Automatic Parallelization of Automata

S.H. van Schouwenburg
University of Twente
s.h.vanschouwenburg@student.utwente.nl

ABSTRACT

There exist many modeling languages for model checking, each with their own verification tools. There have been efforts to get some better interoperability among modeling languages, and LTSmin is one such attempt.
LTSmin is a toolset which contains tools to abstract models in a high level modeling language into a Labelled Transition System (LTS). This is done by doing a state space generation. That state space can be stored in ETF (Enumerated Table Format).
Since the model checking is then done on the state space instead of the original model, the question is how much information this state space contains compared to the original high level language. In this work, we investigate this by transforming high level models in DVE (Distributed Verification Environment) format into ETF and then convert that ETF back into a DVE model. We then perform benchmarks to see how much performance we have lost (or gained).
Model checking on DVE performs best on a model with parallel processes, as opposed to a monolithic single process model. So the transitions in the state space must be divided over separate parallel processes. Each process can only write to its own variables, but can read from all of them. So we need an algorithm which identifies variables in the state space that can have their own process. It must be done in such a way that no transition alters more than one of such variable, or that transition will not fit in any process.
One way of doing this is by using an independent set algorithm (ISA) to identify (mark) the variables. We have investigated whether a weighted independent set algorithm performs better than an unweighted algorithm.
We have benchmarked different models and compared the originals with the plain (non-parallel), weighted and unweighted generated models. The main conclusions are that generated models never approach the efficiency of the originals and that the parallel versions usually perform significantly better than the plain versions.

Keywords

Parallelization, Model Checking, DVE, ETF, LTSmin, Independent Set Algorithm

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
13th Twente Student Conference on IT June 21, 2010, Enschede, The Netherlands.
Copyright 200X, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

1. BACKGROUND

Model checking is one of the methods used nowadays to check the correctness of digital systems. There are a lot of different modeling languages, each with its own set of model checking tools. Examples of modeling languages include the process algebras like μCRL[9, 4] or state-based languages like PROMELA[5] or DVE[6, 1].

Each modeling language has their own way of representing a model, and each of them comes with their own set of model checking tools. This means that when you have modelled the system in a certain language, you are more or less bound by the tools for that language.

There have been attempts to make model checking tools interoperable, mainly based on language translations.[8, 10] Another approach is one that abstracts the model checking from the language, like what has been done by LTSmin.[7, 3]

1.1 DVE

The high level modeling language we have used in this paper is DVE (Distributed Verification Environment). A full description of the DVE modeling language is out of the scope of this paper, but we will present here the basic building blocks of a DVE model. A DVE model consists of global variables, several processes and optionally some channels. Each process has a list of symbolic state names, an initial state, transitions and optionally some local variables.

Transitions have a starting and ending state and any of the following:

1. A guard: a boolean expression, which, in addition to the starting state limit when the transition can be taken. Guards can read from local and global variables, and test whether a specific process is in a specific state.

2. An effect: one or more assignments on local and/or global variables.

3. A sync section: a channel end on which this transition will be synchronized.

Program 1 shows an example of a simple DVE model.

1.2 LTSmin

LTSmin is a toolset for model checking on labelled transition systems. As its input it has a labelled transition system (LTS) in a certain specific format (called the DIR format). It also has some tools to generate those labelled transition systems from other modeling languages, so the toolset becomes language independent. LTSmin does a reachability analysis from the LTS to generate the state

Program 1 Simple DVE example

```
process X {
  byte a,b;
  state x0, x1, x2, x3, x4;
  init x0;
  trans
    x0 -> x1 { effect a=0; },
    x0 -> x1 { effect a=1; },
    x1 -> x2 { effect b=0; },
    x1 -> x2 { effect b=1; },
    x2 -> x3 { guard a != b; },
    x2 -> x4 { guard a == b; },
    x3 -> x0 {},
    x4 -> x0 {};
}
system async;
```

space on which model checking can be done. This state space can then be exported in an Enumerated Table Format (ETF)[2], which is a plain text memory dump. The generated ETF of our example DVE model is shown in appendix B.

