AGH University of Science and Technology
Department of Applied Computer Science
Al. Mickiewicza 30
30-059 Kraków, POLAND

Outline of CCL notation syntax

Konrad Kułakowski
AGH University of Science and Technology
Department of Applied Computer Science
Kraków, POLAND
kkulak@agh.edu.pl

Tomasz Szmuc
AGH University of Science and Technology
Department of Applied Computer Science
Kraków, POLAND
tsz@agh.edu.pl

Published online: 10.01.2013

DACS Technical Report No. 1/2013

Department of Applied Computer Science Technical Reports (DACS TR) series serves as an accesible platform for publishing research results of the staff members at DACS of AGH University of Science and Technology.

See the `http://www.kis.agh.edu.pl` for the series homepage.

Cover design: Marcin Szpyrka

LaTeXclass design: Marcin Szpyrka

Contact us at: `kkulak@agh.edu.pl`

DACS Technical Report No. 1/2013

Outline of CCL notation syntax*

Konrad Kułakowski
AGH University of Science and Technology
Department of Applied Computer Science
Kraków, POLAND
kkulak@agh.edu.pl

Tomasz Szmuc
AGH University of Science and Technology
Department of Applied Computer Science
Kraków, POLAND
tsz@agh.edu.pl

Abstract. *Concurrent Communicating Lists (CCL)* is the novel *Clojure/Java* based library, which facilitates executable modeling of concurrent systems. It provides a kind of semi-formal notation modeled on the *Lisp* language and its modern equivalent *Clojure*. Models designed with the help of *CCL* library can be simulated and executed, as well as formally verified, using provided library tools. The main aim of this report is to briefly collect all the main syntax features of *CCL* and explain them by examples. The report contains several short listings of code, describing various aspects of the *CCL* syntax and showing how the *CCL* model can be built up from pieces.

Keywords: CCL, Concurrent Communicating Lists, functional programming, executable modeling, formal methods

1 Introduction

The main purpose of the report is to present in a convenient way the basic syntax mechanisms provided by the *CCL* library. *CCL* is designed as a set of *Clojure* language declarations, functions and macro-definitons [6] forming an algebraic-like specification language. It allows programmers to design, prototype, test and verify their concurrent control applications. Like *Clojure*, *CCL* works on the top of the *Java Virtual Machine*, which provides *CCL* portability across different hardware platforms, and brings access to the large standard libraries of both *Clojure* and *Java*. Such tight integration makes possible mixing *CCL* code with *Clojure* and *Java*. In particular it is possible to execute a piece of code written in *Clojure* and *Java* from the *CCL* model, and reversely, the *CCL* model execution can be initiated from *Clojure* and Java..

CCL is well integrated with other software components such as *CCL Sim* [11] and *Robust* [9, 12, 10]. The *CCL Sim* enables model debugging and simulation in a graphical environment, whilst the integration with the *Robust* library allows different robotic constructions to be directly controlled from the *CCL* executable model [13].

The *CCL* models are conceptually similar to those known from process algebra. For this reason, many terms defined in the context of *CCL* have their counterparts in many well known process algebras, such as *CSP, CCS, ACP or LOTOS [14, 7, 2, 8].* In all of them, one of the basic concepts is a process understood as a *"behaviour pattern of an object (...), described in terms of a limited set of*

*

events selected as its alphabet" [7]. The concept of process may have different names. For instance, in CCS process is called *agent*, whilst *LOTOS* prefers to call the process as *behavior expression* [4]. Because *CCL* tries to be close to the *Lisp* programming paradigm where every single piece of code should be a correct list expression [5], the *CCL* expressions which are executed by *CCL* processes are also defined in the form of lists. *Nlists* are formed from *actions*, other processes and operators. *Actions*, sometimes called *events* (*CSP*) [15], in *CCL* convention are called *primitives*. Similar to the notion of action in process algebras, primitives are atomic, i.e. *CCL* is not interested in a primitive's internal structure. On the other hand, since primitives are defined as *Clojure* functions ready to run when the model is executed, they have to be implemented so that they do not brake model execution. In particular, they should be consistent with *Java's* recommendation as regards interruption handling.

