CAAL: Concurrency Workbench, Aalborg Edition

Jesper R. Andersen, Nicklas Andersen, Søren Enevoldsen, Mathias M. Hansen, Kim G. Larsen, Simon R. Olesen, Jiří Srba, and Jacob K. Wortmann

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, 9220 Aalborg Øst, Denmark

Abstract. We present the first official release of Caal, a web-based tool for modelling and verification of concurrent processes. The tool is primarily designed for educational purposes and it supports the classical process algebra CCS together with its timed extension TCCS. It allows to compare processes with respect to a range of strong/weak and timed/untimed equivalences and preorders (bisimulation, simulation and traces) and supports model checking of CCS/TCCS processes against recursively defined formulae of Hennessy-Milner logic. The tool offers a graphical visualizer for displaying labelled transition systems, including their minimization up to strong/weak bisimulation, and process behaviour can be examined by playing (bi)simulation and model checking games or via the generation of distinguishing formulae for non-equivalent processes. We describe the modelling and analysis features of Caal, discuss the underlying verification algorithms and show a typical example of a use in the classroom environment.

1 Introduction

Concurrency is a classical topic taught at many universities as a bachelor or master degree course in Computer Science. For an introductory course in concurrency, the typical content includes the use of a simple language for the description of parallel processes (e.g. CCS, CSP, ACP or Petri nets) that is used for modelling concurrent systems and for explaining the key concepts of equivalence checking and model checking. At Aalborg University, we offer such an introductory course called *Semantics and Verification* to the 6th semester software engineering and computer science students. The course is based on our Reactive Systems book [1] that, among others, introduces the CCS process algebra (Calculus of Communicating Systems [14]) and bisimulation/model checking approach, including the corresponding game characterization. In order to motivate the students to study and appreciate the theoretical concepts in concurrency, we engage them in a few medium-size modelling exercises. This hands-on modelling experience makes them realize that designing even small concurrent systems is difficult and that support by an adequate tool can be very useful. For this purpose, we introduce the open-source tool Caal (standing for Concurrency workbench developed at AALborg university) that supports CCS and TCCS

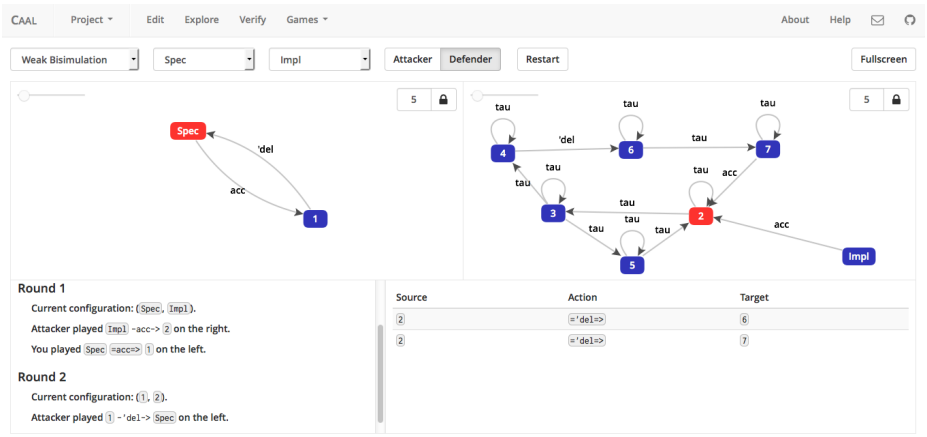


Fig. 1: Game module in CAAL

as the input language. CAAL is programmed in TypeScript, a typed superset of JavaScript that compiles into plain JavaScript. The input language of CAAL is an extension of the well-known Concurrency Workbench (CWB) [5] input syntax, so existing CWB projects can be opened in CAAL. The tool is hosted at

`http://caal.cs.aau.dk`

and it runs in any modern browser but a stand-alone installation is possible too.

