Appendix A Representation of control flow

In this appendix we describe the action language representation of the basic control flow elements in YAWL (figure A 1).

Fluents enabled\(_a\) are used to represent that a task named \(a\) is enabled, fluents en\_arc\(_a,b\) are used to represent that the arc from \(a\) to \(b\) is enabled. The rules are as follows:

**Sequence:**

\[
[a]\text{enabled}_a \\
[b]\neg \text{en\_arc}_a,b
\]
enabled_b ← en_arc_a_b
¬enabled_b ← ¬en_arc_a_b

AND-split:

[a]en_arc_a_b  [a]en_arc_a_c
[b]¬en_arc_a_b  [c]¬en_arc_a_c
enabled_b ← en_arc_a_b  ¬enabled_b ← ¬en_arc_a_b
enabled_c ← en_arc_a_c  ¬enabled_c ← ¬en_arc_a_c

AND-join:

[b]en_arc_b_a  [a]en_arc_a_c
[c]¬en_arc_b_c  [c]¬en_arc_a_c
enabled_c ← en_arc_b_c, en_arc_a_c  ¬enabled_c ← ¬en_arc_a_c

Nondeterministic XOR-split:

[a]en_arc_a_b  [a]en_arc_a_c
[b]¬en_arc_a_b  [b]¬en_arc_a_c
[c]¬en_arc_a_b  [c]¬en_arc_a_c
enabled_b ← en_arc_a_b  ¬enabled_b ← ¬en_arc_a_b
enabled_c ← en_arc_a_c  ¬enabled_c ← ¬en_arc_a_c

XOR-split:

Analogous to the nondeterministic one, except for the first two rules which become:

[a]en_arc_a_b ← Cond  [a]en_arc_a_c ← not Cond

XOR-join:

[b]en_arc_b_a  [e]en_arc_c_a
[a]¬en_arc_b_a  [a]¬en_arc_c_a
enabled_a ← en_arc_b_a  enabled_a ← en_arc_c_a
¬enabled_a ← ¬en_arc_b_a, ¬en_arc_c_a
Appendix B Experiments

In this appendix we report some experiments on the feasibility and scalability of the approach in the paper with current Constraint ASP technology. In particular, we run in clingcon (Gebser et al. 2009; Ostrowski and Schaub 2012) version 2.0.2 (and in some cases clingo, version 3.0.3), on a machine with Intel Xeon E5520 processors (2.26Ghz) and 32 GB RAM, an encoding with an optimization for the persistence of variable values: rather than propagating values of variables across tasks which do not set their value, a fluent lastchange(Var, S, S1) is propagated from S to the next state, representing that S1 is the last state where the value of Var was set; therefore, the value of a variable in a state is actually evaluated in the state where it last changed. This eliminates a number of equalities on constraint variables (e.g. value(pn, S') $ = value(pn, S)$ in section 5) that should be processed by the constraint solver.

**Experiment 1** is based on the process structure in figure B 1 as a basic block, which has a structure similar to the one of the running example; in particular, process models are as follows:

- The process structure is (see figure B2) a sequence of $r$ blocks with the structure in figure B 1 (i.e., for $i < r$, task 19 of the $i$-th block is followed by task set of the $i + 1$-th block). We use $l_i$ to denote the instance, in the $i$-th repetition, of a task or condition labeled $l$ in figure B 1.
- Conditions in different elements of the sequence vary; e.g., condition $c_{1i}$ ($c_1$ in block $i$) is different from $c_{1j}$ for $i \neq j$.

Different classes of process models are checked; each class has $r$ as a parameter and classes differ in branching conditions. Depending on the logical relations among such conditions, there can be up to $2^{4r}$ alternative runs of the process (since there are $2^4$ in a single block).

In problem class 1.1 we further have:

- The task set$_1$ sets two integer variables $v, v'$;

---

**Fig. B 1. Basic process model**

**Fig. B 2. Models for experiment 1; each block has the structure in figure B 1**
Table B 1. Running times for experiment 1

