Computer Science • 15 (1) 2014 http://dx.doi.org/10.7494/csci.2014.15.1.61

Sławomir Maludziński
Grzegorz Dobrowolski

MODEL CHECKING PROCESSES SPECIFIED IN JOIN-CALCULUS ALGEBRA

Abstract *This article presents a model checking tool used to verify concurrent systems specified in join-calculus algebra. The temporal properties of systems under verification are expressed in CTL logic. Join-calculus algebra, with its operational semantics defined by a chemical abstract machine, serves as the basic method for the specification of concurrent systems and their synchronization mechanisms, allowing for the examination of more complex systems. The described model checker is a proof of concept for the utilization of new methodologies of formal system specification and verification in software engineering practice.*

Keywords join-calculus, model checking, formal methods, automatic software verification

1. Introduction

Recent changes in processor design have influenced software development methodologies. Increases in computational speed are gained by adding CPU cores rather than increasing processor frequency. To benefit from this, applications need to have several threads of execution; in turn, increased concurrency leads to algorithms whose execution is difficult to understand and verify. One of the approaches in dealing with this complexity is to analyze systems using formal methods.

This article describes a model checker which uses join-calculus algebra [10] to specify concurrent systems with their properties defined in CTL logic [4]. Join-calculus algebra uses the notions of asynchronous messages, processes, join patterns, and reaction rules to facilitate the formal specification of a concurrent system. Its operational semantics are defined by a chemical abstract machine (CHAM) [5], where terms are added and removed from a chemical solution according to reaction rules. Informally, processes are executed as a result of matching asynchronous messages to join patterns. When such a match occurs, a process is executed with formal parameters of a message pattern substituted with actual parameters of sent messages. Furthermore, processes may define new reaction rules which can be used for matching further messages (reflexive CHAM). The exact matching between messages and join patterns is not specified and, as a result, it is undetermined which reaction rule will be executed. Despite its apparent complexity, this property lets us express more advanced constructs and synchronization mechanisms between processes. After messages are matched, they can no longer be used to match other join patterns. The names of join patterns are recursively bound in the process which defines them and within reaction rules within that process.

The described model checker is a proof of concept for the usage of new methodologies of formal system specification and verification in software engineering practice. The proposed application of join-calculus algebra for system specification, together with its representation by a chemical abstract machine, is a 6tnovel approach. The devised specification language has been designed for ease of use. As new programming languages (Cω, Join Java) which use the notion of join patterns are developed, the presented methodology serves as an evaluation of how these languages can be formally verified. The choice of model checking for system verification was determined by its successful application in other domains and its recent developments. The possible extension of the tool through the notion of localizations (defined in distributed join-calculus [11]) would allow the specification and verification of distributed, mobile, or multi-agent systems.

The next section will present the state of the art (Sec. 2). Section 3 and its subsections describe the model checker in more detail. The specification language is described in section 3.1. Lexical and semantic analyses are outlined in section 3.2, followed by descriptions of the code generator (Sec. 3.3) and interpreter (Sec. 3.4). The Kripke structure construction algorithm, which is one of the most important parts of the model checker, is mentioned in section 3.5. The CTL verification algorithm

used in the tool is referenced in section 3.6. Section 4 demonstrates a simple mutual exclusion problem, which is specified and solved with the model checker. The last section summarizes the article and presents some possible future extensions.

2. State of the art

Model checking [2, 12] involves the creation of a finite Kripke structure and verifying whether or not a given property is satisfied by it. Assuming the structure is finite, a sequence of instructions is given upon completion of the verification algorithm. This sequence determines whether the property is satisfied (witness) or not satisfied (counterexample).Most commonly, the verified system is specified using formal methods such as Petri Nets, finite automata, or process algebra, while the property is specified by a temporal logic formula. This method is known as temporal logic model checking, and it has been used successfully in industry to verify communication protocols [14], electronic devices [6], programming languages [7], and multi-agent systems [16]. Advances in model checking methodologies allow the verification of systems of considerable sizes [15] which make the verification effective in practice. The authors of [9] developed a methodology similar to the one described in this paper, where a nominal calculus is represented by history dependent automata and is model checked using software tools.