CAAL offers an editor with online syntax correction, an explorer for the visualization of the generated labelled transition systems, including different minimizations w.r.t. to strong and weak bisimulation as well as the display of strong/weak and timed/untimed transitions. The explorer module enables an interactive exploration of the state-space via a predefined depth of the view horizon (suitable for exploring large state-spaces), automatic layout with the possibility to lock and rearrange the position of nodes, zoom functionality, simplification w.r.t. structural congruence and export as a raster graphics image. The verification module of CAAL allows to formulate equivalence and model checking queries and verify them either individually or collectively. It is possible to generate distinguishing formulae for non-equivalent processes or enter the game module (see Figure 1) and interactively play (bi)simulation and model checking games.

Related work. Concurrency WorkBench (CWB) [5] and its continuation Concurrency WorkBench of the New Century (CWB-NC) [6] are perhaps the best known tools for modelling and analysing CCS processes. Throughout a number of years, CWB has been the tool of choice in courses on concurrency at Aalborg University. Unfortunately, both CWB and CWB-NC are no longer in active development and the latest binaries are from 1999 (CWB) and 2000 (CWB-NC). The download links on the CWB-NC homepage do not work any more and it has become more and more difficult to acquire and install CWB as it relies on an outdated compiler (as a consequence e.g. Mac OS X binaries are not

available). Moreover, CWB is only a command line tool and despite the fast verification algorithms it implements, the graphical interface is lacking. Apart from the fact that Caal provides a modern user interface, integrated process editor and the possibility to visualize the processes and play a variety of (bi)simulation and model checking games, the verification approach also differs. While CWB is using global partitioning algorithms for checking equivalences, we use local on-the-fly approach based on dependency graphs.

Recently, there have been efforts to provide graphical add-ons to CWB as e.g. the Bisimulation Game-Game project [15], but there is no support for model checking games and the tool relies on transition graphs generated by CWB. The tool pseuCo [7] allows to compile an educational Java-based concurrent language into CCS and visualize/minimize the resulting transition systems. TAPAs [4] is another educational tool for specifying and analyzing concurrent processes described in CCSP (CCS plus additional CSP operators). It has a nice GUI but it does not consider bisimulation/model checking games and timed process algebra. Other tools supporting CCS language are more on the experimental level (command line input) and are not targeted towards educational purposes. Let us name here e.g. implementation of CCS in Maude [16] or in Haskell [3].

Finally, there exist mature tools with modern designs like FDR3 [9], CADP [8] and mCRL2 [10], with expressive input languages and efficient analysis methods. Our tool does not aim to compete with them in terms of performance, we are instead focusing on the educational aspects.

2 Modelling Features

Caal supports the CCS and timed CCS (TCCS) input syntax. Let $\mathcal{A}$ be a finite set of channels, let $\overline{\mathcal{A}} = \{\overline{a} \mid a \in \mathcal{A}\}$ be the set of dual channels[1] and let $\mathcal{A}ct = \mathcal{A} \cup \overline{\mathcal{A}} \cup \{\tau\}$ be the set of actions. Let $\mathcal{K}$ be a finite set of process names. The collection of CCS expressions is given by the abstract syntax

$$P, Q ::= K \;\Big|\; \alpha.P \;\Big|\; P + Q \;\Big|\; P \mid Q \;\Big|\; P[f] \;\Big|\; P \setminus L \;\Big|\; 0$$

where $K \in \mathcal{K}$, $\alpha \in \mathcal{A}ct$, $L \subseteq \mathcal{A}$ and $f : \mathcal{A}ct \longrightarrow \mathcal{A}ct$ is the relabelling function satisfying $f(\tau) = \tau$ and $f(\overline{a}) = \overline{f(a)}$ for every $a \in \mathcal{A}ct$. By convention $\overline{\tau} = \tau$. The behaviour of each process name $K \in \mathcal{K}$ is given by its defining equation $K \stackrel{\text{def}}{=} P$. The syntax of TCCS is further extended with the delay prefix operator such that for every nonnegative integer d and a process expression P, we have that $d.P$ is also a process expression.

The SOS rules for the CCS and TCCS operators are given in Table 1. For TCCS we support at the moment the discrete time semantics that is defined in Table 2. The semantics is for simplicity given for single-unit time delays as longer delays are just a syntactic sugar for a series of one time unit delays.

