CAAL Tutorial

Jacob Karsten Wortmann, Jesper Riemer Andersen,
Nicklas Andersen, Mathias Munk Hansen,
Simon Reedtz Olesen, Søren Enevoldsen

June 9, 2015

This tutorial gives an informal introduction to the main features of CAAL and how to use them. CAAL supports the process algebras Calculus of Communicating Systems (CCS) and Timed CCS (TCCS), and both equivalence and model checking analysis of processes through verification and games. CAAL consists of four different modules; an editor module for modelling processes, a module for visualization of processes, a module for equivalence and model checking, and a game module. CAAL is available at:

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

1 The Language CCS

CCS is a process algebra used to model concurrent systems. We shall now informally introduce CCS.

The most basic process of all is the 0 (or *nil*) process. It cannot perform any action whatsoever and thus stops all computation. The most basic process constructor in CCS is *action prefixing*. The formation rule for action prefixing is as follows:

If P is a process and a is a label, then $a.P$ is a process.

Using the formation rule for action prefixing and the 0 process we can build two example processes:

$$\texttt{shake}.0 \quad \texttt{shake}.\texttt{walk}.0 \ .$$

The first process can only perform the shake action and then *dies* (becomes the 0 process). The second process is a more complex process, which after performing the shake action, can also perform the walk action. Names can also be assigned to processes. For example, we can give the second process a name:

$$\texttt{Boy} = \texttt{shake}.\texttt{walk}.0 \ . \tag{1}$$

Naming processes allows us to introduce recursive definitions of process behaviors. For example, we can define a recursive process specification as follows:

$$\texttt{Tree} = \texttt{shake}.\overline{\texttt{apple}}.\texttt{Tree} \ .$$

This tree can be shaken which causes it to deliver an apple and afterwards returns to its initial state where it can be shaken again. Note the bar over $\overline{\text{'apple}}$ (the apostrophe denotes a bar in CAAL), which indicates that it is an output action. This tree only allows one type of apple. In order for the tree to support multiple colors of apples, we use the *choice operator*. Now the tree can be defined as:

$$\texttt{ColorTree} = \texttt{shake}.(\overline{\texttt{'greenapple}}.\texttt{ColorTree} + \overline{\texttt{'redapple}}.\texttt{ColorTree}) \ . \tag{2}$$

The idea is that after the tree has been shaken, it can deliver either a green or a red apple. In general, the formation rule for choice is:

If P and Q are processes, then $P + Q$ is also a process.

The process $P + Q$ is able to do either P or Q, but not both. As soon as P is performed any further execution of Q is preempted and vice versa.

Another operator is the *parallel composition operation*. Composition describes two or more processes running in parallel and possibly interacting with each other. For example, if we continue the example from Equation 1, we can shake the tree in order to receive an apple and then walk to the next tree after an apple has fallen to the ground. This can be described by the CCS process:

$$\texttt{Girl} = \overline{\texttt{'shake}}.\texttt{apple}.\overline{\texttt{'walk}}.\texttt{Girl} \ . \tag{3}$$

The CCS expression $\texttt{Tree} \mid \texttt{Girl}$ describes a system consisting of two processes; the tree and the girl. These two processes can communicate through their shared communication channels; shake and apple.

However, neither the girl nor tree are required to communicate with each other. They could communicate over their channels with any other processes they have been composed with, or simply perform the shake, apple, or $\overline{\text{walk}}$ actions directly without communication.

P and Q may proceed independently or they may communicate through shared channels.

When two processes communicate through the same input and output action the resulting action is called a τ-action. It might be best if only one had access to the apples that fall from the tree. CCS allows this through an operation called *restriction*. This allows us to hide certain channels from the environment. If we continue from Equation 3 and expand the example to accept red or green apples:

$$\begin{aligned} \texttt{Man} &= \overline{\texttt{'shake}}.(\texttt{Man1} + \texttt{Man2}) \ , \\ \texttt{Man1} &= \texttt{redapple}.\overline{\texttt{'walk}}.\texttt{Man} \ , \\ \texttt{Man2} &= \texttt{greenapple}.\overline{\texttt{'throw}}.\texttt{Man} \ . \end{aligned} \tag{4}$$

Now we can define the Orchard using the ColorTree from Equation 2 and the refined Man from Equation 4:

$$\texttt{Orchard} = (\texttt{ColorTree} \mid \texttt{Man}) \setminus \{\texttt{shake, redapple, greenapple}\} \ . \tag{5}$$

The restricted channels shake, redapple, and greenapple may only be used for communication between the tree and the man. Their scope is restricted to the process Orchard. In general, the formation rule for restriction can be described as follows:

If P is a process and L is set of channel names, then $P \setminus L$ is a process.

In $P \setminus L$ the channel names in L can only be used to communicate within P. It might be beneficial for the orchard to have access to other sorts of fruit. This can be done by defining a generic orchard that can be shaken, then drop its fruit and reset:

$$\texttt{GenericOrchard} = \texttt{shake}.\overline{\texttt{'fruit}}.\texttt{GenericOrchard} \ .$$

Through appropriate renaming of the GenericOrchard it is possible to obtain a more specific Orchard. For example:

$$\texttt{PearOrchard} = \texttt{GenericOrchard}\ [\texttt{pear/fruit}] \ .$$

PearOrchard is a process that behaves like GenericOrchard but outputs pears instead of a generic fruit. The renaming operation can be described as:

If P is a process and f is a function from labels to labels, then $P[f]$ is a process.

