Improving Symbolic Confluence Detection

Djurre van der Wal
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
djurrevanderwal@gmail.com

ABSTRACT

This paper shows that methods for symbolic confluence detection can be improved on three separate fronts. First, it is demonstrated that the degree of success of symbolic confluence detection can depend on the utilization of $\sum$ operator support. Second, a small amount of evidence is added to the notion of improving symbolic confluence detection by using an SMT prover instead of a BDD prover. Third, the paper presents a new confluence type called *triangular confluence* that seems to occur more frequently than commutative confluence. The conclusions of this paper are based on benchmarks of the symbolic confluence detection functionality of the mCRL2 toolset in which the improvements have been newly incorporated.

Keywords

Symbolic confluence detection, confluence conditions, state space reduction, formal specification language, mCRL2 toolset, process specifications, labeled transition systems, CVC4, theorem proving

1. INTRODUCTION

Validating and verifying a system can be achieved by exploring its state space: the collection of all possible states of that system and the transitions between those states. A common problem with state spaces is that they can become unmanageably large. This problem is called *state space explosion.*

One way to limit the size of a state space is by searching it for *confluence.* Confluence is a phenomenon where all paths within a state space that start at a certain state consist of the same visible transitions in the same order and can therefore be considered equivalent (see Figure 1). After the removal of duplicate paths from a state space, the state space is branching bisimilar to the original state space and therefore maintains most of its properties[1].

There are two main methods to accomplish state space reduction based on confluence: static confluence reduction, where a state space is analyzed that has already been generated; and on-the-fly confluence reduction, where confluence is removed during state space generation. In particular, when two paths within the state space are equivalent and one of them starts with a number of invisible τ transitions, these transitions can be *prioritized* during state space generation, meaning that they would be chosen over any alternative non-τ transitions. The states that follow after the non-τ transitions would disappear from the state space automatically [11].

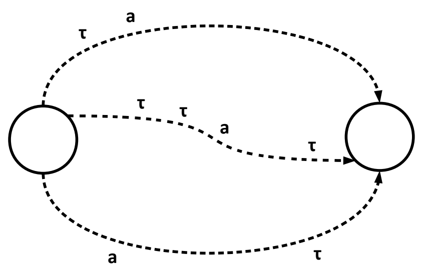

Figure 1. Equivalent paths

On-the-fly confluence reduction requires confluence information of a state space that can be acquired beforehand by analyzing the process from which the state space is generated. This is called *symbolic confluence detection.* In 2002, Blom and Van De Pol presented a symbolic confluence detection method based on the presence of strongly commutative transitions in a state space [3]. This method was implemented in the context of the μCRL (*micro* Common Representation Language) toolset for modelling, validating and verifying concurrent systems [6], but it has also been implemented in the context of μCRL's successor, mCRL2 [4] [7] [8] [9] [12] [13].

A peculiar shortcoming of the mCRL2 implementation is the inability to handle $\sum$ operators within process specifications. This shortcoming is peculiar because the feature *was* supported by μCRL. As a work-around, mCRL2 can be forced not to cluster summands with a $\sum$ operator during the compilation of a process specification and to rewrite a linear process that already contains $\sum$ operators. However, although this does not change the behavior of the process in question, it may very well affect the degree of success when applying confluence detection.

Another topic of this research is the method by which confluence within a state space is formally proved. mCRL2 achieves this by verifying the conditions for commutative confluence with a binary decision diagram (BDD) pro-

[1]All properties that can be expressed in action-based CTL*-X (computation tree logic without next-time) or in Hennessy-Milner logic.

22th *Twente Student Conference on IT* Jan 23st, 2015, Enschede, The Netherlands.

ver [10] [14]. In line with a paper about mCRL2 from 2013, which states that satisfiability modulo theory (SMT) provers are playing an increasingly important role in the mCRL2 toolset [4], this research investigates the effects of using an SMT prover instead. Because SMT provers are generally more suitable for evaluating expressions that involve arithmetics and induction-based types [5], it is expected that the SMT prover can confirm confluence where the BDD prover cannot. The chosen SMT prover is CVC4, a relatively new SMT prover with a good reputation [1].

Finally, the symbolic confluence detection method of Blom and Van De Pol is based on strong commutation, but there exist other confluence conditions that can be exploited [11]. This paper introduces and defines a new confluence type called *triangular confluence* and presents findings on how it performs relative to commutative confluence. Triangular and commutative confluence are also compared to *trivial confluence*, which reveals to what extent the described conditions actually occur within process specifications. The new confluence conditions have been added to mCRL2's functionality for symbolic confluence detection. In the future, even more confluence conditions may be discovered and made available in a similar fashion.

2. PRELIMINARIES

2.1 Commutative confluence detection

The method that Blom and Van De Pol presented in 2002 is based on the presence of strongly commutative transitions in a state space [3]. The method is applied to process specifications of the form

$$P(d) = \sum_{i \in I} \sum_{e_i} c_i(d, e_i) \to a_i(d, e_i).P(g_i(d, e_i)) \tag{1}$$

where

d is a vector of state variables;
I is the set of transition labels of process P;
e_i is the vector of local variables for transitions labeled i;
c_i is the enabling condition for transitions labeled i;
a_i is the action corresponding with transitions labeled i;
g_i is the next-state function for transitions labeled i.

As can be seen, a process specification consists of summands for all occuring transition labels. In turn, those summands are further organized by the local variables in e_i into smaller summands that each contain an enabling condition c_i which determines whether the process can go from its current state d to some state g_i via a specific transition a_i. Thus, summands can be identified by a transition label i and a vector of local variables e_i.

The confluence detection method of Blom and Van De Pol checks all summand pairs in the process specification that contain at least one τ summand for strong confluence (including summand pairs where both pairs are the same τ summand). If a τ summand only occurs in summand pairs that pass the check, that τ summand is marked as confluent and can be prioritized during subsequent state space exploration.