3. Model checking join-calculus algebra

The described tool is implemented in the C programming language using the Flex and Bison tools [1] to generate a language scanner and parser. Its main modules (Fig. 1) are separated into two groups; the first does static processing for a system specification and generates bytecode, while the second loads bytecode, executes it, and creates a Kripke structure. The Kripke structure is checked for whether or not the CTL formula under verification is satisfied in the model. When a formula is satisfied, a sequence of instructions (witness) is generated, which leads to the appropriate state. Otherwise the model checker outputs sequences which lead to states where the formula is not satisfied (counterexample).

During the model checking phase, the model checker uses the modules outlined in Figure 2. The lowest is a module which allocates the necessary memory for the interpreter structures. It can also duplicate and compare memory regions where structures are allocated. In this way, the interpreter may execute different process instructions, and the model checker can build other Kripke structure worlds. The specification language uses first-class reaction rules; as a result, it is difficult to deallocate the activation frames of the processes. This task is accomplished by the memory manager, which uses a garbage collector to deallocate all inaccessible interpreter structures. The interpreter executes all running processes and, in particular, matches sent synchronous and asynchronous messages with join patterns. When a match occurs, it executes a new process and may stop the current one. In the case of the model checker,

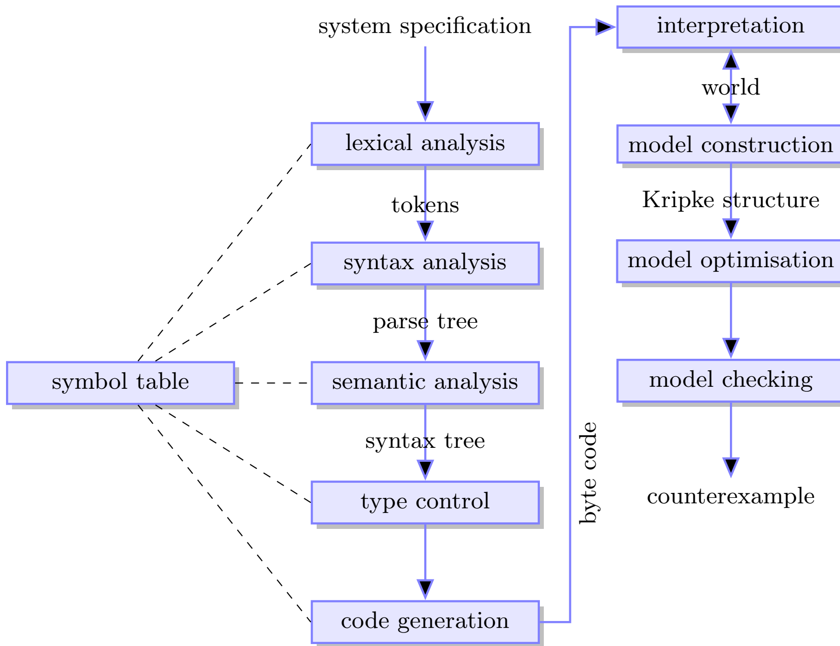

Figure 1. Model checker phases.

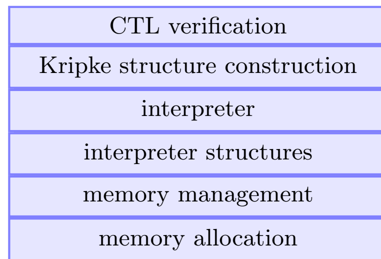

Figure 2. Model checker modules.

all of the possible process executions and pattern matches are taken into account, and the model checker module creates a process by executing the Kripke structure build algorithm. The CTL formula under verification is normalized and verified by the CTL verification module. The following subsection gives more detail about the model checker modules.

3.1. Specification language

The described model checker uses formal grammar to specify concurrent systems and their properties. The specification language is statically typed with first-class reaction

rules. Some improvements were made to facilitate the system specification and its CTL properties – the possibility to express CTL formula under verification and its fairness constraints. The language uses the following keywords: **async, boolean, ctlspec, else, fail, false, fairness, if, print, return, to, typedef, true, var, vector, void**. Some keywords are reserved for distributed join-calculus **location, go, halt, fail** and for the specification of CTL formula constraints **ctlspec, fairness**.