One of the basic operators used for constructing more complex structures from basic ones is the *prefixing operator*. In *CCL* the precedence of expressions is determined by their order in the named list, where the expressions on the left followed before those on the right. Other operators such as *choice operator* or *concurrent composition* also are available. CCL supports interruption mechanism by introducing an *interruption operator.* It has a very practical prototype, and comes from *Java* and is similar to the threads interruption handling in *Java*. On the other hand, it provides the ability to declare the interruption handler as it is possible to declare a shutdown hook in the *Java* virtual machine. Thanks to the interruption mechanism the user has an possibility not only to raise an exception, but also correctly handle it.

2 CCL notation

CCL notation consists of several operators and constructions. The expressions presented below are sufficient to construct a *CCL* model sufficient for many interesting and complex problems (a few of them are presented below). The first six constructions refer to sequential processing, whilst the last four (such as *concurrent composition* or *synchronization queues*) support the modeling of concurrency in the system. Due to some similarities with popular process algebras, the semantics of these constructions should not be difficult to understand.

2.1 Named List (nlist)

A named list *(nlist)* is one of the fundamental constructions of *CCL*. Nlist is a sequence of instructions processed within a single thread, where every instruction can be: a primitive call, a condition check, a variable change, an exception rising or communication via synchronization queues. Nlist can be declared using macro *def-nlist* (Listing: 1, line: 1).

```
1 (def-nlist list-name list-body)
```

Listing 1: Nlist definition scheme

where *list-name* is the name of the list and *list-body* is a sequence of instructions following it. For instance, a simple nlist *TickTacker* printing infinitely on the console *"ticks"* and *"tacks"* may look as follows (Listing: 2, line:1):

```
1 (def-nlist TickTacker ((tick) (tack) TickTacker))
```

Listing 2: TickTacker nlist

A *TickTacker* body contains a list of two consecutive primitives, *tick* and *tack*, and ends up with a *TickTacker* list re-call. So, from the execution point of view, this is an infinite sequence of ticks and tacks. Tick and tack have to be defined and registered as *CCL* primitives. Therefore, first they have to be defined as Clojure functions (Listing: 3, lines: 1 and 2), then declared as primitives (Listing: 3, line: 3).

```
1 (defn tick [] (println "tick"))
2 (defn tack [] (println "tack"))
3 (reg-as-prim tick tack)
```

Listing 3: Tick and Tack as Clojure functions

2.2 Primitives

The *tick* and *tack* primitives are defined as *Clojure* functions, which take no parameters on their input and do only one *Clojure* action *println* (printing the given text on to the console).

```
1 (defn tt [prim-name]
2   (do
3     (println prim-name)
4     (wait 500)))
5 (reg-as-prim tt)
6
7 (def-nlist TickTacker ((tt "tick") (tt "tack") TickTacker))
```

Listing 4: Simple parametrized primitive

Since primitives are just *Clojure* functions, they can also take parameters on their input. Thus, instead of defining two different primitives, *tick* and *tack,* only one parameterized primitive can be defined (Listing: 4, lines: 1 - 4). Now, according to the *Clojure* syntax, the primitive *tt* first prints the given text, then waits 500 milliseconds. The *TickTacker's* definitions also evolve and, after changes, *TickTacker* looks as on (Listing: 4, line: 7).

2.3 Nlist local variables

Nlist's local variables are another syntax mechanism that supports nlist processing. A local variable is declared at the time of the first assignment. Its name must start with colon (similar to *Clojure's* keywords). Local variables serve as a data interchange mechanism between different primitives and synchronization queues. They are also used in the logical conditions of the conditional choice operator. In the *TickTacker* example, the input data can be passed to the *tt* primitive as variables, not as literals. A definition of *TickTacker* using variables is as follows (Listing 5):

```
1 (def-nlist TickTacker ((:val "tick") (tt :val)
2                        (:val "tack") (tt :val)
3                        TickTacker))
```

Listing 5: Nlist local variables

After this modification, *TickTacker* consists of five expressions where *(:val "tick")* assigns the value *"tick"* to *:val* , then in the next expression the value of *:val* is passed into primitive *tt*.