The check itself is performed by evaluating a *commutation formula*, the confluence condition for strong commutation. In this context, strong commutation entails that every outgoing transition pair (τ, a) of a state converges via the paths τ, a and a, τ (see Figure 2); if this is the case, the τ transitions are confluent. In order to determine whether a summand pair is strongly commutative, one can evaluate the commutation formula

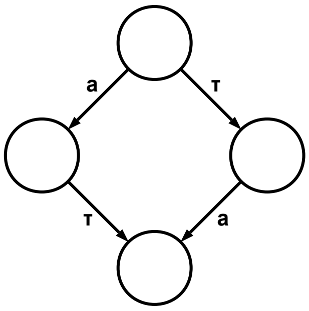

Figure 2. Strong commutation

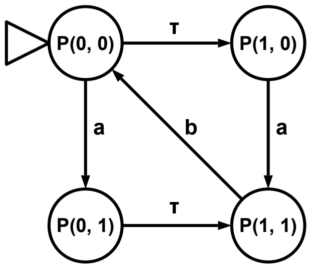

Figure 3. State space of Example 1

$$\begin{aligned} &\forall d, e_a, e_\tau \bullet c_a(d, e_a) \land c_\tau(d, e_\tau) \\ &\to \exists e_a{}', e_\tau{}' \bullet c_\tau(g_a(d, e_a), e_\tau{}') \land c_a(g_\tau(d, e_\tau), e_a{}') \\ &\quad \land a_a(d, e_a) = a_a(g_\tau(d, e_\tau), e_a{}') \\ &\quad \land a_\tau(d, e_\tau) = a_\tau(g_a(d, e_a), e_\tau{}') \\ &\quad \land g_a(g_\tau(d, e_\tau), e_a{}') = g_\tau(g_a(d, e_a), e_\tau{}') \end{aligned} \tag{2}$$

Note that $a_\tau(d, e_\tau) = a_\tau(g_a(d, e_a), e'_\tau)$ is trivially true and can be removed from the expression. However, the complexity of this remaining expression is still considerable. Blom and Van De Pol therefore strengthen the commutation formula by assuming that $e_a{}' = e_a \land e_\tau{}' = e_\tau$ to

$$\begin{aligned} &\forall d, e_a, e_\tau \bullet c_a(d, e_a) \land c_\tau(d, e_\tau) \\ &\to c_\tau(g_a(d, e_a), e_a) \land c_a(g_\tau(d, e_\tau), e_a) \\ &\quad \land a_a(d, e_a) = a_a(g_\tau(d, e_\tau), e_a) \\ &\quad \land g_a(g_\tau(d, e_\tau), e_a) = g_\tau(g_a(d, e_a), e_\tau) \end{aligned} \tag{3}$$

The assumption implies that both τ transitions must be generated by the same summand, and similar for both a transitions. Using the reduced commutation formula, which forms the basis for the confluence detection method of Blom and Van De Pol, a stricter type of confluence that frequently occurs in practice can be detected.

2.2 Examples

The very basic situation of Example 1 is shown in Figure 3. It shows a labeled transition system with 4 states and 4 transitions: 2 a transitions and 2 invisible τ transitions. The labeled transition system is generated from the following process specification:

$$\begin{aligned} P(x, y) = \ &(y = 0) \to a.P(x, 1) \\ &+ (x = 0) \to \tau.P(1, y) \\ &+ (x = 1 \land y = 1) \to b.P(0, 0) \end{aligned} \tag{4}$$

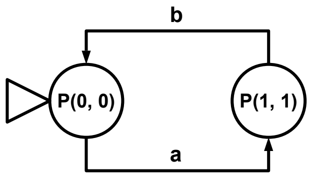

Figure 4. Reduced state space of Example 1

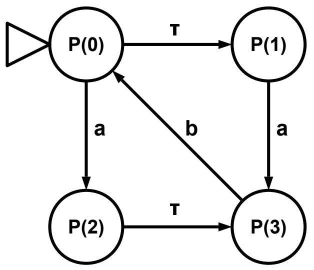

Figure 5. State space of Example 2

Given the process specification and the generic commutation formula for summand pairs, it is now possible to evaluate the commutation formula for the τ summand and the a summand:

$$\begin{aligned}&\forall x, y \bullet x = 0 \wedge y = 0 \\ &\rightarrow x = 0 \wedge y = 0 \wedge a = a \wedge (1,1) = (1,1)\end{aligned} \tag{5}$$

Since this expression is a tautology, it follows that the two summands are strongly commutative, and therefore all τ transitions that are generated by the τ summand must be confluent. By prioritizing those transitions, a new, reduced state space can be generated. The labeled transition system of this state space is displayed in Figure 4.

Instead of the linear process of Example 1, Example 2 uses the following linear process:

$$\begin{aligned}P(x) = \ & (x = 0) \rightarrow a.P(2) \\ + \ & (x = 0) \rightarrow \tau.P(1) \\ + \ & (x = 1) \rightarrow a.P(3) \\ + \ & (x = 2) \rightarrow \tau.P(3) \\ + \ & (x = 3) \rightarrow b.P(0)\end{aligned} \tag{6}$$

This process specification results in an equivalent labeled transition system, with the only difference being the variable vectors of the state (see Figure 5). However, now the commutation formula for the τ summand and the a summand is no longer a tautology, and for this reason the τ transitions can no longer be recognized as confluent:

$$\begin{aligned}&\forall x \bullet x = 0 \wedge x = 0 \\ &\rightarrow 2 = 0 \wedge 1 = 0 \wedge a = a \wedge 2 = 1\end{aligned} \tag{7}$$

Examples 1 and 2 demonstrate that describing the same process in different ways can result in different degrees of success when checking them for confluence.

2.3 Triangular confluence detection

The method of Blom and Van De Pol is based on strong commutation, but there are other confluence types that can be detected with algorithms of comparable complexity. One such scenario is the scenario with *confluent triangles.*

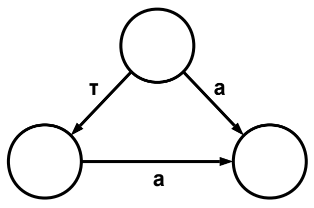

Figure 6. Triangular confluence

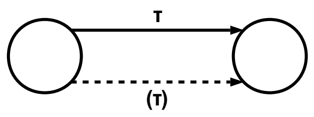

Figure 7. Trivial confluence

Figure 6 shows a confluent triangle. The confluence condition that can detect triangular confluence effectively checks if every outgoing transition pair (τ, a) of a state converges via the paths a and τ, a. The confluence condition for triangular confluence is

$$\begin{aligned}&\forall d, e_\tau, e_a \bullet c_\tau(d, e_\tau) \wedge c_a(d, e_a) \\ &\rightarrow c_a(g_\tau(d, e_\tau), e_a) \\ &\quad \wedge g_a(g_\tau(d, e_\tau), e_a) = g_a(d, e_a) \\ &\quad \wedge a_a(d, e_a) = a_a(g_\tau(d, e_\tau), e_a)\end{aligned} \tag{8}$$

If a τ summand only occurs in summand pairs that satisfy this condition, that τ summand is marked as confluent, precisely as prescribed by the detection method for commutative confluence.