[1] In Caal dual channels are prefixed with an apostrophe and the output bar is displayed automatically by the editor.

$$\text{ACT}\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{SUM1}\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{SUM2}\frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\text{COM1}\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad \text{COM2}\frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \quad \text{COM3}\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{CON}\frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'}\ K \stackrel{\text{def}}{=} P \qquad \text{REL}\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \text{RES}\frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L}\ \alpha, \overline{\alpha} \notin L$$

Table 1: SOS rules for CCS and TCCS

$$\text{ONE}\frac{}{d.P \xrightarrow{1} (d-1).P}\ d \geq 1 \quad \text{ACT}\frac{}{\alpha.P \xrightarrow{1} \alpha.P}\ \alpha \neq \tau \quad \text{REL}\frac{P \xrightarrow{1} P'}{P[f] \xrightarrow{1} P'[f]}$$

$$\text{SUM}\frac{P \xrightarrow{1} P' \quad Q \xrightarrow{1} Q'}{P + Q \xrightarrow{1} P' + Q'} \qquad \text{CON}\frac{P \xrightarrow{1} P'}{K \xrightarrow{1} P'}\ K \stackrel{\text{def}}{=} P \qquad \text{RES}\frac{P \xrightarrow{1} P'}{P \setminus L \xrightarrow{1} P' \setminus L}$$

$$\text{COM}\frac{P \xrightarrow{1} P' \quad Q \xrightarrow{1} Q'}{P \mid Q \xrightarrow{1} P' \mid Q'}\ \text{if } P \mid Q \xrightarrow{\tau} \not\rightarrow$$

Table 2: SOS rules for unit delays in TCCS (d ranges over nonnegative integers)

We assume the classical definitions of weak/strong and timed/untimed equivalences and preorders like simulation and trace preorder/equivalence, and bisimilarity (see e.g. [1]) that are supported in CAAL, including their game characterization via two-player games between attacker (trying to disprove the validity of the equivalence/preorder) and defender (supporting its validity).

As for model checking, the tool supports a subset of the modal μ-calculus [12] with recursively defined fixed points given by the syntax:

$$\phi ::= tt \mid ff \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle\alpha\rangle\phi \mid [\alpha]\phi \mid \langle\langle\alpha\rangle\rangle\phi \mid [[\alpha]]\phi \mid X$$

where $\alpha \in \mathcal{A}ct$ and where X is a variable from a finite set of variables such that every variable has exactly one declaration of the form $X \stackrel{\min}{=} \phi$ (minimum fixed point) or $X \stackrel{\max}{=} \phi$ (maximum fixed point). Here the modal operators are available in their strong variants $\langle\alpha\rangle$ (there is an α-successor) and $[\alpha]$ (for all α-successors), as well as the weak ones $\langle\langle\alpha\rangle\rangle$ and $[[\alpha]]$ that abstract away from τ-actions. We use the abbreviations $\langle A\rangle\phi$ and $[A]\phi$ for a set of actions $A \subseteq \mathcal{A}ct$, standing for $\vee_{\alpha\in A}\langle\alpha\rangle\phi$ and $\wedge_{\alpha\in A}[\alpha]\phi$, respectively. By $\langle -\rangle\phi$ we understand $\langle\mathcal{A}ct\rangle\phi$ and similarly for $[-]\phi$. The same conventions are used for the weak modalities.

For most practical applications it is enough to consider formulae where the recursively defined variables do not contain cyclic references (hence a variable X can refer to itself and/or to another variable Y, but Y may not refer back to X, neither directly or indirectly via other variables). This restriction, adopted by CAAL, allows for faster implementation of the verification engine and makes

the interpretation of model checking games between defender (claiming that a process satisfies a given formula) and attacker (claiming that it does not satisfy the formula) a lot easier as we can always uniquely determine whether we are in the context of a minimum or a maximum fixed point. Defender is then the winner of any infinite play whenever we are in the context of maximum fixed point and attacker is the winner if we are in the minimum fixed-point context.