2 The Language TCCS

TCCS is an extension of CCS with time, which means that we still have all the syntactical elements of CCS but with a new syntactic element, the *delay prefixing operator*. With this operator we can model processes like

$$\texttt{5.a.0} \ ,$$

which means that after delaying for 5 time units the a-action becomes available.

We extend the Orchard example and add time to it. We add a time constraint to the tree specifying that if the falling apple has not been caught within 3 time units then it falls to the ground. Extending the ColorTree from Equation 2 we get:

$$\texttt{ColorTree} = \texttt{shake.ColorTree1} \ ,$$
$$\texttt{ColorTree1} = \overline{\texttt{'greenapple}}.\texttt{ColorTree} + \overline{\texttt{'redapple}}.\texttt{ColorTree} + \texttt{3.tau.ColorTree} \ .$$

The ColorTree has the choice of dropping either a green or a red apple. If the tree drops a particular apple then it commits to that choice, but simply delaying will not commit to any choice. For example, after delaying for 2 time units the tree can still drop green or red apples.

However, after 3 time units the τ-action becomes available which prevents any further delays. An action must be performed immediately when a τ-action is available. If no one is ready to catch the apple within 3 time units the apple falls to the ground.

Let us say that after the man has shaken the tree he needs to rest for 2 time units before he is ready to catch an apple. Extending the Man from Equation 4 we get:

$$\texttt{Man} = \overline{\texttt{'shake}}.2.(\texttt{Man1 + Man2}) \ .$$

It is not possible for the man to rest for more than 2 time units because a handshake is available between the Man and the ColorTree (i.e. a τ-action becomes available). We can also define an unfit man who requires a longer break after shaking the tree:

$$\texttt{SlowMan} = \overline{\texttt{'shake}}.5.(\texttt{Man1 + Man2}) \, .$$

If we define the orchard as

$$\texttt{Orchard} = (\texttt{ColorTree} \mid \texttt{SlowMan}) \setminus \{\texttt{shake, redapple, greenapple}\} \ .$$

then the slow man will never be able to catch an apple since his required break makes him unable to catch the apples before they fall to the ground.

3 Editor

The editor is used to input CCS and TCCS programs. The editor has full support for CCS and TCCS syntax, and features live syntax checking to assist the user if syntactical errors occur. The "Parse"-button will notify the user of any contextual errors, such as referencing an undefined process. Furthermore, CAAL supports saving of the project to both a local file and the browser cache, as well as an autosave feature that allows the user to restore unsaved work if an unexpected error should occur. Using the editor we can input the examples from Equation 2 and Equation 4 as shown in Figure 1.

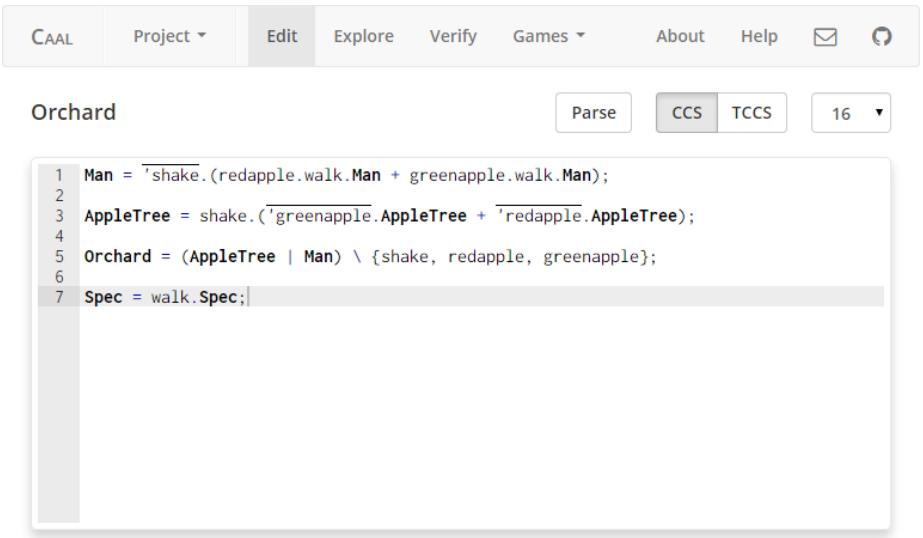


Figure 1: Editor.

4 Verification

After having defined a process in the editor we may want to verify its correctness. We introduce the several forms of verification that CAAL supports.

4.1 Equivalence Checking

CAAL supports the following equivalences and preorders:

- simulation,
- simulation equivalence,
- bisimulation,
- trace inclusion, and
- trace equivalence.

This section focuses on bisimulation. Strong bisimulation is a notion relating two processes such that whenever one of the processes can perform an α-action the other process must also be able to perform an α-action. The resulting pair must again be related by strong bisimulation.

We also have the notion of weak bisimulation. We use the term "weak" to indicate that we abstract away from τ-actions. Whenever one of the processes can perform an α-action the other process also must be able to perform a matching α-action, where it is allowed to perform zero or more τ-actions before and after performing the α-action. The resulting pair must again be related by weak bisimulation.