2.4 Trivial confluence detection

Trivial confluence is another confluence type that occurs in situations where the outgoing transitions of a state are all τ transitions and all go to the same state (see Figure 7), which means that those τ transitions must be confluent by default. During this research, confluence is considered trivial when it can be detected with the confluence condition

$$\begin{aligned}&\forall d, e_\tau, e_a \bullet c_\tau(d, e_\tau) \wedge c_a(d, e_a) \\ &\rightarrow a = \tau \wedge g_a(d, e_a) = g_\tau(d, e_\tau)\end{aligned} \tag{9}$$

The fact that trivial confluence detection is embedded in mCRL2's functionality for symbolic confluence detection means that the frequency at which the tool detects trivial confluence rather than other confluence types is unknown. This uncertainty can be resolved by comparing the performance of a confluence condition with the performance of trivial confluence.

2.5 mCRL2

The mCRL2 toolset [4] [7] is a toolset for the modelling, validation and verification of concurrent systems. The associated formal specification language of the same name [8] [9] [12] [13] has been designed to restrict the expressive freedom of users as little as possible. mCRL2 is written in C++ and contains over 60 tools that can be used to visualize, simulate, minimize and model check complex systems. The toolset can be downloaded from `mcrl2.org` and used freely for any purpose under version 1.0 of the Boost Software License (boost.org).

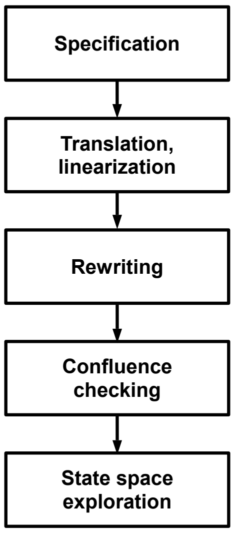

Figure 8. Confluence reduction pipeline in mCRL2

The code snippets below demonstrate how the mCRL2 language can be used to specify the processes specifications of Example 1 and Example 2:

```
% Example 1:
proc P(x, y: Int)
  = (y == 0) -> a . P(x, 1)
  + (x == 0) -> tau . P(1, y)
  + (x == 1 && y == 1) -> b . P(0, 0);
init P(0, 0);
```

```
% Example 2:
proc P(x: Int)
  = (x == 0) -> a . P(x = 2)
  + (x == 0) -> tau . P(x = 1)
  + (x == 1) -> a . P(x = 3)
  + (x == 2) -> tau . P(x = 3)
  + (x == 3) -> b . P(x = 0);
init P(0);
```

2.6 Confluence reduction in mCRL2

The pipeline for confluence reduction in mCRL2 is depicted in Figure 8 and can be described as follows. First, the user creates a specification of the process in question in the mCRL2 formal specification language. The process specification is translated into a binary format, which is then linearized and possibly rewritten. The output is a linear process.

Following the method of Blom and Van De Pol, commutation formulas are generated by the confluence checker for each summand pair in that linear process. An automated theorem prover is used to evaluate those commutation formulas. A τ transition is proved confluent if and only if all commutation formulas generated from it hold. Confluent τ transition are prioritized in the typical final step, namely generating a labeled transition system through exhaustive state space exploration.

The confluence detection functionality of the mCRL2 toolset is contained within the `lpsconfcheck` tool. When provided with a linear process, it iterates over all summand pairs that include a τ transition. That τ transition is marked confluent if all summand pairs are either disjoint or have a commutation formula that is confirmed to be a tautology. The latter is checked by mCRL2 binary decision diagram (BDD) prover [10] [14]. If both summands of a summand pair are τ summands, `lpsconfcheck` also allows either of them to be marked as confluent if they go to the same state (or set of states).

When desired, `lpsconfcheck` can check if a commutation formula that is not a tautology is, in fact, an invariant of the linear process. This is accomplished by checking whether

$$\forall d, e_a \bullet f(d) \wedge c_a(d, e_a) \rightarrow f(g_a(d, e_a)) \qquad (10)$$

where f is the (possibly rewritten) commutation formula. This is also checked by mCRL2's BDD prover.

2.7 CVC4

The SMT prover selected for this research is the relatively new CVC4 prover (`cvc4.cs.nyu.edu`). CVC4 is open-source and extensible theorem prover with a good reputation. 'CVC' stands for 'Cooperating Validity Checker', which refers to the cooperative nature of implemented decision procedures [1]. Compared to its predecessor, CVC3 [2], which is employed by mCRL2 for path elimination in binary decision diagrams, CVC4 is faster and has a lower memory usage, although it is still built around a boolean satisfiability (SAT) solver and a decision procedure.

3. RESEARCH

3.1 Research questions

This research strives to answer the following research questions:

1. How does $\sum$ operator support affect the performance of symbolic confluence detection?
2. How does replacing mCRL2's BDD prover with a CVC4 prover affect the performance of symbolic confluence detection?
3. How do symbolic detection of commutative confluence, triangular confluence, and trivial confluence perform relative to one another?

3.2 Approach

All research questions involve performance comparisons, each of which is achieved empirically by running a benchmark. The benchmark consists of a total of 76 process specifications that originate from the example and demonstration material that is part of the mCRL2 toolset. Only a minority of those process specifications (28 test cases) inherently contain τ transitions labels. Therefore, benchmark test cases instead consist of process specifications in which all transition labels are replaced by τ. This approach is used when analyzing a certain state space property to which the transition labels of the corresponding process specification are irrelevant, such as the presence of deadlocks. In case of the benchmarks, however, replacing all transition labels by τ simply maximizes the number of relevant summand pairs, which provides much more data; and since this part can make use of process specifications that do not inherently contain τ transition labels, the amount of data increases even further.