For guarded CCS processes (where every occurrence of a process name is within the scope of action prefixing) we know that two processes are bisimilar if and only if they satisfy exactly the same set of formulae of the Hennessy-Milner logic [11]. In case of strong bisimilarity we allow only the strong modalities in the formulae, and in case of weak bisimilarity we consider only the weak modalities. The theorem implies that if two processes are not bisimilar, we can find the so-called *distinguishing formula* that is satisfied in one of the processes but not in the other one. This can be useful when debugging CCS processes.

Finally, the tool supports also the extension of the logic with time modalities so that we can have formulae of the form $\langle d \rangle \phi$, $[d]\phi$, $\langle\langle d \rangle\rangle \phi$ and $[[d]]\phi$ where d is a nonnegative integer. The modality $\langle d \rangle \phi$ requires that it is possible to delay d time units and then satisfy ϕ, while the modality $[d]\phi$ expresses that whenever it is possible to delay d time units then ϕ must be satisfied. Even though the future after a given time delay is always deterministic, there is a difference between the two operators as if a process cannot delay d time units (due to some enabled τ actions that are urgent in the TCCS semantics) then $[d]\phi$ will be always satisfied while $\langle d \rangle \phi$ will never be satisfied. The weak time delay modalities allow us to interleave the single-unit time delays with arbitrary many τ-actions. The modalities are in CAAL further extended with time intervals such that $\langle d_1, d_2 \rangle \phi$ with $d_1 \leq d_2$ is the abbreviation for $\langle d_1 \rangle \phi \vee \langle d_1 + 1 \rangle \phi \vee \langle d_1 + 2 \rangle \phi \vee \ldots \vee \langle d_2 \rangle \phi$, and similarly $[d_1, d_2]$ stands for $[d_1]\phi \wedge [d_1 + 1]\phi \wedge [d_1 + 2]\phi \wedge \ldots \wedge [d_2]\phi$. The intervals in weak modalities are defined analogously.

3 Verification Engine

The verification algorithms for both equivalence/preorder checking and model checking are based on a fixed-point computation over a structure called *dependency graph* [13]. Such graphs, for the verification problems in question, can be generated on-the-fly and there exist efficient local algorithms by Liu and Smolka [13] for computing the fixed points.

A *dependency graph* is a pair $G = (V, E)$ where V is a finite set of nodes and $E \subseteq V \times 2^V$ is a finite set of hyperedges of the form (v, T) where $v \in V$ is the source node and the nodes in $T \subseteq V$ are called the target nodes. An *assignment* on G is a function $A : V \to \{0, 1\}$. We define a function F from assignments to assignments as follows: $F(A)(v) = 1$ if and only if there is $(v, T) \in E$ such that $A(v') = 1$ for all $v' \in T$. As all assignments form a complete lattice w.r.t. to the natural point-wise ordering and the function F is monotonic, there is by Knaster-Tarski theorem a unique minimum and maximum fixed point of the

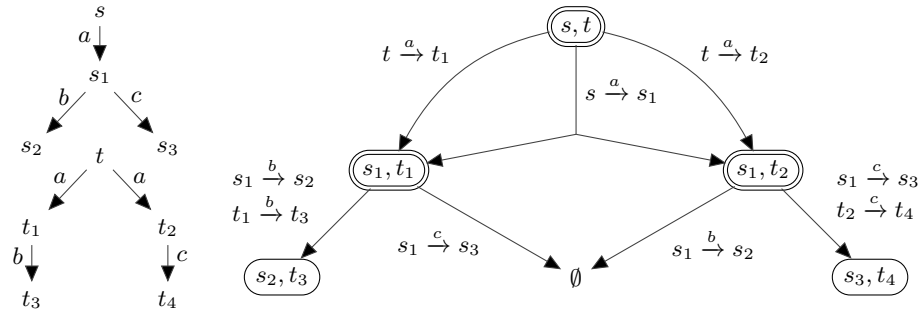


Fig. 2: Two processes s and t (left) and the constructed dependency graph (right)

function F, denoted by A_{min} resp. A_{max}. The fixed points for G can be computed in linear-time by the use of local on-the-fly algorithms [13].