Example 1

We have the CCS processes:

$$\begin{aligned}
\texttt{Man} &= \overline{\texttt{'shake}}.(\texttt{redapple.walk.Man} + \texttt{greenapple.walk.Man}) \ , \\
\texttt{AppleTree} &= \texttt{shake}.(\overline{\texttt{'greenapple}}.\texttt{AppleTree} + \overline{\texttt{'redapple}}.\texttt{AppleTree}) \ , \\
\texttt{Orchard} &= (\texttt{AppleTree} \mid \texttt{Man}) \ \setminus \{\texttt{shake, redapple, greenapple}\} \ , \\
\texttt{Spec} &= \texttt{walk.Spec} \ ,
\end{aligned}$$

We want to check if the processes Orchard and Spec are strongly or weakly bisimilar. Figure 2 shows the result of the verification. The processes Orchard and Spec are not strongly bisimilar, but they are weakly bisimilar, as indicated by the red cross and the green check mark, respectively.

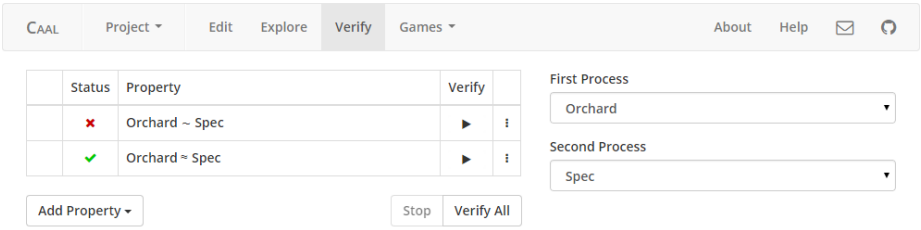


Figure 2: Verification of bisimulation.

4.2 Model Checking

CAAL supports model checking through use of recursive Hennessy-Milner Logic (HML) formulas. HML formulas are used to check if a given process satisfies certain properties. For instance we might want to check if our man:

- is always able to walk after receiving an apple,
- is able to shake the tree right now,
- is able to get hold of a red apple.

CAAL has support for the full syntax and semantics of recursive HML, and also supports formulas with multiple nested variables, with the restriction that variables are not allowed to be mutually recursive.

$$\begin{aligned} &\texttt{X min= <a>X or Y} \\ &\texttt{Y max= [b]Y} \end{aligned} \tag{6}$$

Equation 6 is an example of a supported HML formula.

$$\begin{aligned} &\texttt{X min= <a>X or Y} \\ &\texttt{Y max= [b]Y and X} \end{aligned} \tag{7}$$

Equation 7 is an example of an HML formula that is not allowed because Y refers back to X.

Example 2

We have the CCS processes

$$\begin{aligned} \texttt{Man} &= \overline{\texttt{'shake}}.(\texttt{redapple.walk.Man + greenapple.walk.Man}) \ , \\ \texttt{AppleTree} &= \texttt{shake}.(\overline{\texttt{'greenapple}}.\texttt{AppleTree + } \overline{\texttt{'redapple}}.\texttt{AppleTree}) \ , \\ \texttt{Orchard} &= (\texttt{AppleTree} \mid \texttt{Man}) \setminus \{\texttt{shake, redapple, greenapple}\} \ . \end{aligned}$$

We want to check if it is possible to reach a state from the Orchard where the Man will never be able to perform a *walk*-transition again. We can express this as the recursively defined property

$$\texttt{X min= [[walk]]ff or <->X},$$

where – is the set of all actions. Figure 3 shows the result of the verification. As we can see, this property is not satisfied, as indicated by the red cross.

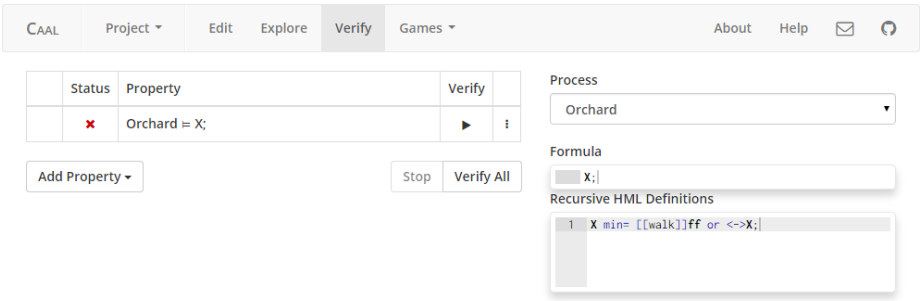


Figure 3: Verification of a recursive HML formula.

4.3 Timed Equivalence and Model Checking

When verifying TCCS we support the same equivalences and preorders as mentioned earlier, as well as an extended version of HML with time called Timed HML (THML). Strong timed bisimulation is almost the same as regular strong bisimulation. Whenever a process can make a move by some action α, the other process must be able to match the move by the same action α. Whenever a process can make a delay, the other process must be able to match the delay. The resulting pairs must again be related by strong timed bisimulation.

Just like it can be useful to abstract away from τ in CCS, it can be useful to abstract away from time in TCCS, which is called "untimed".