The benchmark results are visualized through scatter plots, whereby the performance of one method is plotted against the performance of another method. Markers above the $y = x$ line indicate better performance for the setting on the Y-axis, while markers below the $y = x$ line indicate better performance for the setting on the X-axis. The performance of one method, however, cannot always be compared directly against the performance of another method because they may derive new process specifications with different numbers of summands from the original process specification of a benchmark test case. Therefore, *effectiveness ratios* are calculated by dividing the detected number of confluent τ summands by the total number of τ summands in the process specification. This also allows measurements that are of different orders of magnitude to occur in the same scatter plot without distorting the scale.

In order to validate all newly written software, the 28 process specifications that inherently contain τ transition labels are also used for a verification benchmark. Each test case of the verification benchmark is analyzed by generating a state space both from the original and from the modified process specification and checking if they are branching bisimilar (mCRL2 has tools for this purpose).

With the techniques described above in place, answering the research questions from Section 3.1 comes within reach. Research question 1 is answered by adding $\sum$ operator support to mCRL2's `lpsconfcheck` and running a benchmark where its performance is measured against the original performance of mCRL2. When running a test case without $\sum$ operator support, the related process specification is prepared deliberately so that it does not contain $\sum$ operators (see Appendix C.1). When running a test case with $\sum$ operator support, no such action is taken.

The first step towards answering research question 2 is to manually confirm several summand pairs for which the CVC4 prover confirms the commutation formula as a tautology where the BDD prover of mCRL2 does not. The next step is to construct a CVC4 prover prototype that can be used by `lpsconfcheck` during a benchmark. The benchmark will determine if the updated prover capabilities are sufficient to significantly boost the efficiency of symbolic confluence detection. The benchmark is run both with and without $\sum$ operator support.

Research question 3 requires further modifications to `lpsconfcheck`. In particular, the confluence conditions for triangular and trivial confluence are added to the tool. However, `lpsconfcheck`'s technique to check whether two summands are disjoint before evaluating the corresponding commutative confluence condition (see Section 2.6) cannot be used for triangular and trivial confluence because they are not generally implied by this property. The technique must therefore be disabled when searching for those confluence types.

With even more future confluence types that could be added in the future in mind, the `lpsconfcheck` is fitted with a new parameter that allows the user to select a sequence of confluence conditions that must be checked by the tool. The benchmarks for answering research question 3 are run with exactly one of the confluence conditions enabled, and both with and without $\sum$ operator support.

4. RESULTS

4.1 Adding $\sum$ operator support

For the first part of the research, $\sum$ operator support was added to mCRL2's `lpsconfcheck`. It modifies the tool such that when two summands are compared, any summation variables shared by both summands are given a new, unique name in the second summand. For example, before the summand

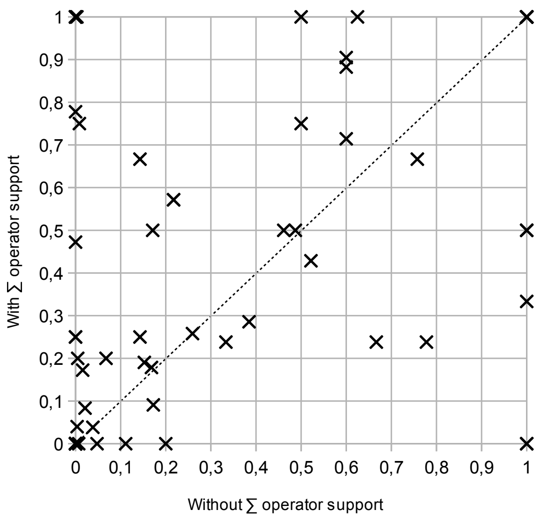

Figure 9. Performance with $\sum$ operator support relative to the performance without it

$$\sum_{i \in I,\, x \in X} c_1(d, i, x) \to a.P(g_1(d, i, x)) \tag{11}$$

is compared to the summand

$$\sum_{j \in J,\, x \in X} c_2(d, j, x) \to b.P(g_2(d, j, x)) \tag{12}$$

`lpsconfcheck` will rename the summation variable x in the latter summand.

The benchmark that tested the effect of adding $\sum$ operator support produced data that can be found in Appendix C.2. In 49 of 80 test cases, the effectiveness ratio without $\sum$ operator support differed from the effectiveness ratio with $\sum$ operator support. The benchmark data of all 80 test cases has been used to create the scatter plot that can be found in Figure 9. In order to repeat the benchmark, follow the instructions in Appendix C.1.

4.2 Comparing provers

During manual comparisons of the BDD prover and the CVC4 prover, a number of commutation formulas were properly evaluated by the CVC4 prover as tautologies, which the BDD prover of mCRL2 was unable to confirm. Therefore, the `lpsconfcheck` tool was extended with an instance of the newly created `Cvc_Prover` class. This class rewrites all declarations of the process specification in question to the CVC4 language. This is a necessary step because the commutation formula may make use of those declarations. The commutation formula itself is also converted to the CVC4 language and added to the rewritten declarations as a query, after which the result is written to a file. The `Cvc_Prover` instance calls CVC4 with the file as an argument. After CVC4 has interpreted and evaluated the expression, it returns the outcome of the query to its standard output stream. The outcome is captured by the `Cvc_Prover` instance, which passes the outcome to `lpsconfcheck`.

The `Cvc_Prover` class translates from mCRL2 to CVC4 by recursively traversing the tree-like structure in which the binary data of a process specification is organized in mCRL2 and converting the encountered elements to text that is compliant with CVC4 syntax. The approach is heavily based on mCRL2's `lpspp` tool, which also converts the binary data of a process specification to text.

Similarities between mCRL2 and CVC4 syntax make many translations by the `Cvc_Prover` class trivial. For example, the constants `true` and `false` in mCRL2 are represented by `TRUE` and `FALSE`, respectively, in CVC4. Similarly, several arithmetic and logical operators are converted to their counterparts in the other language. Other translations require greater effort. Appendix B describes the translations for quantifiers, the conditional operator, enumeration types, and list operators in brief.