In the `state` section it enumerates all the variables in a `name:type` format. Then comes an `edge` section which is not used for DVE models. After that an `init` section containing the initial values of the variables. Then come the most important `trans` sections, which define all the transitions. They contain lines like the following:

```
* 1/1 3/5 *
```

This represents a transition from any state where the second variable is 1, the third variable is 3 and the first and fourth variables don't matter, to a state where the third variable becomes 5 and all the rest stays the same.[1] A variable in a transition is either completely ignored, or fully specified. There cannot be a transition like */2 (i.e.: It does not matter what variable 3 was before, but after this transition it is 2). If, for example, the variable can take on 5 different values (0-4) then there would be 5 lines: one for each source value (e.g.: 0/2, 1/2, 2/2, 3/2, 5/2).

The transitions are grouped in sections, and within a section the variables that are ignored and those that are specified are always the same. These groupings can play a large role in performance for LTSmin utilities. This transition grouping is also one of the identifying features of LTSmin and is explained in more detail in *Bridging the Gap between Enumerative and Symbolic Model Checkers*.[7]

Finally there are some `sort` sections. They provide a translation for symbolic variable types from values to symbols. They can be seen as 0-based arrays. In DVE, the only symbolic types are the states, with the symbols being the state names.

The question is how effective this state space representation is compared to the higher level representation. Since this representation is much more basic than the original high level model we might have lost some information.

To check this, we can try to convert the generated ETF back into the higher level language (see figure 1), and compare model checking on the generated model with model checking on the original model.

[1] You could say the second variables "becomes" 1, but it already is 1

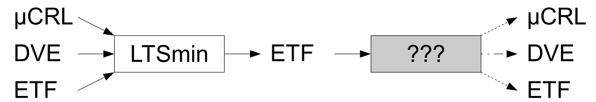

Figure 1. Process of converting higher level modeling languages into ETF and back again. The grey box is to be filled in.

2. PROBLEM STATEMENT

The main problem for this paper is converting the ETF generated by a tool like LTSmin back to the original model. We then benchmark the new model and compare it to the original model to see how the new model performs compared to the old model. In this paper we will be focussing on DVE (Distributed Verification Environment[6, 1]) as the language for the original models.

At the basic level this is not that hard, since you can just generate one DVE process containing all the transitions in the ETF. However, such a monolithic structure is not efficient for model checking. The goal is then to get the most efficient parallelization possible.

2.1 Parallelization

The simplest way of converting the ETF to DVE is to make all variables global, and have one process with all the transitions. Since none of those variables are state-transitions, the process has only the initial state and all transitions are self-transitions.

It is more efficient if state variables can be identified[2] for separate parallel processes. We will call these variables *marked variables*. Since other processes cannot write to state variables, this implies that no transition can alter more than one marked variable.[3] So whenever a transition alters two or more variables, at most one of them can be marked.

The transitions can then be distributed over the processes identified by their marked variable (which is its state variable). Whenever a transition alters a marked variable it will go to the process of that variable. It could be that some transitions do not alter a marked variable. In that case we create one additional process called the *bucket process* which will contain all those transitions.

Note that the simple way described at the beginning of the subsection is equal to marking no variables, so that all transitions will go into the bucket process.

Taking our example, we have the following transitions (variable names and line numbers added for clarity):

```
 v:  a   b   x
 1: 0/0  *  0/1
 2: 0/1  *  0/1
 3: 1/0  *  0/1
 4: 1/1  *  0/1
 5:  *  0/0 1/2
 6:  *  0/1 1/2
 7:  *  1/0 1/2
 8:  *  1/1 1/2
 9: 0/0 0/0 2/4
10: 0/0 1/1 2/3
11: 1/1 0/0 2/3
```

[2] A *state variable* is the virtual variable representing the state of a process

[3] Technically this could be done by synchronization, but that is out of the scope of this paper

```
12: 1/1 1/1 2/4
13:  *   *  3/0
14:  *   *  4/0
```

One way to mark these is by marking the variables a and b. Transitions 1 to 4 will go into process A, transitions 5 to 8 will go into process B, and the rest will go into the bucket process. Another way will be to mark variable x, in which case all transitions will go into process X.