Example 3

We have the TCCS processes:

$$\begin{aligned} \texttt{Man} &= \overline{\texttt{'shake}}.2.(\texttt{redapple.walk.Man + greenapple.walk.Man}) \ , \\ \texttt{Tree} &= \texttt{shake}.(\overline{\texttt{'greenapple}}.\texttt{Tree} + \overline{\texttt{'redapple}}.\texttt{Tree} + 3.\texttt{tau.Tree}) \ , \\ \texttt{Orchard} &= (\texttt{Tree} \mid \texttt{Man}) \setminus \{\texttt{shake, redapple, greenapple}\} \ , \\ \texttt{Spec} &= \texttt{walk.Spec} \ , \end{aligned}$$

We want to check if the Orchard is weakly timed or weakly untimed bisimilar to Spec. The Orchard is not weakly timed bisimilar to Spec, but they are weakly untimed bisimilar shown in Figure 4.

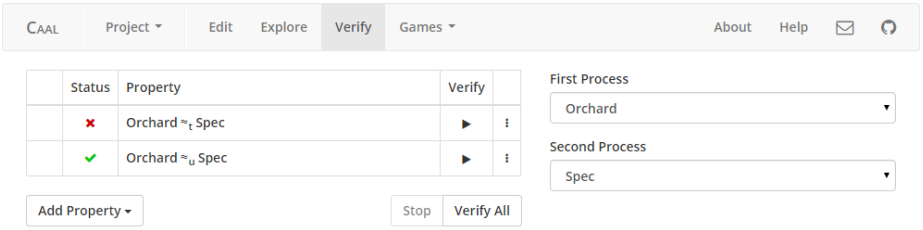


Figure 4: Verification of bisimulation with time.

As seen in Example 2, HML allows us to verify that the system satisfies certain properties, but it is often interesting to verify that the system does so with respect to time.

Example 4

We want to verify that the Man from Example 3 can never receive a red apple if he waits for less than 2 time units after shaking the tree. We can express this property as the recursively defined THML formula:

```
X max= ['shake]<0,1>[redapple]ff and X;
```

As seen in Figure 5 the property is satisfied.

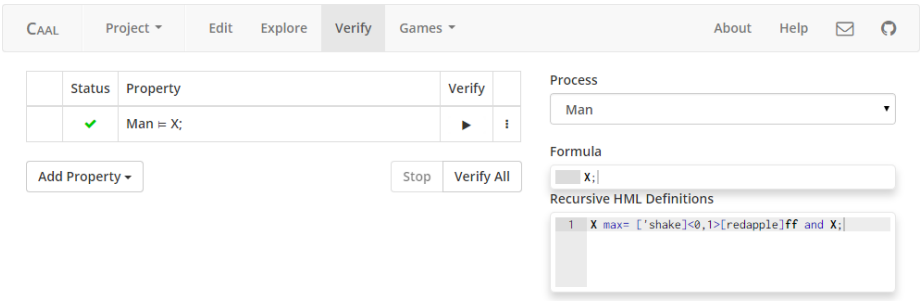


Figure 5: Verification of a recursive THML formula.

5 Debugging Options

Verifying properties for CCS processes might not always yield the expected result. This might mean a bug is present in the CCS processes. We introduce the tools available for debugging in CAAL.

5.1 Explorer

The explorer makes it possible to graphically explore the Labelled Transition System (LTS) generated by a process. To begin, the desired process is selected

from the drop-down menu at the top left. The outgoing transitions from the selected process are then displayed. The explorer is shown in Figure 6.

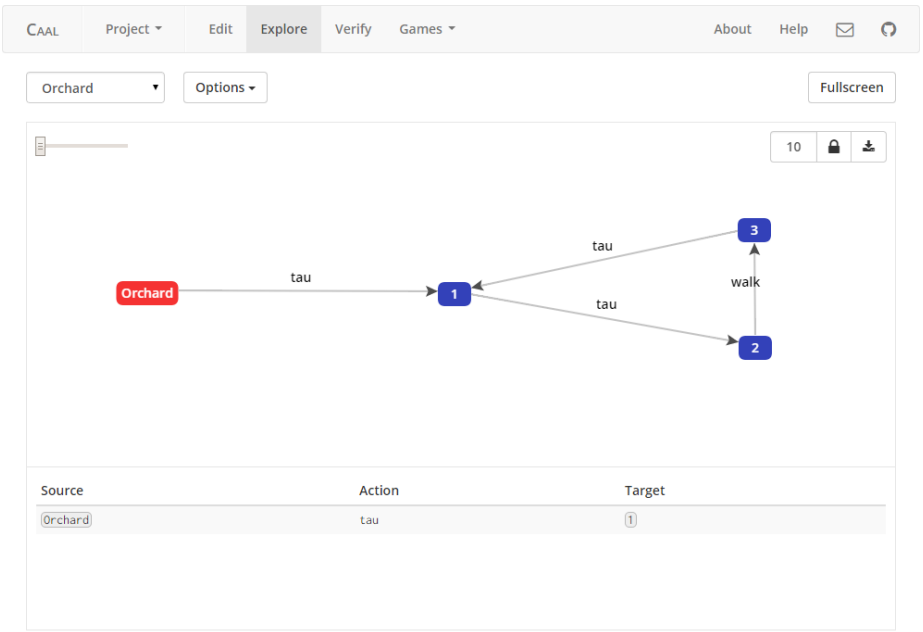


Figure 6: Explorer.