Ultimately, however, the `Cvc_Prover` class does not provide translations to CVC4 for all features of mCRL2. Below an overview of supported features is given:

- Basic integer operators
- Conditional operator
- min, max, abs, div, mod
- List operators
- Booleans, integers, reals
- Logical operators and quantifiers
- User-specified mappings and equations
- Enumeration sorts
- Structured sorts

Several features that are *not* supported are:

- Positive numbers, natural numbers
- exp, abs, floor, ceil, round
- Casts between numerical sorts
- Lambda abstraction
- Function updates
- Sets, bags, and the corresponding operators
- Projection functions in structured sorts
- Structured literals

The benchmark that tested the effect of the CVC4 prover prototype produced positive results in seven cases. In two other cases, the same results can be achieved by letting CVC4 assist the BDD prover through path elimination. The data of these test cases can be found in Appendix C.3. Unsupported features were encountered in 23 cases; these cases could therefore not be benchmarked. It should also mentioned that the number of cases in which the benchmark yielded different results for different settings (seven out of 53) was considered too small to merit a scatter plot.

4.3 Comparing confluence conditions

mCRL2's `lpsconfcheck` has been modified and benchmarked as described in Section 3.2. See Appendix A for more information on the new usage of `lpsconfcheck`.

The benchmark for comparing commutative, triangular and trivial confluence produced 40 cases in which the results for commutative and triangular confluence were different. The data for these cases can be found in Appendix C.4. Test cases were also included in this data set when trivial confluence was present and yielded different results than either commutative or triangular confluence.

The data has been visualized with three separate scatter graphs: one for comparing commutative and trivial confluence (see Figure 10), one for comparing triangular and trivial confluence (see Figure 11), and one for comparing commutative and triangular confluence (see Figure 12). In Figure 10, 54 out of 152 markers (two per test case) are located above the $y = x$ line. This value is 72 for Figure 11. Figure 12 contains 34 markers where triangular confluence outperforms commutative confluence; the reverse is true in five cases.

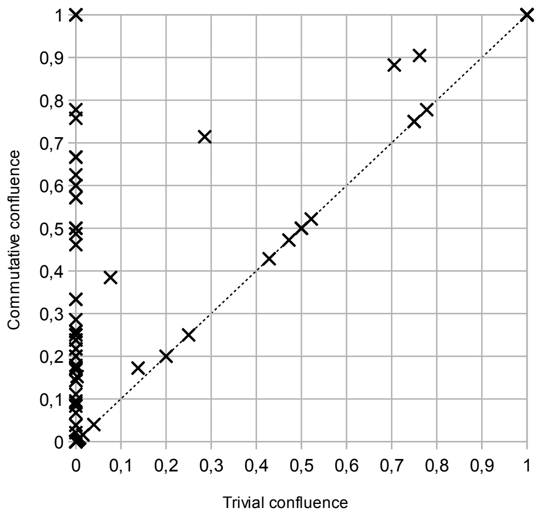

Figure 10. Performance of commutative confluence detection relative to trivial confluence detection

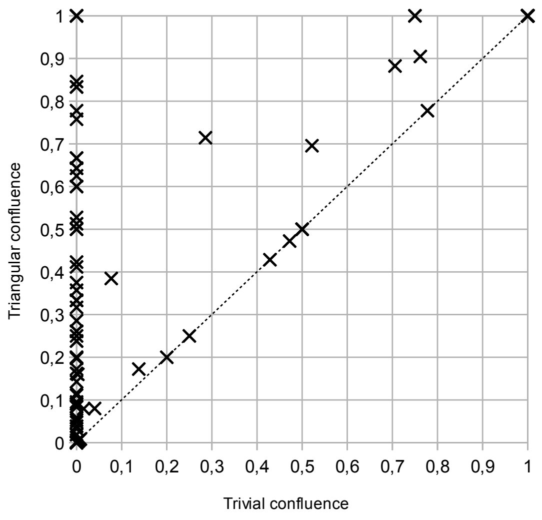

Figure 11. Performance of triangular confluence detection relative to trivial confluence detection

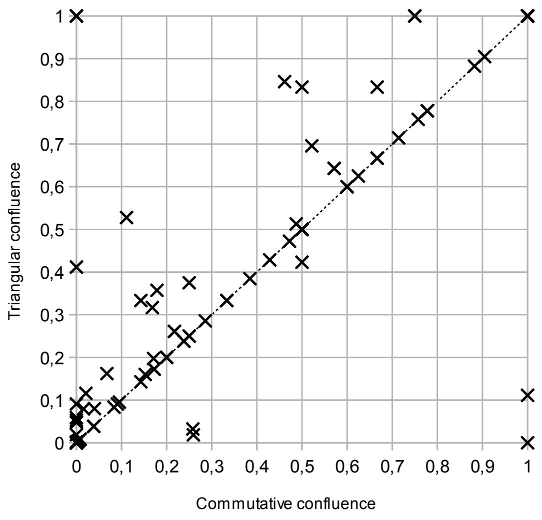

Figure 12. Performance of triangular confluence detection relative to the performance of triangular confluence detection

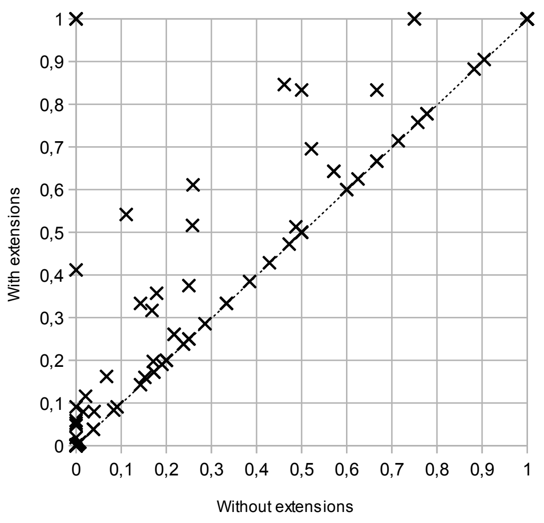

Figure 13. Performance with the extensions developed during this research relative to the performance without them

4.4 Overall improvement

The overall improvement of symbolic confluence detection through the extensions developed for this research has also been measured by running an extra benchmark. The result is displayed in the scatter plot of Figure 13, in which the performance of mCRL2 with the extensions is plotted against the performance of mCRL2 without them. In this scatter plot, 34 out of 152 markers are located above the $y = x$ line.