2.2 Independent Set

An *independent set* is a set of nodes in a graph such that no node in the set neighbors another node in the set. Formally: Given an graph G with nodes V and edges E, the independent set I is defined as

$$\text{IS}(I) := I \subseteq V \qquad \forall v, w \in I : (v, w) \notin E$$

A ***maximal*** *independent set* is an *independent set* such that there exists no node that, when the node is added to the set, it is still an *independent set*. Formally: a maximum independent set I_m is defined as

$$\text{mIS}(I_m) := \text{IS}(I) \wedge \forall v \in V \setminus I_m \, \exists w \in I_m : (v, w) \in E$$

A ***maximum*** *independent set* is an *independent set* such that there exists no *independent set* with more nodes in it. By definition a ***maximum*** *independent set* is also a ***maximal*** *independent set*. Formally, a maximum independent set I_M is defined as

$$\text{MIS}(I_M) := \text{IS}(I_M) \wedge \neg \exists I_m \, \text{IS}(I_m) \wedge |I_m| > |I_M|$$

Essentially the marking of the variables comes down to an independent set problem. When you create a graph where the nodes represent the variables and the edges represent two variables being altered in one transition, then every independent set represents a marking of the variables as described in the previous subsection.

The trick here is finding the best independent set algorithm for this problem. Note that a *maximum* independent set algorithm might not be the best since it can (and probably will) mark *secondary variables* which are only used sparingly. This will result in a main variable (which probably was a state variable in the original process) not being marked. This in turn can result in a lot of unmarked transitions, resulting in a big bucket process. This not necessarily a drawback and its impact is investigated.

A *maximal* independent set however, is always optimal. If a set is not maximal, this means there are additional variables which could be marked. If they are not marked, all transitions writing them would go in the bucket process. This would make the bucket process unnecessarily large.

Since a maximum independent set algorithm is probably not optimal, we might need a weighted independent set algorithm. We can then assign weight to the variables such that variables which are used more frequently get more weight.

It must be noted that the independent set problem is NP-complete. For small number of variables an exponential algorithm might be feasible, but for larger models approximations might be necessary.

3. RESEARCH GOALS

The goal of this paper is to create an algorithm to convert an ETF model, created by LTSmin from a DVE model, back to an as efficient as possible DVE model. This is done by parallelizing the ETF model into separate processes.

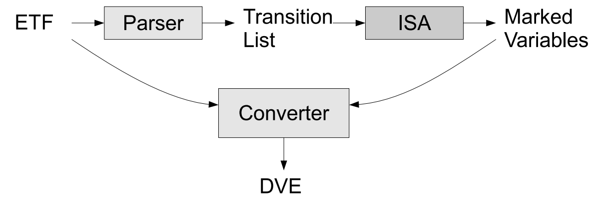

Figure 2. Process of converting ETF back to DVE.

3.1 Research Questions

The research goal spawns some extra research questions:

1. Will a normal independent set algorithm do? Or should a weighted independent set algorithm be used?

2. In case of weighted independent set algorithms, how should the weights be defined? And in what way should they be optimized? (e.g.: Maximum sum, maximum product)

3. Which independent set algorithm performs better? An exhaustive search, or fast but suboptimal one. In the latter case, which one?

4. RESEARCH TOOL

The chain of converting ETF into DVE is shown in figure 2. The Parser parses the ETF and output a list of transitions, specifically which variables are read (transition guards) and which variables are altered (transition effects). Then the Independent Set Algorithm will select from those variables a list of variables to be marked. The Converter will then use the variable list to convert the ETF into a DVE.

4.1 Parser and Converter

Since one of the input languages in most LTSmin tools is ETF, we reused the parser from LTSmin and modified it to output the transition list. Each transition section in the ETF gets a line stating which variables are read, which are written and the number of transitions that section has.