2.4 Nlist parameters

The nlist parameters are variables, which can be passed into the given nlist passage from outside. They are declared together with the name of the nlist, then in the nlist's body they can be used in the same way as any other variables. In the case of the *TickTacker* example, an nlist parameter mechanism can be used to implement a tick-tack counter (Listing 6). Of course, incrementation of the variable *:val* is in the *Lisp* manner i.e. operator + precedes all of its arguments.

```
(def-nlist (TickTacker :cnt) (
               (:val "tick") (tt :val)
               (:val "tack") (tt :val)
               (tt :cnt) (TickTacker (+ :cnt 1))))
```

Listing 6: Nlist parameters

As a result, after every tick-tack passage a subsequent number is written. Because of the counter, every tick-tack run has its own number assigned. Thus, the next extension means that the odd runs will print *tick* before *tack*, whilst even runs behave inversely; they will print *tack* before *tick*. To do this, a *conditional choice* operator will be used.

2.5 Conditional Choice

Conditional choice allows the operator to select which *nlist* body is to be executed next. Its syntax is similar to *cond Clojure/Lisp's* macro, although the axiom *cond* is replaced by a question mark, and the condition results are just nlist bodies. Another difference lies in the fact that if a conditional clause is a part of another nlist, no part of this *nlist* is evaluated, apart from the chosen nlist body. The *conditional choice* scheme is as follows (Listing: 7).

```
(?
  (condition-1)
    (nlist-body-as-a-condition-result-1)
  (condition-2)
    (nlist-body-as-a-condition-result-2))
```

Listing 7: Conditional choice - scheme

Since the *TickTacker* example has been equipped with a counter *:cnt*, odd and even ticks can be distinguished. Let us use a conditional choice to change a *TickTacker* in a way that for odd counter values it will print *"tack"*, whilst for even it will print *"tick"*[1]. In the following example, the standard *Clojure* predicates such as odd? and even? are used to build condition clauses.

```
(def-nlist (TickTacker :cnt)
  (?
    (odd?  :cnt)
      ((tt "tack")
      (tt :cnt) (TickTacker (+ :cnt 1)))
    (even?  :cnt)
      ((tt "tick")
      (tt :cnt) (TickTacker (+ :cnt 1)))))
```

Listing 8: Conditional choice - example

2.6 Random Choice

A *random choice* allows us to specify an nlist clause containing sub-clauses processed with some likelihood proportional to the assigned weight. The operator's syntax is similar to the *conditional choice* operator, except for the fact that the condition clauses are replaced by weights determining sub-clause execution probability. For instance, a *TickTacker*, where *"ticks"* occur four times more frequently (approximately) than *"tacks"* can be specified as follows (Listing 9):

[1] Of course, the same result can be easily achieved without using a conditional choice.

```
1 (def-nlist TickTacker
2   (??
3     1 ((tt "tack") TickTacker)
4     4 ((tt "tick") TickTacker)))
```

Listing 9: Random choice

2.7 Sequential Composition

A *sequential composition* operator allows the specification of any number of nlists, which are processed sequentially. The operator is important mainly because of its compositional meaning. It allows the user to extract some sequences of actions into the separately named *nlist* definitions.

```
1 (def-nlist Ticker ((tt "tick")))
2 (def-nlist Tacker ((tt "tack")))
3 (def-nlist TickTacker ((-- Ticker Tacker) TickTacker))
```

Listing 10: Sequential composition

The *sequential composition* operator symbol -- follows immediately after the opening bracket. Since the sequential composition close is directly followed by the *TickTacker* self-recurrent declaration, it is advised to not put their self-recurrent declarations into both *Ticker* and *Tacker*. (Listing 10). Otherwise, the expressions followed the close may become unreachable.