The states in the LTS can be selected by clicking them. The currently selected state is colored red and the outgoing transitions from that state will be displayed in the table below the LTS.

A number of different options are available in the explorer:

Zoom The slider at the top left will zoom in on the currently selected state. Sometimes the LTS becomes too large to tell the different states and transitions apart, which is when zooming helps. When zoomed in, the LTS will automatically be centered on the currently selected state whenever it is changed.

Expand Depth The number at the top right is the number of states to expand the LTS with. For example, if we have a depth of five, then all states which are up to five transitions away from the currently selected state will be displayed.

Lock The padlock at the top right will lock/unlock the LTS. The states in the LTS are automatically positioned, but may sometimes become cluttered if there are too many states or transitions. Locking the LTS makes it possible to manually rearrange the states in the LTS.

Export The download button at the top right will download an image of the currently displayed LTS.

Transitions The LTS can be displayed using either strong or weak transitions. By default the LTS displayed is using strong transitions. In the case of TCCS, there are also options for timed and untimed transitions.

Collapse The LTS can be collapsed using either strong or weak bisimulation collapse. Strong bisimulation collapse means that all states which are strongly bisimilar are collapsed into a single state. Figure 7 shows the Orchard with strong bisimulation collapse, and Figure 8 shows the Orchard with weak bisimulation collapse. In cases where the LTS becomes very large the zoom option might not be sufficient. In such cases all unwanted actions can be relabelled to τ and removed using weak bisimulation collapse.

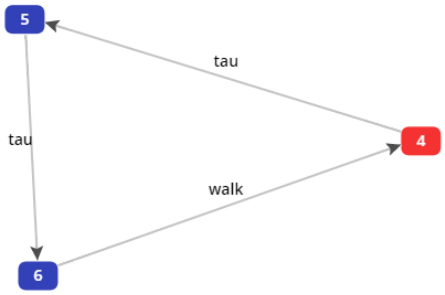


Figure 7: Orchard with strong bisimulation collapse.

Figure 8: Orchard with weak bisimulation collapse.

5.2 Games for Equivalences and Preorders

CAAL supports games for the following equivalences and preorders:

- strong/weak bisimulation,
- strong/weak simulation, and
- strong/weak simulation equivalence.

Furthermore, CAAL also has games for the timed and untimed variations of the above equivalences and preorders. In this section we will focus on the game for strong bisimulation. The games for the other equivalences and preorders are similar, but with different rules.

The strong bisimulation game consists of an "attacker", a "defender", and two processes s and t to play on. The goal of the attacker is to show that the

processes are not strongly bisimilar, and the goal of the defender is to show that they are. The game is played over a number of rounds, where each round starts in a pair of states called the current configuration. Initially, the current configuration will be (s, t). Each round is played according to the following rules:

1. The attacker performs a transition under some action α from s to s' or from t to t'. If the attacker cannot perform any transition the defender wins.
2. The defender must now respond with a transition.
   - If the attacker played s to s', then the defender must perform a transition t to t' under the same action α. If the defender cannot perform any transitions, then the attacker wins.
   - If the attacker played t to t', then the defender must perform a transition s to s' under the same action α. If the defender cannot perform any transitions, then the attacker wins.
3. The game continues for another round with the pair (s', t') as the current configuration.

If a cycle is detected in the game, i.e. if we reach a configuration (s', t') which has previously been the current configuration the defender wins the game.

If the attacker has a universal winning strategy, then s and t are not strongly bisimilar. If the defender has a universal winning strategy, then s and t are strongly bisimilar. If a player has a universal strategy, then that player will always be able to win regardless of what the other player does.

We show an example of a strong bisimulation game. Instead of showing the simple game between the Orchard and Spec processes, we will define a pear tree to play against the apple tree. We can define the pear tree as a relabelling of the ColorTree from Equation 2:

$$\texttt{PearTree} = \texttt{ColorTree}\ [\texttt{pear/greenapple}, \texttt{pear/redapple}]\ .$$

Figure 9 shows a strong bisimulation game where the player is playing as attacker against the computer in the defender role.

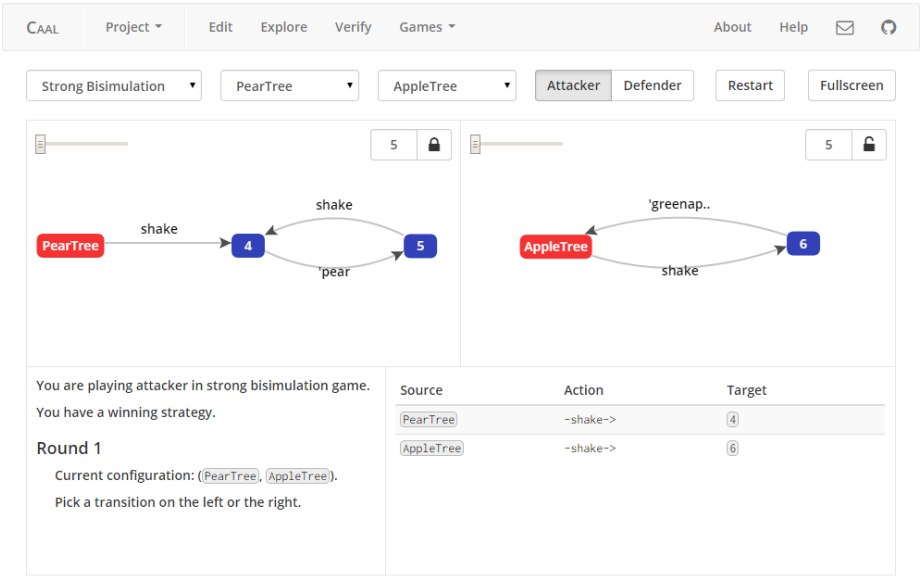


Figure 9: Screenshot of the strong bisimulation game.