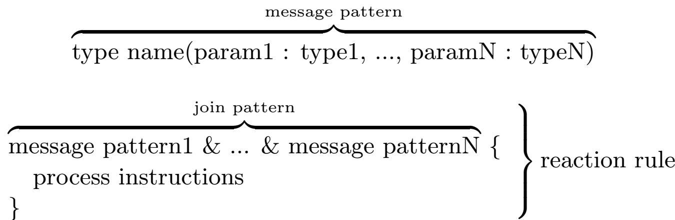

Figure 3. Message pattern, join pattern, process instructions and reaction rule.

A system specification consists of three main parts: types definition, system specification, and the CTL properties which need to be verified.

The types definition lets us declare types used within the system specification. Types include interval, enumeration, message pattern, and vector. The introduction of types allows us to verify whether or not a language can be executed during syntactic analysis and whether there will be no runtime errors caused by a types mismatch. The interval and enumeration types allow the smallest possible representation of numeric values and, thus, limit memory usage. The vector type lets us return multiple values from a process. The message pattern type allows us to create variables and assign message patterns of such type.

The system specification contains reaction rules and statements. Figure 3 presents an example reaction rule. Each reaction rule is comprised of a join pattern and corresponding process instructions. A process is executed when a match occurs between sent messages and the messages patterns of a join pattern. Each process may define nested reaction rules. Join pattern names are recursively bound inside each join pattern at the same nesting level. Statements include assignment, conditional instruction, return, message send, and parallel execution. The values of variables are calculated with arithmetic expressions. It is also possible to declare instruction blocks.

The system's CTL properties are declared as the last part of its specification. They are expressed using arithmetic and CTL **AG**, **AF**, **AU**, **AX**, **EG**, **EF**, **EU**, **EX** operators.

3.1.1. Example

Figures 4 and 5 demonstrate the specification language's most significant features. Figure 4 specifies three reaction rules as well as synchronous and asynchronous messages. The sending of the synchronous message stops process execution until the

```
1  spec main
2  {
3     void syncMsg() & async asyncMsg() { // join pattern and reaction rule
4     }
5     void syncMsg() {
6     }
7     async asyncMsg() {
8     }
9
10    asyncMsg(); syncMsg();             // a/synchronous message
11 }
```

Figure 4. Specification language – example.

process which was invoked finishes. After messages are sent, it is possible to execute processes which pertain to each of the reaction rules. This non-determinism in choosing reaction rules lets one define more advanced synchronization mechanisms and is one of the strengths of join-calculus process algebra.

Figure 5 presents other instructions from the specification language. Line 1 defines an integer type which is then used to declare variables (10, 20). Lines 3 – 6 define types of messages which can be returned from reaction rules. Variables of those types are declared in lines 11 and 12. It is also possible to return a vector of values (defined in 6) which is used in line 30. Reaction rules are defined in lines 14, 18, and 19. A concurrent statement which is used in line 33 allows us to execute two processes simultaneously. The specification also demonstrates how to pass message patterns between processes. In this example, `addmul` is passed as a parameter of `accept` and used to compute some value. Furthermore, lines 15, 23, 25, and 27 show how to return values from processes.

3.2. Lexical and semantic analysis

In the first step, the model checker tokenizes and parses the system specification. The scanner removes comments and distinguishes tokens such as keywords, constants, and identifiers. The parser checks if the specification is a proper sentence in the language grammar and accordingly builds a parse tree. The parser accounts for operator priority and associativity. During parsing, all found symbols are placed in a symbol table. The syntax tree is checked for any semantic errors (e.g. unmatched types) during semantic analysis. Variable scope is also analyzed for proper value assignment. When the specification does not yield any syntactic or semantic errors, a bytecode is generated.

3.3. Code generation

The generated bytecode uses instructions which are interpreted by a stack machine. There are eight groups of instructions: stack operation (push, pop, dup), jump (conditional, unconditional), message send (synchronous, asynchronous), return, store,

```
1  typedef int { 0 .. 1023 }        // types declaration
2
3  typedef int call_t(int);
4  typedef int accept_f(int);
5  typedef void accept_t(accept_f);
6  vector rv_v { call_t, accept_t }
7
8  spec rendezvous
9  {
10    var value : int,               // variable declaration
11        call1 : call_t,
12        acc1 : accept_t;
13
14    int addmul(val : int) {        // join pattern
15       return val * val + val;
16    }
17
18    rv_v newRendezVous() {
19       int call(vi : int) & void accept(func : accept_f) {
20          var r : int;
21
22          r = func(vi);
23          return r to call;
24          |
25          return to accept;
26       }
27       return call, accept;
28    }
29
30    call1,acc1 = newRendezVous();
31
32    value = call1(12);             // message send
33    |                              // concurrent statement
34    acc1(addmul);
35  }
36
37  ctlspec AG (value == 156);
```

Figure 5. Specification language – *rendezvous.*

block (begin, end), concurrency (beg, end), and primitives. As the tool uses a stack machine, all expressions are transformed into reverse Polish notation.