2.8 Concurrent Composition

A *concurrent composition* operator allows the specification of any number of nlists, which are processed concurrently. Every nlist within the *concurrent composition* operator is processed independently in a separate execution thread. An inspiration for the *concurrent composition* operator was the composition operator in *CCS* [4].

```
1 (def-nlist Ticker ((tt "tick") Ticker))
2 (def-nlist Tacker ((tt "tack") Tacker))
3 (def-nlist TickTacker (| Ticker :tickProc Tacker :tackProc))
```

Listing 11: Concurrent composition

Of course, following the *Lisp* style the *concurrent composition* operator symbol follows immediately after the opening bracket and is one for any number of operands. With the help of the *concurrent composition* operator, our study case *TickTacker* can be modified so that it consists of two separate nlists: *Ticker* and *Tack*er. Each of these nlists is processed independently within the two separate processes *tickProc* and *tackProc*. The appropriate code is as follows (Listing 11).

2.9 Synchronization Queues

One of the fundamental structures mediating the communication between different processes is a buffer. Because of its importance and relative simplicity in design, it is one of the most commonly modeled artifacts [3]. Very frequently in practice a buffer communication model is implemented in the form of synchronization queues *(SQ)*. In some languages, like Java, such a mechanism has become a part of the language standard[2].

CCL adopts *SQ* as a basic synchronization mechanism. A synchronization queue is used to synchronize two or more nlists where some of them write to the queue, while others read from the queue. All the *SQs* implement a common queue interface (Table 1) allowing elements to be inserted into and

[2]http://download.oracle.com/javase/1.5.0/docs/api/java/util/AbstractQueue.html

Table 1: A Common SQ interface - method meaning

SQ methods	description
`(q-get q-name)`	returns and removes the first element from the queue
`(q-peek q-name)`	returns the first element from the queue
`(q-put q-name)`	inserts the element at the end of the queue
`(q-try-put q-name)`	as q-put but if the queue is full, false is returned
`(q-size q-name)`	returns the number of elements in the queue
`(q-capacity q-name)`	returns the maximal size of the queue

Table 2: Synchronization queues

SQ Type	SQ capacity	Read policy	Write policy	Operations ineffective
1	0	Non-block.	Non-block.	any
2	0	Blocking	Non-block.	*peek, try-put*
3	0	Non-block.	Blocking	*peek, try-put*
4	0	Blocking	Blocking	*peek, try-put*
5	$n > 0$	Non-block.	Non-block.	*peek, try-put*
6	$n > 0$	Blocking	Non-block.	*try-put*
7	$n > 0$	Non-block.	Blocking	none
8	$n > 0$	Blocking	Blocking	none

removed from the queue. There are two additional methods for reading data: *q-get* and *q-peek* and two methods for writing: *q-put* and *q-try-put*. There are also two additional methods which allow the number of elements in the queue and the capacity of the queue to be checked - *q-size* and *q-capacity* respectively.

Depending on the initial queue settings (i.e. size and blocking policy), semantics of the read/write operations may vary slightly. Due to the queue capacity and adopted blocking policy, eight types of *SQ* have been defined (Table: 2).

Every queue may be blocked at its (both) ends i.e.: if the queue is empty, each reader must wait for an element to be written and, correspondingly, if a queue is full, every writer must wait to get an element from the queue. If the queue is neither full nor empty, and its capacity is greater than zero, each of these conditionally blocking methods behaves in a non-blocking manner. In the case of a 0-length queue (which is always both: *full* and *empty* at the same time*)*, every method designed as conditionally blocking is always locked up when called, and such a call must wait for synchronization with another running thread.

There are two main groups of *SQ* types: *SQs* with zero-capacity and *SQs* with non-zero capacity (Table: 2). The 0-length *SQs* do not have any internal capacity, not even a capacity of one. Thus, writing to the queue is effective only if it happens at the same time as reading. In other words, the element being inserted to the 0-capacity queue has to be immediately taken from it. To make *read* and *write* operations occur exactly at the same time, they should be mutually synchronized, so at least one of them has to be blocking[3]. For this reason all the operations for the SQ type 1 (Table: 2) are ineffective (i.e. their result is undefined). Because of the size of zero-capacity *SQs,* checking whether the queue is full or empty is pointless. Thus, for any 0-capacity queues the methods which first try to check the content of the queue (i.e. as *peek* or *try-put)* are pointless as they are ineffective.