For the converter we reused the ETF parser again, but we had to build the DVE from scratch. First the processes had to be identified, but since the input is the list of marked variables, they would be the basis for the processes. Furthermore an additional bucket process was defined.

Next, all transitions were distributed over the processes. When a transition writes a marked variable it will go to its respective process. When it does not write a marked variable, but when it reads any one marked variable, it will also go to its respective process. This is an optimization so that self-transitions will also go to their own process. When no marked variable is written, and either no marked variable or two or more marked variables are read, then the transition goes to the bucket process.

Then all variables are scoped. There are three scopes: global variable, local variable and process variable. All the marked variables become process variables, the variables who are specified only in transitions belonging to one process are local variables. All the others are global variables.

At last the DVE output is generated. With the above information this is rather trivial. The main concern is skipping empty processes. Processes can be empty for two reasons: it is the bucket process, but all transitions

have a proper process, or a variable was defined but never used. In the last case that variable will automatically be marked, because there exists no transitions which alters that variable along with another (potentially marked) variable. Thus it will become a process, but have no transitions.

One of the left over problems is the naming of the variables and state names. Since ETF does not limit characters in variable names, all variables names have to be converted to legal DVE identifiers. We converted all non-alphanumeric characters to underscores. Should an identifier start with a numeric character, an underscore is prepended.

From DVE -> ETF -> DVE conversion there are two main sources of non-alphanumeric characters: local variables and arrays. A local variable `a` of process `X` will be encoded as `X.a` in the ETF and thus as `X_a` in the generated DVE. Since the ETF only provides simple numeric types (only nonnegative integers) this means arrays will be split into separate variables. An array `byte fork[3]` will be split into three variables: `fork[0]`, `fork[1]` and `fork[2]` in the ETF, and `fork_0_`, `fork_1_` and `fork_2_` in the generated DVE. They cannot easily be joined together as one array again, because they can be local variables in separate processes or even become process variables themselves.

The last problem is that of state names. One simple solution is to just give them names like `s0`, `s1`, `s2`, etc. But when possible we try to reuse the old state names, should a process variable in the original DVE also become a process variable in the generated DVE.

4.2 Independent Set Algorithm

The other major component in the tool chain is the independent set algorithm. It has to convert a transition list into a marked variable list. Independent set algorithms come in two major flavors: normal (i.e.: non-weighted) and weighted. The normal algorithms are easier to implement, but the weighted versions give more control of the final list.

4.2.1 Normal Independent Set Algorithms

The first few algorithms to test are normal (i.e. non-weighted) independent set algorithms (ISA). Since they are not weighted, they are prone to selecting secondary variables. They are however easy to write and they don't need a list of read variables (transition guards).

ISA 1: Exhaustive Search.

This will be a simple algorithm which just checks all combinations of variables and outputs the largest set. This will guarantee a maximum independent set, but runs very slowly in $O(2^n)$.

Indication in benchmark: complete

ISA 2: Most Neighbors.

This is a greedy algorithm which iteratively removes the variable with the most neighbors from the graph, until none of the variables have any more neighbors. This is a relatively fast algorithm ($O(n^2)$), but it will not always return a maximum independent set.

This algorithm was not implemented.

ISA 3: Least Neighbors.

This is a greedy algorithm which iteratively marks a variable with the least number of neighbors and then removes that variable and all its neighbors from the graph. This algorithm is potentially faster than the most neighbors algorithm, since it removes more than one node each iteration. However it is more prone to selecting secondary variables.

Indication in benchmark: min

4.2.2 Weighted Independent Set Algorithms

The normal independent set algorithms have a tendency to mark secondary variables because they can have very few neighbors. Aside from the fact that the resulting DVE model will probably differ from the original model,[4] it will also result in transitions not writing a marked variable. Since these transitions will go in the bucket process, this will make the bucket process bigger.

So we need an independent set algorithm which will prefer main variables (variables which are often accessed) over secondary variables (variables which are very sparsely accessed). This will be a weighted independent set algorithm (wISA).

For the weighted independent set algorithms we have a few things to worry about: How should the weights be calculated? In what way should the weights be optimized? And which algorithm should we use?