The game settings in the top specifies that it is a strong bisimulation game between the processes PearTree and ColorTree where the player is playing as attacker. We also have the option to restart the game to the (PearTree, ColorTree) configuration.

The LTSs generated by the processes PearTree and ColorTree are shown in Figure 10, where the current configuration of the game is highlighted in red. The two LTSs have the same options (e.g. lock, zoom, etc.) as in the explorer.

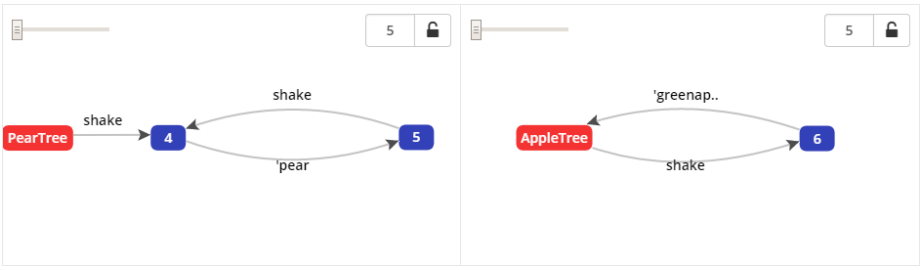


Figure 10: Game LTSs.

Figure 11 shows the different transitions available to the player. It consists of three columns:

Source The source state of the transition. Can be either the current state in the left LTS or the current state in the right LTS.

Action The label of the transition.

Target The destination state of the transition.

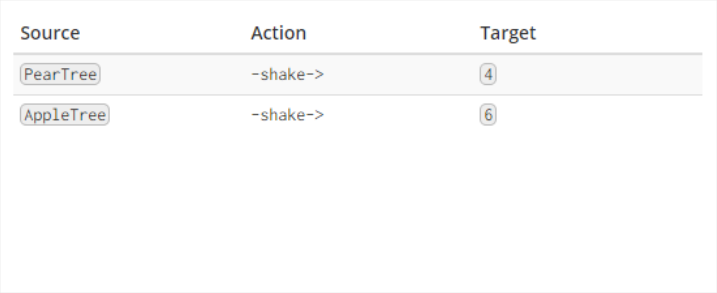


Figure 11: Available transitions.

Figure 12 shows the different steps of a full game in the game log. The initial state of the game log is shown in Figure 12a, where the role of the player and whether or not the player has a universal winning strategy is shown. The player then knows if a loss was due to a bad move. The game log then prompts the player to pick a transition.

Figure 12b shows the game log after the player has made the attack

$$\text{PearTree} \xrightarrow{\text{shake}} 4$$

on the left LTS, where 4 is the identifier of the target state.

Figure 12c shows the response of the defender

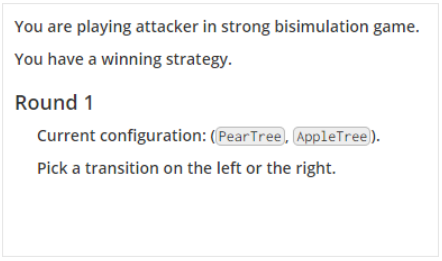
$$\text{ColorTree} \xrightarrow{\text{shake}} 6$$

on the right. The next round of the game starts and the game log shows the current configuration of the game $(4, 6)$. The player can now attack again on the left or right.

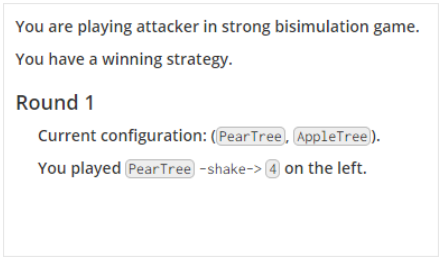
Figure 12d shows the player attacking with the transition

$$4 \xrightarrow{\text{pear}} 5$$

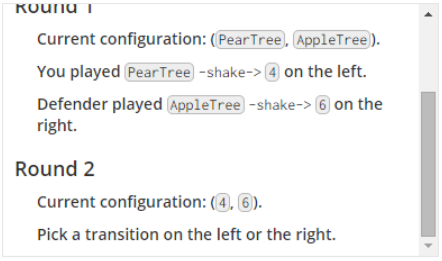
on the left. The defender cannot match the pear transition on the right. The player wins the game which means that the processes PearTree and ColorTree are not strongly bisimilar.