5. DISCUSSION

When considering the value of this research, it should be noted that the research relies heavily on benchmarks based on the process specifications that are part of the mCRL2 toolset as example and demonstration material and that these are not necessarily representative. Additionally, test cases were not tailored for symbolic confluence detection in preparation of the benchmarks. mCRL2 provides multiple tools that, when applied individually or when applied in sequence, may produce a wide range of different specifications that all describe the same process, and some specifications may be more suitable for symbolic confluence detection that others. Nevertheless, all test cases were prepared consistently in manners that were considered common. Appendix C.1 elaborates on this topic.

The fact that the scatter plots from the previous section use effectiveness ratios (fractions of the total number of τ summands) means that the distance of a marker to the $y = x$ line is generally not proportional to the number of τ summands that was marked as confluent. For example, if one τ summand is marked as confluent of a process specification with only one τ summand in total, the corresponding marker is located much further from the $y = x$ line than the marker of a process specification of which one out of several τ summands has been marked as confluent.

Also note that `lpsconfcheck`'s timeout value for the evaluation of a single confluence condition has been set to 8 seconds for all benchmarks in order to reduce computation time. This means that mCRL2's BDD prover and the CVC4 prover prototype had that amount of time to confirm whether an expression is a tautology or not, and it is possible that this has affected the results.

The scatter plot in Figure 9 shows that $\sum$ operator support can contribute positively (in 31 test cases), neutrally (in 31 other test cases) and negatively (in 18 test cases) to symbolic confluence detection. This observation is explained by the fact that, with $\sum$ operator support enabled, mCRL2 is likely to organize the summands of a process specification differently. Most commonly, summands with the same transition labels are clustered into a single summand. During symbolic confluence detection, this changes the corresponding confluence conditions, and thus influences whether τ summands can be proved to be confluent or not.

6. CONCLUSIONS

The data from the benchmarks provides answers to the research questions from Section 3.1.

In answer to research question 1, the scatter plot in Figure 9 clearly shows that existing symbolic confluence detection can benefit significantly from utilizing $\sum$ operator support, and it is therefore safe to conclude that it improves the symbolic confluence detection functionality of mCRL2. However, since it can be equally beneficial to avoid $\sum$ operator support, one should apply the method only on a case-by-case basis.

With regard to the performance comparison of confluence detection with the CVC4 prover prototype and confluence detection with mCRL2's BDD prover, the benchmark data indicates that the CVC4 prover has some advantages, at least where test cases could be translated to the CVC4 language. This is demonstrated in a small number of test cases (seven out of 47).

Research question 3 concerned the relative performance of commutative, triangular, and trivial confluence. Figure 10 and Figure 11 show that commutative and triangular confluence are both likely to detect confluent τ summands that are not trivially confluent. When comparing commutative and triangular confluence to each other, triangular confluence has a higher frequency of success, although there is still a significant number of cases in which detection of commutative confluence is superior (see Figure 12). Triangular confluence is therefore considered to be a valuable addition to symbolic confluence detection.

7. FUTURE WORK

The results of this research suggest that there is potential in improving symbolic confluence detection by expanding the range of available confluence conditions even further. Trivial and triangular confluence are confluence types with a lower complexity than commutative confluence (meaning that they involve more states and transitions), and investigating increasingly more complex confluence types is probably the most methodical approach towards exploiting this aspect of symbolic confluence detection.

Based on the answer to research question 2, it seems warranted that mCRL2 makes more and more use of SMT provers in general, and it would be sensible to include mCRL2's functionality for symbolic confluence detection in this development. This also assumes that CVC4 is representative for SMT provers in general and that using other SMT provers will have similar results. In order to test this assumption, one could run comparative benchmarks with other SMT provers. An efficient way to approach this is by constructing a tool that converts mCRL2 expressions to SMT-LIB (an intermediary language supported by multiple SMT provers) rather than to the language of a specific prover. Preferably, this tool should be more feature-complete than the CVC4 prover prototype used for this research.

With regard to the CVC4 prover prototype, it should be added that its current design is quite inefficient and can benefit greatly from optimization. Currently, instead of making direct use of CVC4 via the available API, the prototype converts mCRL2 binary data to text, saving it to a file, and running CVC4 as a separate process, which creates additional overhead for the operating system and requires CVC4 to perform a translation from the CVC4 language to CVC4 binary data. There are two reasons for this design. First, converting mCRL2 binary data to text allows for easier debugging and experimentation. It was also easier to implement because mCRL2's functionality for printing binary data could simply be modified. Second, the text is saved to files because there were difficulties establishing a reliable pipeline with CVC4. It is unknown whether these difficulties are at the side of the prototype or at the side of CVC4.

8. ACKNOWLEDGEMENTS

Thanks to *Prof. dr.* J.C. van de Pol for guidance and advice throughout this research project, to *Prof. dr. ir.* J.F. Groote for information on the current developments in the mCRL2 toolset, and to *Dr. ir.* J.W. Wesselink for help with the development of new mCRL2 software.

9. REFERENCES

[1] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[2] C. Barrett and C. Tinelli. Cvc3. In *Computer Aided Verification*, pages 298–302. Springer, 2007.

[3] S. Blom and J. van de Pol. State space reduction by proving confluence. In *Computer Aided Verification*, pages 596–609. Springer, 2002.

[4] S. Cranen, J. F. Groote, J. J. Keiren, F. P. Stappers, E. P. de Vink, W. Wesselink, and T. A. Willemse. An overview of the mCRL2 toolset and its recent advances. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213. Springer, 2013.

[5] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[6] M. V. Espada and J. Van De Pol. An abstract interpretation toolkit for μCRL. *Formal Methods in System Design*, 30(3):249–273, 2007.

[7] J. F. Groote, J. Keiren, F. P. Stappers, J. Wesselink, and T. A. Willemse. Experiences in developing the mCRL2 toolset. *Software: Practice and Experience*, 41(2):143–153, 2011.

[8] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. Van Weerdenburg. *The formal specification language mCRL2*. Internat. Begegnungs-und Forschungszentrum für Informatik, 2007.

[9] J. F. Groote and M. Reniers. Modelling and analysis of communicating systems. *Technical University of Eindhoven, rev*, 1478:15–18, 2009.

[10] J. F. Groote and J. van de Pol. Equational binary decision diagrams. In *Mathematical Foundations of Computer Science 2000*, pages 161–178. Springer, 2000.