We shall now hint at how the verification questions for CCS/TCCS can be encoded in fixed-point computations on dependency graphs. The idea, depicted in Figure 2 for strong bisimulation, is that nodes in the dependency graph are pairs of processes and for any transition from one of the two processes, we create a new hyperedge with targets that correspond to all possible transitions under the same label from the other process. The hyperedges in the dependency graph are annotated with the transitions that initiated their creation. If for some pair of states there is a transition for which the other process does not have any answer, the resulting set of target nodes is empty and the created hyperedge ensures that the pair will get the value 1 in the minimum fixed-point assignment (denoted in our example by a double circle around the pair). One can prove that for any pair of nodes (s', t') in the dependency graph it holds that $s' \sim t'$ (the states are strongly bisimilar) if and only if $A_{min}((s', t')) = 1$. In order to establish that $A_{min}((s, t)) = 1$, it is enough to explore only a fraction of the dependency graph (e.g. constructing only two hyperedges from (s, t) to (s_1, t_1) and from (s_1, t_1) to the emptyset is sufficient). As the construction of the complete dependency graph can often be avoided by using on-the-fly algorithms, it is sometimes possible to show nonequivalence even for processes with infinitely many reachable states, a situation where the traditional partitioning algorithms will never terminate.

We can also use the computed fixed point on the dependency graph to derive a distinguishing formula for the processes s and t in Figure 2. First, for every node that has a hyperedge with an empty set of targets, we can directly find such a formula, like for the node (s_1, t_1) where $s_1 \models \langle c \rangle tt$ while $t_1 \not\models \langle c \rangle tt$. From this formula we can now inductively construct a distinguishing formula $[a]\langle c \rangle tt$ for the root node (s, t). Note that this is not the only distinguishing formula, if we e.g. use instead the hyperedge from (s, t) with the two target nodes, we derive the formula $\langle a \rangle(\langle c \rangle tt \wedge \langle b \rangle tt)$ that is arguably more complex than the formula $[a]\langle c \rangle tt$. The problem of finding the simplest distinguishing formula is nontrivial and CAAL uses a greedy heuristic approach to report reasonably small distinguishing formulae.

Algorithm 1 Simple Communication Protocol (CCS) in CAAL

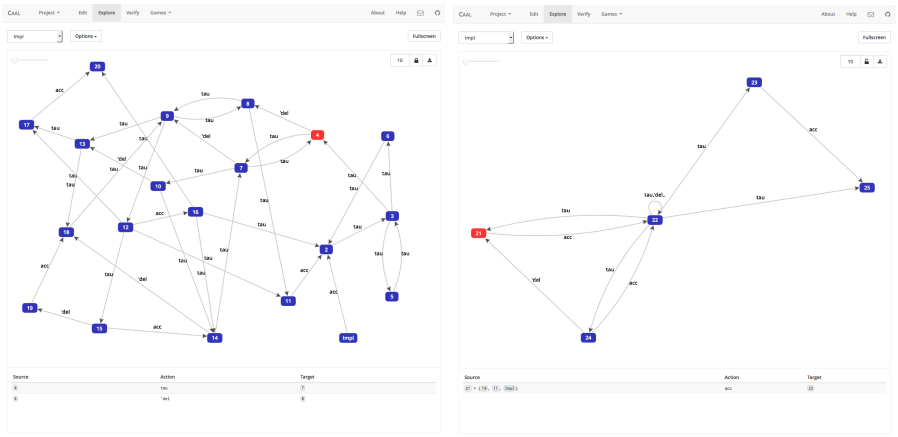
```
 1: Send = acc.Sending;
 2: Sending = 'send.Wait;
 3: Wait = ack.Send + error.Sending + 'send.Wait;

 4: Rec = trans.Del;
 5: Del = 'del.Ack;
 6: Ack = 'ack.Rec;

 7: Med = send.Med';
 8: Med' = 'trans.Med + tau.Err + tau.Med;
 9: Err = 'error.Med;

10: set L = {send, trans, ack, error};
11: Impl = (Send | Med | Rec) \ L;

12: Spec = acc.'del.Spec;
```

The approach via dependency graphs is used also for trace-like equivalences and the corresponding dependency graphs are described in master theses available at the tool's homepage. For recursive formulae the construction of dependency graphs requires several copies of the graphs, one for each fixed-point definition, but the same uniform approach is also used here. Finally, the dependency graphs are used for guiding the tool in bisimulation and model checking games.

