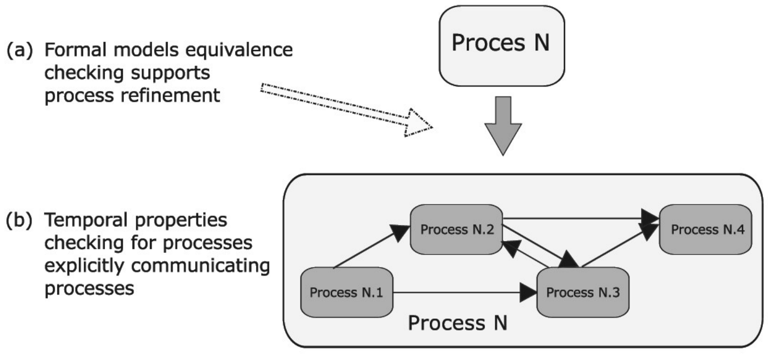
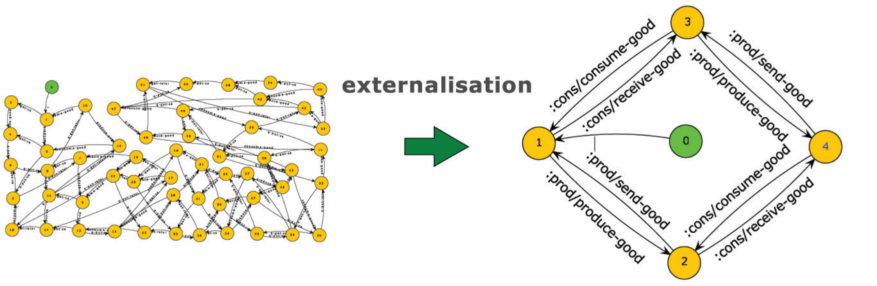

(a) Formal models equivalence checking supports process refinement
Proces N
(b) Temporal properties checking for processes explicitly communicating processes
Process N.1
Process N.2
Process N.3
Process N.4
Process N

Fig. 8. The place of *formal methods in Dataflow-centric CCL lifecycle process*

Formal methods can also actively support the model refinement phase (Fig. 8a) by providing efficient mechanisms such as the *weak bisimulation* checking [3, 12] or the trace equivalence checking [8, 9].

The effect of reducing the model's space states as a result of queue externalization can be observed even in the case of such a small model as *producer-consumer*. The first model (Fig. 1) with the explicit synchronization queue *truck* is described by the state space with 55 states and 103 transitions among them (Fig. 9a), whilst the second, being a combination of two sub-models, needs only 5 states and 9 transitions among them. Thus, the queue externalization brings a more than tenfold reduction in the number of states and transitions on the diagram, allowing the designer to abstract away from communication technicalities and quickly grasp the overall idea of the modeling system.



(a) LTS for Producer-Consumer model with the explicit synchronization queue

(b) The compound LTS for Producer and Consumer models The reduced number of states allows the designer to focus only on the most important elements of the model

Fig. 9. State space reduction as a result of externalization

6. Summary

In the article the *Dataflow-centric CCL lifecycle process* and the *CCL's externalization* concepts have been presented. The *externalization* mechanism, as presented in the paper, is crucial for managing complexity of the model. It meets the *Principle of Compositionality* [1], thus creating the ground for building complex, real-life systems. *Externalization* also supports the stepwise-refinement approach [14] to the model development. The presented *dataflow-centric CCL lifecycle process* is a practical implementation of the formal-method assisted stepwise-refinement paradigm using the *CCL library* notation.

Although the model development process, and mechanisms for model complexity reduction are implemented on the *CCL* platform, they should be easily customizable for other notations and languages. The clear indications of where and to what extent the formal model verification can be applied should contribute to further popularization of formal methods in software engineering practice.

References

[1] Aldini A. *et al.*, *A process algebraic approach to software architecture design*. sti.uniurb.it, Berlin, Heidelberg, 2009.

[2] Baier C. *et al.*, *Principles of Model Checking*. mitpress.mit.edu, 2008.

[3] Barbosa P. *et al.*, *Checking Semantics Equivalence of MDA Transformations in Concurrent Systems*. J Univers Comput Sci., 15, 11, 2009, 2196–2224.

[4] Bergstra J.A. *et al.*, eds: *Handbook of Process Algebra*. Elsevier Science, 2001.

[5] Fencott C., *Formal methods for concurrency*. International Thomson Computer Press, 1996.

[6] Garavel H. *et al.*, Cadp 2010: *A toolbox for the construction and analysis of distributed processes*. Tools and Algorithms for the Construction and Analysis of Systems – TACAS, 2011.

[7] Graham P., *ANSI Common Lisp*. Prentice Hall, 1995.

[8] Holzmann G., Smith M., *Automating software feature verification*. Bell Labs Tech J., 5, 2, 2000, 72–87.

[9] Karvi T. *et al.*, *Stepwise Development of Process-Algebraic Specifications in Decorated Trace Semantics*. Form Method Syst Des., 26, 3, 2005, 293–317.

[10] Kułakowski K., *CCL Sim, the simulation environment for concurrent systems*. To be appeard in Proceedings of Dependability and Complex Systems DepCoS–RELCOMEX, 2012.

[11] Kułakowski K., Szmuc T., *Concurrent Communicating Lists – executable modeling library*. Submitted to Journal of Functional Programming, 2012, 1–43.

[12] Milner R., *A calculus of communicating systems*. 1982.

[13] VanderHart L., Sierra S., **Practical Clojure**. Apress, 2009.

[14] Wirth N., *Program development by stepwise refinement*.

[15] Yourdon E., *Modern structured analysis*. Prentice Hall, 1989.