Since n-capacity queues have some storage space, calling methods which check the size of the queue are possible. A queue which has all the methods effective is a non-zero capacity both side blocking *SQ* (type 8, Table: 2). On the other hand, a both side non-blocking non-zero capacity queue does not have *peek* and *try-put* methods available (type 5, Table: 2). For all the types of *SQ* with a

[3]It is assumed that the occurrence probability of two different actions (read and write) in two independently executed threads at the same momentum of time is 0.

non-blocking *write*, a successful-write strategy is implemented. In other words, if the queue is full, the very first item is removed and the new item is added at the end. This is due to the fact that in many practical applications the most recent data is the most important. With the help of synchronization queues, the *TickTacker* case study can be converted into the well-known *Consumer-Producer* problem [1], where *ticks* and *tacks* are provided by separate nlists called *TickerProducer* and *TackerProducer* (Listing: 12, lines: 3-6), whilst printing service is provided by the nlist: *TickTackererConsumer* (Listing: 12, lines: 8-9). *TickerProducer* and *TackerProducer* add appropriate messages to the *tick-TackerQ*, call *tick* or *tack* primitive correspondingly then start the whole sequence of actions once again. *TickTackerConsumer* gets the message from *tickTackerQ*, assigns it to the local variable *x*, then prints the message by calling the primitive *toe*.

In order to connect producers and consumers at the beginning of the refreshed *TickTacker* case study, a *tickTackerQ SQ* is defined (Listing: 12, line: 1). The queue has the length 1 and a blocking input and output[4].

TickTackerWithSQ prints series of *"ticks"* and *"tacks*" on the console as a result. The frequency of individual words on the output depends on the efficiency of their producers.

```
1  (def-queue tickTackerQ :size 1 :rb :wb)
2
3  (def-nlist TickerProducer
4       ((q-put tickTackerQ "tick") (tick) TickerProducer))
5  (def-nlist TackerProducer
6       ((q-put tickTackerQ "tack") (tack) TackerProducer))
7
8  (def-nlist TickTackerConsumer
9       ((:x (q-get tickTackerQ)) (toe :x) TickTackerConsumer))
10
11 (def-nlist TickTackerWithSQ (| TickTackerConsumer
12                                TickerProducer
13                                TackerProducer))
```

Listing 12: TickTacker with SQs - example

2.10 Processing Interruption

It is very often impossible to predict all the circumstances in which the actual system has to work. Sometimes there is a need to cancel the current action or postpone it for a while. Such unexpected situations are common, especially in the case of real time systems where changing external conditions (e.g. the sudden appearance of an obstacle) may force the whole system to change its plans. As a result, the system controller must interrupt the current processing and follow the rescue plan relevant to the cause of the interruption. Ways of coping with unforeseen events are present in most object-oriented and procedural languages. These are signal-sending and exception-handling mechanisms. The major difference between them is that the exception handling is closed within a single thread, whilst signal sending crosses the thread/process boundary.

[4]Keyword :rb means that the read is blocking, whilst :wb means that the write is blocking

```
1 (def-nlist TickerTackerWriter (
2     (! ((tt "recovery") TickerTackerWriter))
3     (tt "tick") (tt "tack") TickerTackerWriter))
4
5 (def-nlist Spoiler ((tt "spoil") (-> :nlid) Spoiler))
6 (def-nlist TickTacker
7     (| TickerTackerProducer :nlid Spoiler))
```

Listing 13: TickTacker with processing interruption

CCL allows one nlist to interrupt another nlist. In such a case the interrupted nlist stops its current operations and may run an interruption handler containing an alternative sequence of instructions. To set the nlist's interruption handler, at least one of its sub-lists has to start with an exclamation mark followed by the alternative list of instructions (Listing: 13, line: 2).
The nlist's name defines only the execution scheme, not the running instance of the given nlist. Since the interruption request operates on the nlist's instance[5], another identifier for the nlist's instance is required.