4 Case Study

We shall now present a simplified version of a communication protocol, where a sender is supposed to forward messages through unreliable medium to a receiver, who then acknowledges it via a direct handshake after which the protocol is again ready to accept another message. A more sophisticated variant of such a protocol (e.g. the Alternating Bit Protocol [2]) is a typical mini-project exercise that we use in our Semantics and Verification course.

The CCS processes describing the protocol are given in Algorithm 1. The sender, defined at lines 1–3, receives a message `acc` from the environment, forwards the message via the internal channel `send` to the medium and then waits for the acknowledgment, an error message from the medium, or tries to resend the message. The receiver, defined at lines 4–6, can receive the message through the medium via the internal channel `trans`, deliver the message to its environment via the output action `'del` and then acknowledge this to the sender. The medium, defined at lines 7–9, communicates with the sender/receiver via the channels `send` and `trans` but can also enter an error state and inform the sender about this (line 9) or silently lose the message and enter its initial state. The implementation of the protocol (line 11) is a parallel composition of the three components described above where all channels except for `acc` and `'del` are restricted, enforcing a handshake synchronization over these channels. Finally, at line 12, we can see the abstract specification of the protocol. We have



(a) Before weak bisimulation collapse (b) After weak bisimulation collapse

Fig. 3: Reachable state-space for the process `Impl`

deliberately introduced some design errors in order to demonstrate the typical mistakes the students make when modelling more advanced variants of communication protocols. In the rest of this section, we shall demonstrate the debugging options that CAAL offers for analysing and correcting such mistakes.

By entering the verification module of CAAL, we can promptly find out that the processes `Impl` and `Spec` are not weakly bisimilar. We can now enter the explorer module in order to visualize and interactively explore the transition system of the process `Impl` as depicted in Figure 3a, however, even for this small example, the system is already too large. We can choose to visualize the collapsed transition system where all weakly bisimilar states are merged together as shown in Figure 3b and here we can already see some design issues. We can e.g. observe that the implementation contains a deadlock (the right-most state).

In general, the labelled transition systems (even after the bisimulation collapse) are often too large to analyse manually. Hence, if two processes are not weakly bisimilar, a natural question to ask is whether they provide the same weak traces (sequences of visible actions). It appears that this is not the case for our example and CAAL informs the user that the process `Impl` can perform the sequence of visible actions `acc`, `'del`, `'del` and such a trace is not possible in the specification. By analysing the trace in the game module, we can see that the problem is at line 3 where the sender has the option to resubmit the message unboundedly many times. Hence we may decide to remove this resubmission option and modify the CCS definition as `Wait = ack.Send + error.Sending`. After this fix, we can now verify that the implementation and the specification are weakly trace equivalent, while they are still not bisimilar. We can ask CAAL to generate a distinguishing formula that holds in `Impl` but not in `Spec` and the tool returns the formula `<<acc>>[['del]]F` that states that the implementation can perform the visible action `acc`, possibly with some additional τ-transitions before and after, such that after this it is not possible to perform the action `'del`

Algorithm 2 Time Annotated Communication Protocol (TCCS) in CAAL

```
1: Send = acc.2.'send.1.ack.2.Send;
2: Rec = trans.1.'del.2.'ack.8.Rec;
3: Med = send.(3.'trans.Med + 5.tau.Med);
4: Impl = (Send | Med | Rec) \ {send, trans, ack};
5: Spec = acc.'del.Spec;
```

(not even preceded by some τ-actions). By entering the game module, the user can play a game against the computer (playing defender) that will reveal to the user (playing attacker) that the formula holds in the state `Impl`. The game will in fact reveal the presence of a deadlock configuration that we already observed in the explorer.