<table>
<thead>
<tr>
<th>Problem class</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 V</td>
<td>0.05</td>
<td>0.19</td>
<td>0.51</td>
<td>1.03</td>
<td>1.71</td>
<td>2.70</td>
<td>4.01</td>
<td>5.08</td>
<td>6.91</td>
<td>8.74</td>
</tr>
<tr>
<td>1.1 NV</td>
<td>0.05</td>
<td>0.20</td>
<td>0.51</td>
<td>1.03</td>
<td>1.73</td>
<td>2.83</td>
<td>4.17</td>
<td>6.10</td>
<td>8.13</td>
<td>10.53</td>
</tr>
<tr>
<td>1.2 V</td>
<td>0.05</td>
<td>0.23</td>
<td>0.61</td>
<td>1.29</td>
<td>2.38</td>
<td>4.14</td>
<td>8.12</td>
<td>12.98</td>
<td>21.57</td>
<td>25.61</td>
</tr>
<tr>
<td>1.2 NV</td>
<td>0.05</td>
<td>0.21</td>
<td>0.60</td>
<td>1.31</td>
<td>2.50</td>
<td>4.10</td>
<td>6.67</td>
<td>11.68</td>
<td>17.38</td>
<td>24.57</td>
</tr>
<tr>
<td>1.3 V</td>
<td>0.04</td>
<td>0.13</td>
<td>0.30</td>
<td>0.58</td>
<td>1.19</td>
<td>1.82</td>
<td>2.58</td>
<td>3.42</td>
<td>4.44</td>
<td>5.65</td>
</tr>
<tr>
<td>1.3 NV</td>
<td>0.04</td>
<td>0.13</td>
<td>0.31</td>
<td>0.59</td>
<td>1.13</td>
<td>1.75</td>
<td>2.38</td>
<td>3.25</td>
<td>4.13</td>
<td>5.18</td>
</tr>
<tr>
<td>1.4 V</td>
<td>0.03</td>
<td>0.16</td>
<td>0.41</td>
<td>0.87</td>
<td>1.26</td>
<td>2.06</td>
<td>2.36</td>
<td>4.60</td>
<td>6.79</td>
<td>11.02</td>
</tr>
<tr>
<td>1.4 NV</td>
<td>0.05</td>
<td>0.16</td>
<td>0.38</td>
<td>0.73</td>
<td>1.18</td>
<td>1.78</td>
<td>2.47</td>
<td>3.33</td>
<td>4.64</td>
<td>5.39</td>
</tr>
</tbody>
</table>

Fig. B 3. Running times compared for experiment 1

- Conditions $c_{1i}$ are $v > k_i$, where $k_i$, for $i = 1, \ldots, r$ are constants such that $k_i > k_{i+1}$.
- Conditions $c_{4i}$ are $v > k_i/2$.
- Conditions $c_{2i}$ and $c_{3i}$ are $v' < k'$ (so they are independent from the other conditions).
- Task $4_i$ has fluent $a_i$ as effect, task $5_i$ has effect $d_i$, task $14_i$ has effect $b_i$.

Note that $c_{1i}$ implies $c_{4i}$ which in turn implies $c_{1i+1}$. This makes branching quite constrained, in particular there are only $O(r)$ different runs.

The following formulae are checked for validity:
- $\Box(a_1 \rightarrow \Diamond b_r)$, which is valid, because $c_{1i}$ implies $c_{4r}$.
- $\Box(d_1 \rightarrow \Diamond b_r)$, which is not valid.

Table B 1, provides, in lines labeled 1.1 V (resp. 1.1 NV) the running times in seconds to verify the validity (resp., non-validity) of the two formulae for values of $r$ up to 10, using the length of the longest run as completeness threshold (for $r = 10$ it is 112 and it
requires 567 seconds to be computed, with more than half time spent to check that there are no runs of length 113.

In problem class 1.2 we have more complex dependencies among branching conditions; in particular, we have the following differences wrt 1.1:

- The task set \( i \) sets an integer variable \( v_i \);
- Conditions \( c_{1i} \) are \( v_i > k_i \);
- Conditions \( c_{2i}, c_{3i} \) are as in 1.1 (and \( v' \) is set by \( set_1 \));
- Conditions \( c_{4i} \) are \( v_i > k_i/2 \) for \( i \neq r \);
- Condition \( c_{4r} \) is \( \bigwedge_{i=1}^{r} v_i > k'_i/2 \) where \( k'_i < k_i \) (and then \( v_i > k_i \) implies \( v_i > k'_i \))

In this case, for \( i \neq r \), \( c_{1i} \) implies \( c_{4i} \), while \( \bigwedge_{i=1}^{r} c_{1i} \) implies \( c_{4r} \). The number of different runs is exponential in \( r \) \( (12^r) \).

The following formulae are checked for validity:

- \( \Box(\bigwedge_{i=1}^{r} a_i \rightarrow \Diamond b_i) \), which is valid, because \( \bigwedge_{i=1}^{r} c_{1i} \) implies \( c_{4r} \).
- \( \Box(\bigwedge_{i=1}^{r} d_i \rightarrow \Diamond b_i) \), which is not valid.

and running times are in lines 1.2 V and 1.2 NV of table B 1. The approach is sensitive to the way a branching condition is implied by other ones. For example, if in problem class 1.2 we change \( c_{4r} \) to:

\[ \sum_{i=1}^{r} v_i > \sum_{i=1}^{r} k_i \]

which is implied by \( \bigwedge_{i=1}^{r} c_{1i} \), as before, the verification is only feasible up to \( r = 3 \).