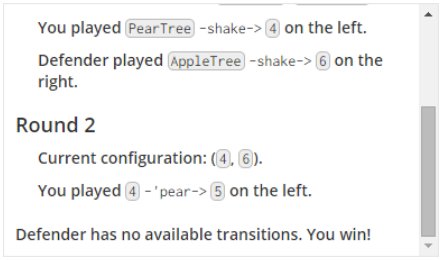
(a) Game log with introduction.



(b) Game log after an attack.



(c) Game log after a defend.



(d) Game log with a winner.

Figure 12: The game log.

5.3 HML Game

An HML game consists of an "attacker", a "defender", a process s, and a formula F. A play of a game starting from the start state s is a maximal sequence of configurations formed by the players according to the following rules. Each round either the attacker or the defender picks a successor configuration if possible.

- The attacker picks a configuration when the formula is of the form $(s, F_1 \wedge F_2)$, or when the choices are either $(s, [\alpha]\, F)$ or $(s, [[\alpha]]F)$.
- The defender picks a configuration when the formula is of the form $(s, F_1 \vee F_2)$, or when the choices are $(s, \langle\alpha\rangle\, F)$ or $(s, \langle\langle\alpha\rangle\rangle F)$.

The winner depends on which configuration the game ends in, or alternatively the context of an infinite play.

- The attacker is the winner in every play ending in a configuration of the form (s, ff) or in play in which the defender gets stuck.
- The defender is the winner in every play ending in configuration of the form (s, tt) or in a play in which the attacker gets stuck.
- The attacker is the winner in every infinite play in context X provided that X is a defined as a minimum fixed-point: $X \overset{\min}{=} F$. The defender is the winner in every infinite play provided that X is defined as a maximum fixed-point: $X \overset{\max}{=} F$.

Figure 13 shows an HML game where the user is playing as defender against the computer.

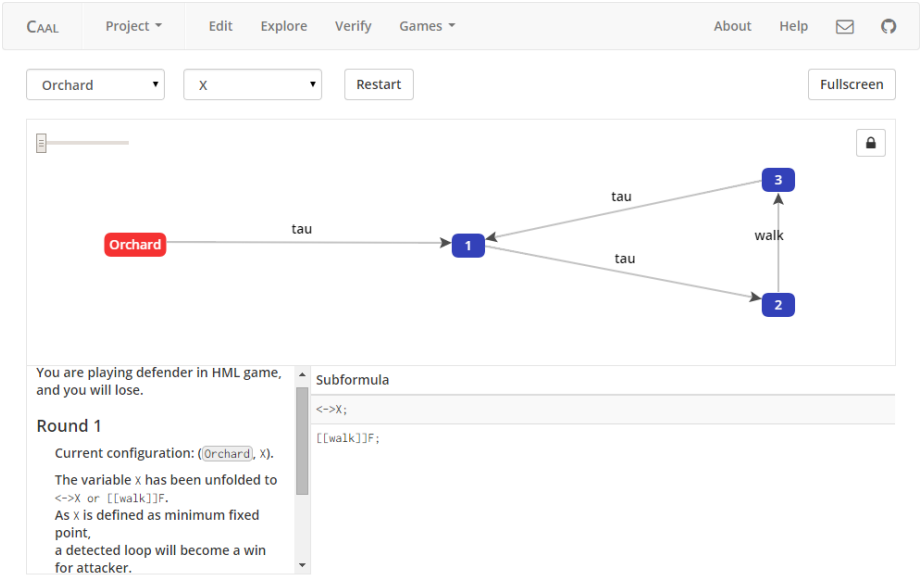


Figure 13: HML Game.

The game consists of a few different elements:

- the process and formula at the top left,
- the LTS in the middle,
- game log at the bottom left, and
- subformula and transition table at the bottom right.

Using Example 2 we now want to play an HML game to verify that the result is correct and that the formula is indeed not satisfied. We have the Orchard process given by

$$\texttt{Man} = \overline{\texttt{'shake}}.(\texttt{redapple.walk.Man + greenapple.walk.Man}) \ ,$$
$$\texttt{AppleTree} = \texttt{shake}.(\overline{\texttt{'greenapple}}.\texttt{AppleTree + } \overline{\texttt{'redapple}}.\texttt{AppleTree}) \ ,$$
$$\texttt{Orchard} = (\texttt{AppleTree} \mid \texttt{Man}) \setminus \{\texttt{shake, redapple, greenapple}\} \ ,$$

and the formula

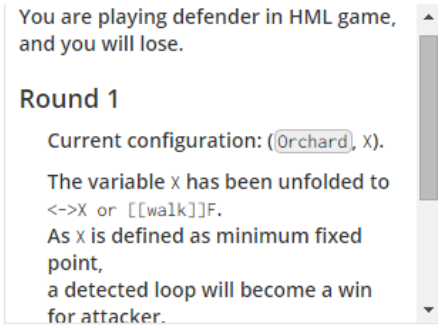
$$\texttt{X min= [[walk]]ff or <->X} \ .$$

The initial game log can be seen in Figure 14a. As it can be seen, we are playing as defender and we are going to lose, matching the claim from Figure 3 that the formula is not satisfied.