Such an identifier is defined as a keyword following the nlist's name within the concurrent composition operator e.g. (Listing 13, line: 7). The interruption raising operator is defined by a sub-list starting with an arrow -> followed by a process name e.g. (Listing: 13, line: 5).

On listing 13 a *TickTacker* study case is shown, consisting of three nlists: *TickerTackerWriter*, *Spoiler* and *TickTacker*. This time *TickTackerWriter* is responsible for writing *ticks* and *tacks* alternately to the console, although it is able to handle interruption and, in such a case, it prints the word *recovery* (Listing: 13, line: 2) then comes back to the standard instructions flow. The *spoiler* nlist contains only two primitives. The first primitive *tt* writes the word *spoil* to the console, whilst the second sends an interruption request to the *TickerTackerWriter* executed by the proces *nlid* (Listing: 13, line: 7). The last nlist joins both previous nlists together, combining them with the help of the *concurrent composition* operator. The final run of *TickTackerWithSQ* results in a series of *ticks* and *tacks* printed into the console alternated by *spoils* and *recoveries.* Since every nlist can be interrupted at any time, any primitive is also susceptible to being interrupted. Thus, when developing primitives, it should be remembered that every piece of code from which an *InterruptedException*[6] can be thrown has to be surrounded by an appropriate *try-catch* clause. For this reason, in the *TickTacker* case study (Listing 13), the primitive *tt* has to be rewritten in order to be ready for incoming interruption (Listing 7).

```
1 (defn tt [prim-name]
2   (do (println prim-name)
3     (try
4       (wait 500)
5     (catch InterruptedException ex
6       (println "primitive interrupted")))))
7
```

Listing 14: Interruption ready primitive

2.11 Externalization

The fundamental way of communication in *CCL* is with *synchronization queues*. They allow different processes defined within the same *CCL* model to communicate and synchronize with each other. The *CCL* model is also able to communicate with the external environment. This communication is carried out by *Clojure* functions registered as *primitives*. This is because *primitives* may get values from the

[5]Many instances of the same nlist can be executed at the same time but only one needs to be interrupted.
[6]Interruption of Java thread may result in raising the exception *InterruptedException*.

model and pass them to the actual hardware as well as read registers of a physical device and return them back to the model. From the *CCL* model perspective, primitives are a bit like a black box. Their internal implementation is not important. Their structure and correctness are not subject to the formal analysis of a model.

Although, at first glance, the perspective of many processes communicating through synchronization queues within a single model seems reasonable, sometimes, especially in the case of a large model, it is necessary to divide it into sub-models. This may be caused by compositional considerations i.e. some parts of the large model may form a consistent sub-system, but no less important are the limited possibilities of analysis. The large model is difficult to inspect manually by engineers, hard to simulate and, due to the state explosion trap, it might be very difficult to analyze by formal methods.

```
1 (def-queue tickTackerQ :size 1 :rb :ext)
2 (defn-queue-wrapper tick-tack-put tickTackerQ :put)
3 (defn-queue-wrapper tick-tack-get tickTackerQ :get)
4 (req-as-prim tick-tack-put tick-tack-get)
5
6 (def-nlist TickerProducer
7      ((tick-tack-put "tick") TickerProducer))
8 (def-nlist TackerProducer
9      ((tick-tack-put "tack") TackerProducer))
10
11 (def-nlist TickTackerConsumer
12      ((:x (tick-tack-get)) (tt :x) TickTackerConsumer))
13
14 (def-nlist TickTackerProducers (| TickerProducer
15                                  TackerProducer))
```

Listing 15: Externalizatoin example (TickTacker model)

In *CCL*, creating new sub-models out of existing ones is called externalization. It relies on identifying existing synchronization queues between the model code, which is intended to be part of a new sub-model and the rest of the code. When these queues are identified, they are marked as externalized and their calls on both ends are wrapped into primitive calls. Thus, the processes, which have so far communicated through these queues, start to communicate through newly defined primitives, exactly in the same way as they communicate with the external environment. In other words, the new sub-model becomes external to the its parental (sub) model. The externalization mechanism provides a way to create a hierarchy of models. Each sub-model can be a subject of individual verification both: formal and by simulation.