3.4. Interpreter

The interpreter uses six basic structures, depicted in Figure 6 and Figure 7. Each process (**PROC.rrule**) is associated with a reaction rule (**RRULE**) which contains the names of messages needed to execute it (**RRULE.msgN**), literals in bytecode (**RRULE.literalN**), pointers to reaction rules nested in the reaction rule itself (**RRULE.rruleN**), and bytecode. Processes execute the instructions (**PROC.IP** of reaction rules. Process data is kept in messages used to execute it (**FRAME.msgN**), in activation frames (**FRAME.varN**), and in blocks (**BLOCK.vars**). Local data is kept on a stack (**PROC.stack**) whose top is pointed to by **PROC.SP**. The current frame and block are pointed to by the **PROC.AF** and **PROC.HC** links. For optimization reasons, the **PROC.AF** link always points to the current acti-

vation frame or block, while **PROC.HC** points to the process activation frame. **FRAME.parent** is a data access link to the activation frame from which each given process was executed. A data access link lets us access variables which are nested in upper levels. When a code block is executed, **PROC.IP** and **PROC.SP** are stored in **BLOCK.IP** and **BLOCK.SP** respectively. A block of code holds a local stack (**BLOCK.stack**) which is used to perform calculations. Sent messages (**MESSAGE**) are delivered to activation frames (**FRAME.msg#**). Each message contains its name (**MESSAGE.name**) and the values of its parameters (**MESSAGE.params**). They also hold control links (**MESSAGE.proc**) to processes which have sent synchronous messages. A message control link allows the return of execution to the original process which sent the synchronous message. When processes are created using parallel instructions, their **PROC.parent proc** control link points to their parent process and allows the resumption of execution when both end. The running state of a process is kept in the **PROC.running** variable. The ability to store and return message patterns from reaction rules is implemented using closures (**RRPOINTER**), which hold the name of the message and a pointer to the activation frame.

PROC	FRAME	MESSAGE	BLOCK	RRULE	RRPOINTER
parent proc	parent	name	parent	#msgs	name
AF	rrule	proc	IP	#literals	frame
HC	#msgs#	params	SP	#rrules	
rrule	msg1		var1	#vars	
IP	msg2		...	msg1	
SP	...		varN	...	
running	msgN		stack	msgN	
stack	var1			literal1	
	...			...	
	varN			literalN	
				rrule1	
				...	
				rruleN	
				bytecode	

Figure 6. Interpreter structures.

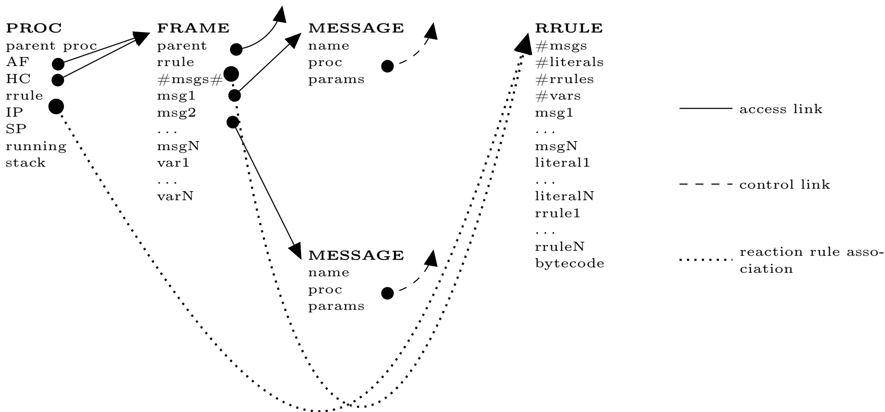

Figure 7. Running process with access and control links.

Interpreter structures allow us to execute processes. Messages are delivered and matched to reaction rules in order to create new processes. In the case of synchronous messages and parallel instructions, the parent process is stopped and the child process starts execution. For the model checker, how the process scheduler works is not relevant. The model checker algorithm finds all possible message matches and process instructions.