As for how to calculate the weights: the first heuristic we thought of is to use the number of transitions the variable is read from and written to.

As for how to optimize the weights: the default is to maximize the sum of the weight of the variables in the independent set, since that is how weighted independent set algorithms in literature are used.

The main objective of this setup is to minimize the bucket process. However, this setup has one major problem: suppose a variable is read and/or written in about every transition (e.g.: a counter variable), then that variable will get a lot of weight, probably bigger than any other independent set, making it the best maximal weighted independent set. This will result in one big process however, which is undesired.

There are a few alternatives to this:

1. Optimize for product instead of the sum of the weights.
2. Optimize the sum of the weight multiplied by the number of variables.
3. Minimize the maximum weight of a node in the set.
4. Let the node weight be the square root of the number of transitions.

All these options have the goal of avoiding "heavy" variables resulting in big processes. They do this by favoring more light variables over a few heavy ones.

Options 1 and 2 have the problem of there being no clear greedy version of the algorithm which performs better than the greedy version of maximizing the sum. One way would be to just randomly select a lot of independent sets and pick the best one. This will not provide consistent results.

Option 3 has the problem that it selects very light variables (in fact, those same secondary variables which we tried to avoid) at the cost of a big bucket, unless you also try to

[4]When the resulting DVE model differs from the original DVE model it is not necessarily a drawback. It can even be beneficial if the new model would perform better than the original.

minimize the bucket somehow. But we have not found a way to do that.

Option 4 has none of the above drawbacks, so we have chosen to implement this option.

wISA 1: Exhaustive Search.

This one is equivalent to the non-weighted version, except it uses weights to find the best set.

Indication in benchmark: w-complete

wISA 2: Greedy Maximum.

This is a generalized version of the Most Neighbors algorithm. Instead of selecting the node with the most neighbors, it selects the node v which minimizes $W(v)/d_G(v)$.

This is the GWMAX algorithm by Sakai, Togasaki and Yamazaki.[12]

This algorithm was not implemented.

wISA 3: Greedy Minimum.

This is a generalized version of the Least Neighbors algorithm. Instead of selecting the node with the least number of neighbors, it selects the node v which maximizes $W(v)/[d_G(v) + 1]$.

This is the GWMIN algorithm by Sakai, Togasaki and Yamazaki.[12]

Indication in benchmark: w-min

5. BENCHMARKING METHOD

The algorithms in the previous subsection must be benchmarked and compared to each other to figure out how well they perform. There are two issues here which can be benchmarked: the cost of generating an independent set (in terms of generation time) and the effective quality of the parallelization by the resulting independent set.

The cost of generating an independent set turned out not to be an issue. The greedy algorithms were so fast that even for the bigger models we tested (> 200 variables) the generation time was only fractions of a second. The exhaustive searches were heavily exponential in nature, so there was only a very small band where generation time was longer than fractions of a second, but still short enough to wait for it. Since initial results suggested that complete searches did not significantly outperform greedy searches, we have used greedy algorithms for every model having more than 20 variables. This was only based on a few tests, so we have not thoroughly benchmarked this.

To measure the effective quality of the parallelization we have compared the effective running times of the following programs:

divine.precompile Precompiling using DiVinE.

divine.generator Enumerate all states in DiVinE.

divine.reachability Reachability analysis in DiVinE collecting all sorts of data.

divine2-compile Compile model using DiVinE 2.

divine2-metrics Enumerate all states in DiVinE 2.

dve-reach Reachability analysis in LTSmin, optionally writing out an ETF file.

dve2lts-grey Enumerate all states in LTSmin.

DiVinE is the model checker for DVE models. It can do reachability analysis, but also property checking. However, we could not do property checking on the generated models because we lost two important pieces of information: accepting states and proper variable translation.

Acceptance of states is something which ETF simply does not store. Future versions might be able to do that, but modifying LTSmin in this way was out of the scope of this paper. The variable translation (e.g.: `X.a` => `X_a`) should be pretty straight forward, but a problem arises when a normal variable is converted to a process. Because whereas for variables you can use any mathematical expression, for processes you can only check whether or not it is in a certain state. This conversion is possible, but was also out of the scope of this paper.