[11] J. F. Groote and J. van de Pol. State space reduction using partial τ-confluence. In *Mathematical Foundations of Computer Science 2000*, pages 383–393. Springer, 2000.

[12] J. J. Keiren and M. A. Reniers. Type checking mCRL2. *Computer Science Report*, 11(11), 2011.

[13] A. Mathijssen. Data types for mCRL2. *This article is available in the mCRL2 Subversion repository*, 2014.

[14] J. van de Pol. A prover for the μCRL toolset with applications: Version 0.1. *Report-Software engineering*, 6:1–32, 2001.

APPENDIX

A. NEW OR EXTENDED TOOLS

In the course of the research, the following mCRL2 tools were created or extended:

`ctau_marker` — This tool marks specific τ (or `tau`) transitions in a linear process as confluent (or `ctau`). It can also be set to number `ctaus`, allowing `ctau` transitions to be traced back to their summand.

`lps2cvc` — This tool translates the declarative part of the mCRL2 specification of a linear process to the CVC4 language. It also generates commutation formulas and translates those as well. The output is a readable file using CVC4 syntax.

`lpsconfcheck` — This tool has been expanded with a number of arguments. First, `-u` is used to disable $\sum$ operator support (it is enabled by default). Second, `-e` forces `lpsconfcheck` to use the `Cvc_Prover` class. For this setting to work properly, CVC4 must be in the system path. Finally, `-xCONDITIONS` selects the confluence conditions that `lpsconfcheck` should use: `c` and `C` select commutative confluence with and without checking whether a summand pair is disjoint, respectively (see Section 2.6 and 3.2), `T` selects triangular confluence, and `Z` selects trivial confluence. By default, `lpsconfcheck` searches for commutative confluence only (`-xc`).

`lpsunlabel` — The purpose of this tool was the creation of more test cases for the benchmarks of this research. This is achieved by removing all translation labels from a linear process, effectively converting all transitions into τ transitions. It does not require translation rules such as the mCRL2 tool `lpsactionrename` and can therefore be applied much easier.

B. TRANSLATIONS

This section gives an overview of several implemented translations from mCRL2 to CVC4.

B.1 Quantifiers

Quantifiers use a different syntax in CVC4:

- `forall v . c` becomes `FORALL(v): c`.
- `exists v . c` becomes `EXISTS(v): c`.

B.2 Conditional operator

The conditional operator is implemented in mCRL2 as a mapping:

```
map
  if: Bool # Type # Type -> Type;
eqn
  if(true, a, b) = a;
  if(false, a, b) = b;
```

However, CVC4 requires this to be converted to a hard-coded structure of the form `IF c THEN a ELSE b ENDIF`.

B.3 Enumeration types

Enumeration types are declared differently by mCRL2 and CVC4, and the `Cvc_Prover` class may be required to generate recognizer functions separately. An example of how enumeration types are defined in mCRL2 is given below:

```
sort
  Bit = struct bit0 | bit1;
  Enum = struct e1 ? is_first | e2 | e3;
```

This is translated to CVC4 as

```
DATATYPE
  Bit = bit0 | bit1,
  Enum = e1 | e2 | e3
END;

is_first: (Enum) -> BOOLEAN;
ASSERT is_first(e1) = TRUE;
ASSERT is_first(e2) = FALSE;
ASSERT is_first(e3) = FALSE;
```

B.4 List operators

mCRL2 supports a number of container types, including lists, and operators to manipulate these lists. In order to be able to evaluate expressions that involve lists, these types must be manually defined in CVC4. Preferably, this would be accomplished through parametric data types; however, since functions in CVC4 do not accept parametric data types as parameter types, inductive data types must be used instead. For example, the type of a list that contains elements of type `Elem` can be created as follows:

```
DATATYPE
  List = cons (head: Elem, tail: List)
END;
```

List operators can subsequently be defined as functions. For example, the size of a list of type `List` can be obtained with the help of

```
List_get_size: (List) -> Int;
ASSERT FORALL(x: List):
  List_get_size(x) =
    IF x = nil THEN 0
    ELSE List_get_size(tail(x)) + 1
    ENDIF;
```

C. BENCHMARKS

This appendix contains the essential data gathered from the benchmarks during the research. Note that, for brevity, only data of test cases that performed differently for different settings is included. The first section describes how the benchmark data can be reproduced.

C.1 Reproducing data

In order to reproduce the data with $\sum$ operator support enabled, run an mCRL2 source file with

```
mcrl22lps file.mcrl2 file.lps
lpsunlabel file.lps file.lps
lpsconfcheck -xc -t8 file.lps file.xc.lps
```

Adjust the argument `-xc` of the `lpsconfcheck` command to search for other confluence types than commutative confluence. The argument `-t8` forces `lpsconfcheck` to not spend more than 8 seconds on the evaluation of a single confluence condition; this setting was used in the course of this research in order to reduce computation times.

Running the same mCRL2 source file with $\sum$ operator support enabled can be achieved similarly with

```
mcrl22lps -n file.mcrl2 file.lps
lpssuminst file.lps file.lps
lpsunlabel file.lps file.lps
lpsconfcheck -u -xc -t8 file.lps file.xc.lps
```

In some cases, the `lpssuminst` command takes a very long time to complete. It was disabled for the benchmarks of this research for the following mCRL2 source files:

File
11073.mcrl2
bke.mcrl2
garage.mcrl2
gpa_10_1.mcrl2
gpa_10_2.mcrl2
gpa_10_3.mcrl2
small1.mcrl2
swp_lists.mcrl2

C.2 Adding $\sum$ operator support

The data from the benchmarking during which the performance of `lpsconfcheck` was compared can be found in Table 2 on page 10.

C.3 Comparing provers

The data produced by the benchmark that compared the performance of `lpsconfcheck` with the BDD prover to its performance with the CVC4 prover prototype can be found in Table 1 at the top of this page.

C.4 Comparing confluence conditions

Table 3 on page 10 contains the essential data from the benchmark during which the three different confluence conditions were compared with each other.

Table 1. Comparing different provers (benchmark data)

File	*N*	*B*	*P*	*C*	*S*
cabp.mcrl2	21	3	9	9	No
dining3_cs_seq.mcrl2	21	5	8	15	Yes
dining3_schedule_seq.mcrl2	21	5	5	21	Yes
dining3_schedule.mcrl2	21	5	5	21	Yes
dining3_seq.mcrl2	21	5	7	15	Yes
leader.mcrl2	29	5	5	29	No
par.mcrl2	23	5	9	9	No
queue.mcrl2	4	1	1	4	Yes
queue.mcrl2	7	1	1	7	No

N: Number of summands in the process specification.