Alternatively, we can directly formulate the deadlock property as a recursive HML formula `X min= [-]ff or <->X`. CAAL confirms that the implementation satisfies the property `X` and in the game module, the computer can convince the user that a deadlock is indeed reachable. The analysis of the discovered deadlock points to the fact that medium should not be allowed to silently discard messages. After changing the definition at line 8 into `Med' = tau.Err + 'trans.Med` we finally achieve a correct implementation (weakly bisimilar to its specification).

We may also ask if the protocol contains a reachable livelock (an infinite sequence of τ-actions that can be executed in a row). CAAL allows to formulate this property using two recursively defined variables `Y min= Z or <->Y` (claiming the reachability of livelock) and `Z max= <tau>Z` (expressing the existence of an infinite τ-sequence). The implementation indeed contains a livelock and the game provides a convincing argument for this fact.

CAAL moreover allows to model TCCS processes. A variant of the communication protocol is given in Algorithm 2, where both the sender, receiver and medium have been annotated with delays such that e.g. the medium needs 3 time units to deliver the message but it will lose it after 5 time units. CAAL will show that the processes `Impl` and `Spec` are weakly untimed bisimilar. However, if the receiver process gets just little bit slower and the delay prefix `8` at line 2 is replaced with delay `9` then the weak untimed bisimulation equivalence does not hold anymore as it is now possible that the medium loses a message.

We can also verify that the TCCS process `Impl` satisfies the formula

```
X max= [acc]<<0,6>><'del>tt and [-]X;
```

expressing the invariant that whenever the action `acc` is performed then the message can be delivered within 6 time units. The property `X` actually does not hold but if we ask instead whether the message can be delivered withing 7 time units then it is satisfied.

5 Conclusion

We presented CAAL, an educational tool for modelling and analysing CCS processes. The tool runs in a browser with limited computational resources but it

benefits from the efficient on-the-fly algorithms. This is clearly sufficient for the typical student exercises and mini-projects. At the moment, we are exploring the parallelization of the fixed point computation and outsourcing this work to a super-computer via the approach "verification as a web-service". Our Reactive System book [1] is now used at more than 18 universities around the world and we expect that CAAL, once publicly announced, becomes a natural supplement to the concurrency courses based on the CCS formalism.

References

[1] L. Aceto, A. Ingolfsdottir, K.G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

[2] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, 1969.

[3] Arnar Birgisson. CCS model checker in Haskell. `https://github.com/arnar/ccs-searching`, 2009. [Online; accessed 03-August-2015].

[4] F. Calzolai, R. De Nicola, M. Loreti, and F. Tiezzi. TAPAs: A tool for the analysis of process algebras. *Transactions on Petri Nets and Other Models of Concurrency*, 5100:54–70, 2008.

[5] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst.*, 15(1):36–72, 1993.

[6] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In *CAV'96*, volume 1102 of *LNCS*, pages 394–397. Springer, 1996.

[7] F. Freiberger, S. Biewer, and P. Held. PseuCo. `http://pseuco.com`, 2014. [Online; accessed 03-August-2015].

[8] H. Garavel, F. Lang, R. Mateescu, and W.Serwe. CADP 2011: A toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.

[9] T. Gibson-Robinson, Ph. Armstrong, A. Boulgakov, and A.W. Roscoe. FDR3—A modern refinement checker for CSP. In *TACAS'14*, volume 8413 of *LNCS*, pages 187–201. Springer, 2014.

[10] J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

[11] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, 1985.

[12] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[13] X. Liu and S.A. Smolka. Simple linear-time algorithms for minimal fixed points. In *ICALP'98*, volume 1443 of *LNCS*, pages 53–66. Springer, 1998.

[14] R. Milner. A calculus of communicating systems. *LNCS*, 92, 1980.

[15] M. Mosegaard and C. Brabrand. The bisimulation game game. `http://www.brics.dk/bisim/`, 2006. [Online; accessed 03-August-2015].

[16] A. Verdejo and N. Marti-Oliet. Executing and verifying CCS in Maude. Technical report, Dpto. Sistemas Informaticos y Programacion, Universidad Complutense de, 2002. `http://maude.cs.uiuc.edu/maude1/casestudies/ccs/`.