The execution of processes and creation of interpreter structures leads to a spaghetti stack. Such structures are difficult to deallocate after a process stops execution. Thus, a garbage collector is used to remove inaccessible structures.

3.5. Kripke structure construction algorithm

The general Kripke structure construction algorithm described in [3] needs to be modified for the special case when join- calculus algebra is used. The majority remains the same, with more detailed steps for constructing states that result from join pattern matching. The algorithm is depicted in Figure 8. Initial state S_0 represents the state of execution of the main process. Subsequent states are constructed by the algorithm's main loop (line 3). The algorithm uses two sets of states: W and W_n. Set W contains all states which are taken into account in its current iteration. States which are to be taken into account in the following iteration are added to set W_n. The algorithm finishes its execution when no new transitions have been added.

```
1  W = {S_0}
2  S = {}, W_n = {}
3  while (transition added) {
4     while (W ≠ ∅) {
5        choose s ∈ W;
6        W = W \ {s};
7
8        create new states {s_n} perm msgs rrules, label rrule;
9        if (s_n ∈ S) make transition R(s,s_n);
10       else { make transition (s,s_n); W_n = W_n ∪ {s_n}; S = S ∪ {s_n} }
11
12       ∀p_i ∈ s create new state {s_n}; exec statement; label process
13       if (s_n ∈ S) make transition R(s,s_n);
14       else { make transition R(s,s_n); W_n = W_n ∪ {s_n}; S = S ∪ {s_n} }
15    }
16    W = W_n
17 }
```

Figure 8. Kripke structure creation algorithm.

The first step (line 5) is to choose a state from set W for which no transitions have been made. For such a state, new states are created which result from the execution of the current instruction of all running processes (line 12). If such a state already exists, then the transition is made to it (line 13); otherwise, a new state is created (line 14) and added to set W_n. Transitions are labeled with the identifier of the process which executed the statement. After iterating over all states in W, the set of newly created states W_n is taken into account (line 16).

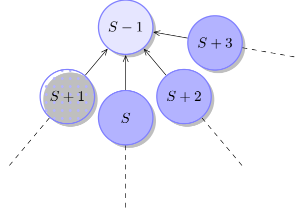

Figure 9. Kripke structure creation algorithm

New states are created by executing process statements (which modify the process instruction pointer). Assignments modify the values of variables. Conditional jumps change the process instruction pointer. Concurrent instructions create two processes which execute simultaneously while their parent process is stopped. Execution of the parent process resumes when both of the sub-processes finish their execution and, as a result, the subsequent iteration of the Kripke construction algorithm needs to take both of them into account. What is more interesting is the result of sending synchronous and asynchronous messages (lines 8–10). Transmission of the synchronous message stops the current process; afterwards, it is possible that one of reaction rules will be executed. A similar case holds for the asynchronous message; however, the process which sends it is *not* stopped. The Kripke construction algorithm needs to create states which result from all possible permutations of matches between reaction rules and sent messages (both synchronous or asynchronous). As some other reaction rules may be invoked by matching messages to the join pattern, the algorithm also needs to construct a state when no messages have been matched (state $S + 1$ on Figure 9). The CTL labeling algorithm which is used in the tool operates on the predecessors of a Kripke world. For this reason, all Kripke structure worlds store their predecessors.

3.6. CTL verification

The proposed tool uses a CTL-labeling algorithm [8]. The algorithm labels all Kripke structure states with subformulas of a formula being verified. As any CTL formula can be expressed using only five operators: $\neg$, $\vee$, **EG**, **EX**, **EU**, the algorithm labels only states which satisfy these operators. Thus, a formula is normalized before verification; i.e., expressed using only these operators. If the initial state of the Kripke structure is not labeled with a label from the formula under verification, then the whole formula is unsatisfied and a counterexample is generated. Otherwise, a witness is generated.