In order to see how *CCL* supports externalization, let us go back to the example of *TickTacker with SQs* (Listing 12). In order to externalize the *TickTackerConsumer* process into a separate sub-model, the *tickTackerQ* queue has to be marked as externalized (Listing 15, line: 1) and wrapped into primitives (Listing 15, lines: 2). To create the queue wrapper, the appropriate macro *defn-queue-wrapper* is called. The user needs to define the name of the future primitive, the name of the queue and the operating policy: *get*, *peek*, *put* or *try-put* (Listing 15, lines: 2). Then, the queue access methods calls need to be replaced by wrapper calls (Listing 15, lines: 7, 9, 12). Due to the clarity of code the additional expression *TickTackerProducer* can be added (Listing 15, lines: 14 - 15). From this point there are two sub-models, one grouping producers: *TickTackerProducer*, and one consisting of *TickTackerConsumer*. Both of them can be verified and analyzed independently, so that when one of them is the subject of analysis it is assumed the second works correctly and vice versa.

3 Summary

CCL is the *Java/Clojure* library facilitating executable modeling, which provides syntax modeled on formal notations, such as process algebras [3]. It offers several syntactic mechanisms, which allow the declarative construction of increasingly complex structures. These are primitives, named list *(nlists), choice operators, concurrent composition operator, synchronization queues,* and *interruption operators. CCL* is more semantically strict than non-formal modeling languages, and thus it is more susceptible to different kinds of strict model analysis. Besides formal verification, *CCL* models can be verified by simulation as well as executed together with other *Clojure* and *Java* components.

The report is not a complete description of the *CCL* library. For instance many topics connected with using *CCL* together with *Robust* and *CCL Sim* are not covered here. More comprehensive materials and usage examples can be found on the *CCL* project's website: *http://www.kulakowski.org/ccl*.

References

[1] M. Ben-Ari. *Principles of concurrent and distributed programming*. Addison-Wesley, 2006.

[2] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, May 1985.

[3] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, 2001.

[4] C. Fidge. A comparative introduction to CSP, CCS and LOTOS. Technical report, Software Verification Research Centre, University of Queensland, 1994.

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

[6] S. Halloway. *Programming Clojure*. Pragmatic Programmers, May 2009.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[8] ISO8807. Information processing systems — open systems interconnection – LOTOS — A formal description technique based on the temporal ordering of observational behaviour. International Standard ISO 8807, ISO8807, 1989.

[9] K. Kułakowski. Robust - Towards the Design of an Effective Control Library for Lego Mindstorms NXT. In *Proceedings of Conference on Software Engineering Techniques CEE-SET 2009*, Sep 2009.

[10] K. Kułakowski. cljRobust - Clojure Programming API for Lego Mindstorms NXT. In *Agent and Multi-Agent Systems: Technologies and Applications*, volume 6071 of *LNCS*, 2010.

[11] K. Kułakowski. CCL Sim, the simulation environment for concurrent systems. In *proceedings of Dependability and Complex Systems DepCoS*, 2012.

[12] K. Kułakowski and P. Matyasik. RobustHX - The Robust Middleware Library for Hexor Robots. In N. Ando, S. Balakirsky, T. Hemker, M. Reggiani, and O. von Stryk, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 6472 of *Lecture Notes in Computer Science*. Springer Verlag, 2010.

[13] K. Kułakowski and T. Szmuc. Modeling Robot Behavior with CCL. In I. Noda, N. Ando, D. Brugali, and J. J. Kuffner, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 7628 of *Lecture Notes in Computer Science*. Springer Verlag, 2012.

[14] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.

[15] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall (Pearson), 2005.