This behavior is apparently due to the solver integration and the constraint solver itself (running times increase significantly with the actual size of the numerical domain).

We then further measure the cost of relying, in general, on an integrated solver that calls a constraint solver for constraint atoms. In two further problem classes, in fact, we use branching conditions that do not involve constraint atoms, but conditions on variables with enumerated type; as stated in the paper, their translation does not involve constraint atoms but just ASP atoms. Therefore, the encoding can be run in clingo.

In particular, problem class 1.3 should be compared with 1.1, in the sense that the same logical relations hold between branching conditions, but in 1.1 the constraint solver is responsible for detecting such logical relations.

- \( set_1 \) sets \( r + 1 \) variables \( v_1, \ldots, v_{r+1} \) with domain \{false, true\};
- Conditions \( c_{1i} \) are \( \bigwedge_{j=i+2-1}^{r} v_j = \text{true} \).
- Conditions \( c_{2i}, c_{3i} \) are as in 1.1 (and \( v' \) is set by \( set_1 \)).
- Conditions \( c_{4i} \) are \( \bigwedge_{j=i+2}^{r} v_j = \text{true} \).

As in 1.1, \( c_{1i} \) implies \( c_{4i} \) which implies \( c_{1i+1} \). The formulae to be verified are as in 1.1 and running times are in lines 1.3 V and 1.3 NV of table B 1; a graphical comparison for 1.1 and 1.3 is in figure B 3 (left).

Then, in problem class 1.4 (to be compared with 1.2), we have
set_i sets three variables va_i, vb_i, vc_i with domain \{false, true\};

• Conditions c_1_i are va_i = true \land vb_i = true \land vc_i = true.
• Conditions c_2_i, c_3_i are as in 1.1 (and v' is set by set_1).
• Condition c_4_r is \bigwedge_{i=1}^{r} va_i = true and is implied by \bigwedge_{i=1}^{r} c_1_i.

Conditions are related as in 1.2. The formulae to be verified are as in 1.2 and results are in lines 1.4 V and 1.4 NV of table B1; a graphical comparison for 1.2 and 1.4 is in figure B3 (right).

The results show that the additional cost of relying on a constraint solver is acceptable for 1.1 and 1.2.

In experiment 2 we tested the case where r blocks, each with the structure in figure B1, are in parallel (using AND-split and AND-join), with a single set task that assigns variables before the split (see figure B4). Parallel execution means that all interleavings of every possible execution of each block should in general be considered (no reduction technique is introduced in our current approach), then the number of possible executions becomes extremely high even for low values of r.

In problem class 2.1, set assigns variables v_1, v_2 and for block i we have:

• Conditions c_1_i are v_1 > k_1.
• Conditions c_2_i and c_3_i are v_2 > k_2.
• Conditions c_4_i are v_1 > k_1/2.
• Task 4_i has fluent a_i as effect, task 5_i has effect d_i, task 14_i has effect b_i.

and the following formulae are checked for validity:

• \Box \bigwedge_{i=1}^{r} (a_i \rightarrow \diamond b_i), which is valid.
• \Box \bigwedge_{i=1}^{r} (d_i \rightarrow \diamond b_i), which is not valid.

In problem class 2.2, set assigns variables v_{1,i}, v_{2,i} for i = 1, \ldots, r and conditions in block i are as in 2.1, but on v_{1,i}, v_{2,i}.

![Fig. B4. Models for experiment 2; block are as in Figure B1 with no initial “set” task](image-url)
Table B 2. Number of different runs for process models in experiment 2

<table>
<thead>
<tr>
<th>Problem class</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 V</td>
<td>8.4·10^6</td>
<td>1.6·10^{15}</td>
<td>1.3·10^{25}</td>
<td>1.5·10^{36}</td>
</tr>
<tr>
<td>2.2 V</td>
<td>1.0·10^8</td>
<td>2.4·10^{17}</td>
<td>2.2·10^{28}</td>
<td>3.1·10^{40}</td>
</tr>
</tbody>
</table>

Table B 3. Running times for experiment 2

<table>
<thead>
<tr>
<th>Problem class</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 V</td>
<td>0.34</td>
<td>2.01</td>
<td>39.48</td>
<td>269.66</td>
</tr>
<tr>
<td>2.1 NV</td>
<td>0.23</td>
<td>0.63</td>
<td>72.01</td>
<td>94.24</td>
</tr>
<tr>
<td>2.2 V</td>
<td>0.28</td>
<td>2.12</td>
<td>22.79</td>
<td>1239.76</td>
</tr>
<tr>
<td>2.2 NV</td>
<td>0.26</td>
<td>0.86</td>
<td>17.46</td>
<td>11594.07</td>
</tr>
</tbody>
</table>

The number of different executions for \( r \) up to 5 is in table B 2.