4. Example: mutual exclusion

Figures 10 and 11 demonstrate the academic problem of a race condition between two running processes. As a result of instruction interleaving in the main process, the `total` variable is set to an improper value. The specification in Figure 10 defines two

processes (line 13) which are executed concurrently. A property of the system under verification is defined by the CTL formula **AF** (*total* == 2500), and one of the counterexamples generated by the utility is presented in Figure 11. The counterexample (read bottom-up) shows the sequence of instructions which lead to the state where the value of the variable `total` equals 3500, which is improper.

```
1  typedef int { 0 .. 4095 }
2
3  spec account {
4      var total : int,
5          sub1  : int,
6          sub2  : int;
7
8     total = 2000;
9
10    { sub1 = total;
11      sub1 = sub1 - 1000;
12      total = sub1;
13    } | {
14      sub2 = total;
15      sub2 = sub2 + 1500;
16      total = sub2;
17    }
18 }
19 ctlspec AF (account.total == 2500);
```

Figure 10. Race condition.

```
1  //
2  // AF (total == 2500)
3  //
4  counterexample (total == 3500)
5  proc: 0 ip: 19   <proc end 0>
6  proc: 2 ip: 17   <proc end 2>
7  proc: 2 ip: 16   total = sub2;
8  proc: 2 ip: 15   sub2=sub2+1500;
9  proc: 1 ip: 13   <proc end 1>
10 proc: 1 ip: 12   total = sub1;
11 proc: 2 ip: 14   sub2 = total;
12 proc: 1 ip: 11   sub1=sub1-1000;
13 proc: 1 ip: 10   sub1 = total;
14 proc: 0 ip: 13   |
15 proc: 0 ip: 8    total = 2000;
16 proc: 0 ip: 0    <proc beg 0>
```

Figure 11. Race condition. Counterexample **AF** (*total* == 2500).

```
1  typedef int { 0 .. 4095 }
2
3  spec account
4  {
5     var total : int;
6
7     void wait() & async free() {
8                       return; }
9     void signal() { free();
10                    return; }
11
12    void credit(amount : int) {
13       var sub1 : int;
14
15       wait();
16           sub1 = total;
17           sub1 = sub1 + amount;
18           total = sub1;
19       signal();
20
21       return;
22    }
23    void debit(amount : int) {
24       var sub2 : int;
25
26       wait();
27           sub2 = total;
28           sub2 = sub2 - amount;
29           total = sub2;
30       signal();
31
32       return;
33    }
34
35    free();
36
37    total = 2000;
38
39    credit(1500);
40    |
41    debit(1000);
42 }
43
44 ctlspec AF (account.total == 2500);
```

Figure 12. Mutual exclusion problem. Semaphore and critical section.

The problem shown above is solved by introducing a semaphore [13]. Its specification is shown in Figure 12 (lines 7–10). The semaphore is defined using join patterns (line 7) and its state is set to `free` in line 35. This allows access to the critical section for any one of the processes. A process which sends a synchronous `wait` message is granted access to the critical section. As the asynchronous message `free` is removed from the chemical abstract machine, no other process can continue its execution after sending a subsequent synchronous `wait` message. After the process which is inside the critical section sends the synchronous message `signal`, the message `free` will be re-added to the abstract state machine. Thus, any other process which previously executed the `wait` method can enter the critical section.

The specification was verified by the described tool and no counterexample could be found. The tool printed only witnesses which lead to a proper system state.

5. Summary and future extensions

The proposed model checker can be used for the specification and verification of concurrent systems. The tool uses join-calculus process algebra to define and reason about complex concurrent systems and is an interesting example of how concurrency can be modeled thanks to the usage of the CHAM, through chemistry abstraction. Simple notions such as asynchronous messages, join patterns, and reaction rules serve as the basis for their specification. The way in which join-calculus algebra is used to verify concurrent systems differs from specifications which use finite state machines, Petri-nets, or temporal logics. The developed methodology offers a high-level abstraction, where reaction rules are first class and can be passed between processes. This possibility offers sophisticated methods of system specification. In this paper, we presented two synchronization mechanisms, semaphore and *rendezvous*. It is possible to define other mechanisms, such as buffers, barriers, and synchronous and asynchronous channels, as well as communication protocols. The presented model checker is still under development, while its major modules have already been implemented – scanner, parser, semantic and types validator, code generator, interpreter, Kripkestructure construction algorithm, and CTL formula verification algorithm. There are still some tasks to be completed (implementation of labels, code loader). Since the tool is written in the C programming language, it is possible to extend it with Kripke structure optimizations such as cone of influence, on-the-fly verification, and others. It is expected that the model checker can be extended by the notion of localizations which are defined by distributed join-calculus [11]. Such an extension would create the possibility of verifying mobile or multi-agent systems, where mobility and agents would be specified using the notion of locations. The article contributes to the field of formal methods in finding best formalisms, methodologies, and tools, which bridge the gap between theoretical results and the everyday practice of software engineering.