B, P, C: Number of summands in the process specification that were marked as confluent when using mCRL2's BDD prover, mCRL2's BDD aided by the CVC4 prover prototype for path elimination, and the CVC4 prover prototype on its own, respectively.

S: Indicates whether or not $\sum$ operator support was enabled ('Yes' means that $\sum$ operator support was enabled; 'No' means that it was disabled).

Table 2. Analyzing the influence of utilizing $\sum$ operator support (benchmark data)

File	*N1*	*D1*	*N2*	*D2*
11073.mcrl2	290	1	25	1
1394-fin.mcrl2	1069	163	21	4
abp_bw.mcrl2	33	25	12	8
abp.mcrl2	16	10	10	10
alma.mcrl2	78	38	52	26
ball_game.mcrl2	1	1	3	0
bke.mcrl2	63	1	29	5
block.mcrl2	20	0	90	70
brp.mcrl2	76	13	18	9
cabp.mcrl2	21	3	12	8
chatbox.mcrl2	72	8	3	0
dining_10.mcrl2	50	30	210	190
dining3_cs_seq.mcrl2	27	18	21	5
dining3_ns_seq.mcrl2	15	9	21	15
dining3_ns.mcrl2	143	0	271	128
dining3_schedule_seq.mcrl2	27	21	21	5
dining3_schedule.mcrl2	27	21	21	5
dining3_seq.mcrl2	27	9	21	5
dining8.mcrl2	40	24	136	120
dkr1.mcrl2	36	36	12	4
domineering.mcrl2	42	2	4	0
exists.mcrl2	4	4	1	1
food_package.mcrl2	822	4	5	1
forall.mcrl2	4	4	1	1
garage-r1.mcrl2	472	4	8	6
garage.mcrl2	8	4	8	6
goback.mcrl2	1000	0	1	1
gpa_10_3.mcrl2	1	0	1	1
knights.mcrl2	329	1	2	2
lambda.mcrl2	4	4	1	1
leader.mcrl2	29	5	55	5
lift3-final.mcrl2	285	6	36	3
lift3-init.mcrl2	222	15	30	6
light.mcrl2	2	2	4	2
list.mcrl2	3	0	1	1
magic_square.mcrl2	8	8	1	1
mpsu.mcrl2	23	12	14	6
numbers.mcrl2	59	59	4	4
par.mcrl2	23	5	14	8
peg_solitaire.mcrl2	141	1	5	0
queue.mcrl2	7	1	4	1
rational.mcrl2	12	12	6	3
scheduler.mcrl2	13	5	7	2
simple.mcrl2	1	1	3	0
small1.mcrl2	2	1	2	2
SMS.mcrl2	54	14	31	8
snake.mcrl2	10	2	4	0
swp_lists.mcrl2	11	0	8	2
trains.mcrl2	13	6	12	6
WMS.mcrl2	101	17	56	10
wolf_goat_cabbage.mcrl2	7	0	4	1

N1, *D1*: Number of summands in the process specification when avoiding $\sum$ operators, and the number of those summands that were marked as confluent, respectively.

N2, *D2*: Number of summands in the process specification when making use of the added $\sum$ operator support, and the number of those summands that were marked as confluent, respectively.

Table 3. Comparing different confluence conditions (benchmark data)

File	*N*	*C*	*T*	*Z*	*S*
11073.mcrl2	25	1	2	1	Yes
1394-fin.mcrl2	1069	163	171	3	No
alma.mcrl2	52	26	22	0	Yes
alma.mcrl2	78	38	40	0	No
bakery.mcrl2	34	0	14	0	No
bke.mcrl2	29	5	5	4	Yes
bke.mcrl2	63	1	5	1	No
brp.mcrl2	76	13	15	0	No
cabp.mcrl2	12	8	10	0	Yes
cabp.mcrl2	21	3	7	0	No
chatbox.mcrl2	72	8	38	0	No
clobber.mcrl2	4	0	4	0	Yes
clobber.mcrl2	106	0	2	0	No
dining_10.mcrl2	210	190	190	160	Yes
dining3_ns_seq.mcrl2	21	15	15	6	Yes
dining8.mcrl2	136	120	120	96	Yes
dkr1.mcrl2	36	36	4	0	No
domineering.mcrl2	4	0	4	0	Yes
domineering.mcrl2	42	0	2	0	No
fischer.mcrl2	16	0	16	0	Yes
garage-r1.mcrl2	8	6	8	6	Yes
garage-r2-error.mcrl2	466	0	1	0	No
garage-r2.mcrl2	466	0	1	0	No
garage-r3.mcrl2	466	0	1	0	No
garage-ver.mcrl2	1647	0	1	0	No
garage.mcrl2	8	6	8	6	Yes
hex.mcrl2	4	0	4	0	Yes
hex.mcrl2	34	0	2	0	No
lift3-final.mcrl2	285	6	33	0	No
lift3-init.mcrl2	222	15	36	0	No
light.mcrl2	2	2	0	0	No
mpsu.mcrl2	23	12	16	12	No
onebit.mcrl2	38	0	2	0	No
par.mcrl2	14	8	9	0	Yes
par.mcrl2	23	5	6	0	No
scheduler.mcrl2	13	5	5	1	No
SMS.mcrl2	31	8	1	0	Yes
SMS.mcrl2	54	14	1	0	No
snake.mcrl2	4	0	4	0	Yes
swp_lists.mcrl2	8	2	3	0	Yes
swp_lists.mcrl2	11	0	1	0	No
trains.mcrl2	12	6	10	0	Yes
trains.mcrl2	13	6	11	0	No
WMS.mcrl2	56	10	20	0	Yes
WMS.mcrl2	101	17	32	0	No

N: Total number of summands in the process specification.

C, *T*, *Z*: Number of summands in the process specification that were marked as confluent when searching for commutative, triangular, and trivial confluence, respectively.

S: Indicates whether or not $\sum$ operator support was enabled ('Yes' means that $\sum$ operator support was enabled; 'No' means that it was disabled).