In spite of such a large search space, verification is feasible (running times are in table B 3) if the length of the longest run is used as a bound. What is not feasible is computing such a bound with the approach used in other cases. Of course, if a process model is hierarchical, e.g. using composite tasks in YAWL, the bound (or an overestimate of it) can be computed by separately computing the longest activity sequence for component blocks (composite tasks).

In experiment 3 the process structure is fixed, the one in figure B 5, while branching conditions of increasing complexity are used, however each variable expression in constraint atoms involves a constant and low (3) number of variables. In this case running times do not explode.

In particular, we have the following conditions which depend on a parameter \( r \) and involve 3\( r \) variables set by activity 0:

- Condition c1 is \( \bigwedge_{i=1}^{r} v_{1,i} > k_1 \).
- Condition c2 is \( \bigwedge_{i=1}^{r} v_{2,i} > k_2 \).
- Condition c6 is \( \bigwedge_{i=1}^{r} v_{3,i} > k_3 \).
- Condition c12 is \( \bigwedge_{i=1}^{r} v_{1,i} + v_{2,i} + v_{3,i} > k_1 + k_2 + k_3 \).
- Task 10 has fluent \( d \) as effect, task 11 has effect \( b \), task 34 has effect \( a \).

The following formulae are checked for validity:
Fig. B5. Process model

<table>
<thead>
<tr>
<th>r</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid formula</td>
<td>0.81</td>
<td>1.61</td>
<td>2.46</td>
<td>3.28</td>
<td>4.01</td>
<td>4.82</td>
<td>5.68</td>
<td>6.59</td>
<td>7.63</td>
<td>7.63</td>
</tr>
<tr>
<td>non-valid formula</td>
<td>0.82</td>
<td>1.64</td>
<td>2.49</td>
<td>3.28</td>
<td>4.21</td>
<td>4.97</td>
<td>6.00</td>
<td>6.59</td>
<td>7.63</td>
<td>7.63</td>
</tr>
</tbody>
</table>

Table B4. Running times for experiment 3

- \(\Box(d \rightarrow \Diamond a)\), which is valid, because \(c_1 \land c_2 \land c_6\) implies \(c_{12}\).
- \(\Box(b \rightarrow \Diamond a)\), which is not valid.

Table B4 provides the results for this experiment.

As stated in the paper, completeness of BMC is a problem, since, while solutions for computing completeness thresholds for general formulae exist, as well as for formulae of a given form, the actual computation of such thresholds tends to be unfeasible.

In Experiment 4 we report some results on a variation, with a non-loop-free workflow, of the running example in the paper. The process is in figure B6: if the assessment
of the order is negative, then, nondeterministically (abstracting input from the customer), either the order is declined or the piece number is changed, restarting evaluation of the modified order. In a single execution, the state (in terms of fluents and constraint atoms) $S'$ reached after “Change pn” is in general different from the state $S$ reached after “Process order” (because of a different boolean value of constraint atoms such as $pn > 50000$), even though $S'$ is reachable (in a different run) from the initial state in the same number of steps as $S$.

For checking the validity of formulae $\square p$, where $p$ is propositional (or, more generally, $p$ can be evaluated in a single state), i.e. searching counterexamples satisfying $\neg \neg p$, the completeness threshold is (Biere et al. 2003), section 5.1) the number of steps to reach all states (reachability diameter, $rd$) which can be overapproximated by the length of the longest loop-free path starting from an initial state. In our case, the reachability diameter cannot be computed in a single answer set with the ASP encoding, since different executions correspond to different answer sets. The length of the longest loop-free path can indeed be computed, searching, for increasing values of $k$, for a simple run of length $k$, i.e., a run made by different states, where two states differ if they assign a different value to a fluent or a constraint atom in the process model or in the formula to be checked. Such a threshold can be used for checking formulae $\square p$, relaxing the requirement $\neg \neg p$ (e.g., checking later that an end state can be reached from the state satisfying $\neg \neg p$, if found); before knowing such a threshold, empirically “large” bounds can be used, possibly finding a counterexample in a time smaller than the one to compute the threshold (of course, validity is not guaranteed if no counterexample is found).

For the non-valid formula (3) in section 6, such a threshold (51) can be found in 108 seconds. Using it as the bound, a counterexample can be found in 0.32 seconds. Using 100, 200 or 300 as bounds, the counterexample can be found in 1, 3.1 and 6.7 seconds respectively.

For the valid formula (2) in section 6, the threshold (36) can be found in 20 seconds; using it, 0.95 seconds are needed to check the validity. In this case, using larger bounds is only feasible up to 60.
References