References

[1] Aaby A. A.: *Compiler Construction using Flex and Bison.* Walla Walla College, 2003.

[2] Baier C., Katoen J. P.: *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008. ISBN 026202649X, 9780262026499.

[3] Ben-Ari M.: *Mathematical Logic for Computer Science.* Prentice-Hall International Series in Computer Science. Springer, 2001. ISBN 9781852333195, URL `http://books.google.pl/books?id=hzWlEy1qqR8C`.

[4] Ben-Ari M., Manna Z., Pnueli A.: *The Temporal Logic of Branching Time.* In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pp. 164–176. ACM, New York, NY, USA, 1981. ISBN 0-89791-029-X. URL `http://dx.doi.org/10.1145/567532.567551`.

[5] Berry G., Boudol G.: The Chemical Abstract Machine. In: *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 81–94. ACM, New York, NY, USA, 1990. ISBN 0-89791-343-4. URL `http://dx.doi.org/http://doi.acm.org/10.1145/96709.96717`.

[6] Cimatti A., Clarke E., Giunchiglia E., Giunchiglia F., Pistore M., Roveri M., Sebastiani R., Tacchella A.: NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In: *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, *LNCS*, vol. 2404. Springer, Copenhagen, Denmark, 2002.

[7] Dennis L. A., Fisher M., Webster M. P., Bordini R. H.: Model Checking Agent Programming Languages. *Automated Software Engg.*, 19(1): 5–63, 2012. ISSN 0928-8910. URL `http://dx.doi.org/10.1007/s10515-011-0088-x`.

[8] Edmund M., Clarke J., Grumberg O., Peled D. A.: *Model Checking.* MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-03270-8.

[9] Ferrari G. L., Montanari U., Tuosto E.: Model Checking for Nominal Calculi. In: *Proceedings of the 8th International Conference on Foundations of Software Science and Computation Structures*, FOSSACS'05, pp. 1–24. Springer-Verlag, Berlin, Heidelberg, 2005. ISBN 3-540-25388-2, 978-3-540-25388-4. URL `http://dx.doi.org/10.1007/978-3-540-31982-5_1`.

[10] Fournet C., Gonthier G.: The Reflexive CHAM and the Join-calculus. In: *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 372–385. ACM, New York, NY, USA, 1996. ISBN 0-89791-769-3. URL `http://dx.doi.org/http://doi.acm.org/10.1145/237721.237805`.

[11] Fournet C., Gonthier G., Lévy J. J., Maranget L., Rémy D.: *A Calculus of Mobile Agents.* In: *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, pp. 406–421. Springer-Verlag, 1996. URL `citeseer.ist.psu.edu/fournet96calculus.html`.

[12] Grumberg O., Veith H., eds.: *25 Years of Model Checking – History, Achievements, Perspectives*, *Lecture Notes in Computer Science*, vol. 5000.

Springer, 2008. ISBN 978-3-540-69849-4.

[13] Herlihy M., Shavit N.: *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.

[14] Holzmann G.: *SPIN Model Checker, Primer and Reference Manual.* 1 st. ed. Addison-Wesley Professional, 2003. ISBN 0-321-22862-6.

[15] Burch J. R., Clarke E. M., McMillan K. L., Dill D. L., Hwang L. J.: Symbolic Model Checking: 10^{20} States and Beyond. In: *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pp. 1–33. IEEE Computer Society Press, Washington, D.C., 1990.
URL `citeseer.ist.psu.edu/burch90symbolic.html`.

[16] Niewiadomski A., Penczek W., Szreter M.: A Model Checker for Real Time and Multi-agent Systems. In: *Proceedings of VerICS 2004*, pp. 88–99. Humboldt University, 2004.

Affiliations

Sławomir Maludziński
AGH University of Science and Technology, Krakow, Poland,
`slawomir.maludzinski@gmail.com`

Grzegorz Dobrowolski
AGH University of Science and Technology, Krakow, Poland,
`grzela@agh.edu.pl`

Received: 22.06.2013
Revised: 27.08.2013
Accepted: 20.12.2013