As defender we have to choose which of the disjunctions we want to continue from. We can pick between two subformulas:

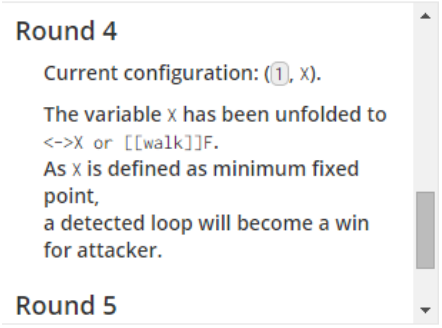
1. `[[walk]]ff` and
2. `<->X` .

Taking case 1 will result in a loss in the next round because the attacker picks a transition so we reach a false formula as shown in Figure 14b. Instead the defender picks case 2 and picks a transition, resulting in `X` being unfolded as it can be seen on Figure 14c. The game continues with the defender picking one of the two cases, each time unfolding `X` and having to pick a transition as seen on Figure 14d. Figure 15 shows the transition table for the defender. Eventually the game will detect a cycle as it can be seen in Figure 14e, which means the defender loses because we played in a minimum fix-point game.

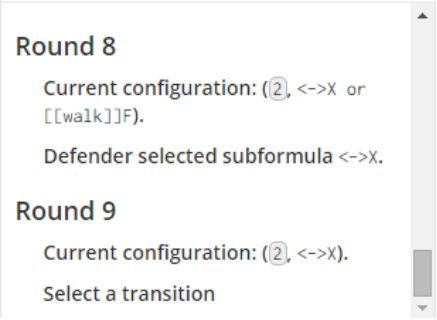


(a) Game log with introduction.

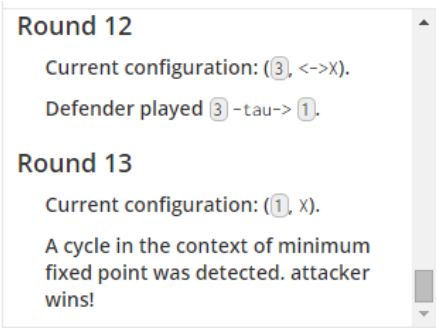
(b) Reaching false formula.



(c) Unfold X.



(d) Select a transition.



(e) Game log with a winner.

Figure 14: Game log for HML Game.

Source	Action	Target
2	walk	3

Figure 15: Transition table.

5.4 Distinguishing Formula

HML formulas can also be used to check if two processes are strongly bisimilar. Two processes are strongly bisimilar if and only if they satisfy the same formulas. This also implies that if two processes are not strongly bisimilar, then there must exist a formula that distinguishes them.

Example 5

We have the processes

$$\begin{aligned} \texttt{Man} &= \overline{\texttt{'shake}}.(\texttt{redapple.walk.Man + greenapple.walk.Man}) \ , \\ \texttt{FastMan} &= \overline{\texttt{'shake}}.(\texttt{redapple.}(\texttt{walk.FastMan + FastMan}) \ + \\ &\qquad \texttt{greenapple.}(\texttt{walk.FastMan + FastMan}) \ . \end{aligned}$$

CAAL is able to generate a distinguishing formula for two processes. Figure 16 shows a generated distinguishing formula for the two processes Man and FastMan which are not strongly bisimilar. This is done by clicking the three vertical dots on the right-hand side and clicking on 'Distinguishing formula'.

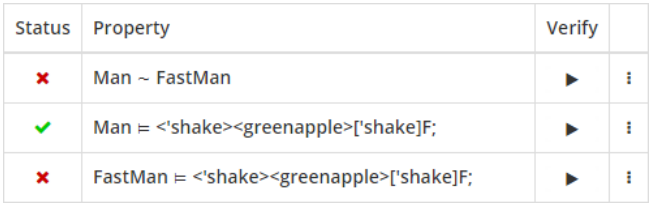


Figure 16: Distinguishing formula.

5.5 Distinguishing Trace

Much like the distinguishing formula, CAAL can also generate a trace that distinguishes two processes. When checking whether two processes are trace equivalent or if one process is a trace inclusion of the other, CAAL will output the distinguishing trace if this is not the case. Figure 17 shows a distinguishing trace for the processes Man and FastMan from Example 5.

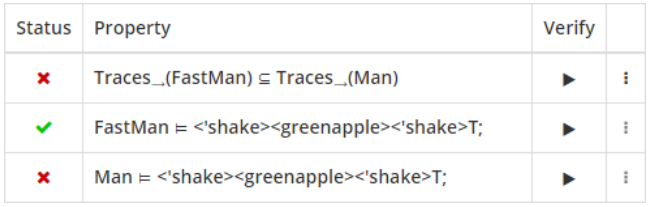


Figure 17: Distinguishing trace.

As we can see, the FastMan affords the trace

$$\overline{\text{'shake}}.\text{greenapple}.\overline{\text{'shake}}\ ,$$

which the Man does not. The distinguishing trace is given as an HML formula so that the HML game can be loaded.

6 Closing Remarks

CAAL is an open source project developed at Aalborg University by Jacob Karstensen Wortmann, Jesper Riemer Andersen, Nicklas Andersen, Mathias Munk Hansen, Simon Reedtz Olesen, and Søren Enevoldsen under the supervision of Jiří Srba and Kim Guldstrand Larsen.

The source code can be found on GitHub at `https://github.com/caal/caal`. We welcome suggestions and bug reports either through the issue tracker on GitHub, or via e-mail at caal@cs.aau.dk.