DiVinE 2 is a new experimental version which works in a completely different way. So it will be interesting to compare performance.

`dve-reach` is the tool used to generate ETF files but it is also the best tool to benchmark DVE modes. `dve2lts-grey` simply enumerates all the states, but can also be used to generate diagnostic information, particularly about transition grouping.

As stated earlier, transition grouping can play a large role in performance, especially for `dve-reach` in both steps (generating ETF and benchmarking DVE). Therefore most LTSmin utilities have regrouping options to generate a more optimal transition grouping. The ones used during benchmarking are no regrouping, `-rgs` (Group Safely) and `-rga` (Group Aggressively). Especially the latter is a trade-off between longer regrouping time and shorter executing time.

Something which was a bit counter-intuitive was the fact that aggressive regrouping at the ETF-stage might actually hinder rather than help performance. The main reason is that aggressive regrouping will sometimes opt to combine groups of different variables. This makes the variable graph more connected and in turn make the independent sets smaller.

For example, where in one group the variables a, b and f were used, and in another the variables a, b and x, then they might be combined into a group with the variables a, b, f and x. This might mean there will be a connection between x and f in the new group, so they can no longer be both in the independent set.

All the programs are ran 3 times and the results were averaged.

5.1 Models

The models we benchmarked were taken from the BEEM (Benchmarks for Explicit Model Checkers)[11] site. To benchmark we needed models which did not run too quickly, but didn't take forever either. Since we wanted to check a lot of different combinations of models, algorithms and programs we targeted for a maximum running time of 10 minutes for any step in the process. For reachability analysis this came down to no more than 1,000,000 states, but at least 100,000 states.

However, some of the easier models took very long to compile. In particular these were models which had very complex transitions and resulted in very large ETF files. Luckily we already had a collection of ETF files for all the BEEM models, so we selected the models which had an ETF size of less than 5000 lines.

The resulting list of 20 models is in appendix C.

For every model we benchmark the original, the plain version (one big process with all transitions, equivalent to an empty independent set), a non-weighted ISA version, and a weighted ISA version. If the number of variables in the model is small enough we take the exhaustive search, otherwise we take greedy minimum.

5.2 Summary of the benchmark variants

Summarizing we check the running times of the following cross product of options:

- 20 different models
- ETF with no regrouping, and `-rgs` and `-rga` options
- Original, plain, non-weighted and weighted ISA
- 7 different programs (2 compilers, 3 enumerators, 2 reachability analyzers)
- For the LTSmin programs no regrouping, and `-rgs` and `-rga` options

All these were run 3 times.

5.3 Test setup

The tests were run on a cluster of 10 equal machines. They run on Dual Intel E5520 processors with 24GB RAM memory. Every benchmark for one model ran completely on one of the machines, so standard deviations only checked on conditions on one machine, not performance differences between machines. But since we have not compared models to each other anyway, this is not an issue.

6. RESULTS

Relevant result graphs are shown in the appendix. Here the results are explained in words. All running times are in CPU seconds of user time.

6.1 Regrouping on ETF level

The effect of regrouping on the original ETF varied by program. Graph 3 summarizes the effects over all the programs. For compiling no regrouping or safe regrouping provided the best results. Aggressive regrouping was slower. The same goes for the DiVinE reachability analyzer. For both enumerators it mattered very little what kind of regrouping was done.

For `dve-reach` the results varied more significantly. There are spikes to both sides where aggressive regrouping is between 10 times faster to 2 times slower. The spikes for safe regrouping are smaller. On average, both regrouping options are faster than no regrouping for `dve-reach`.

Important note: `dve-reach` is used two times in the whole chain, and both have regrouping options. We are talking about the effect of regrouping at the first `dve-reach` at the start of the chain, on the last `dve-reach` at the end of the chain.

For the rest of the benchmark results we have used no regrouping on ETF level, since that is the most neutral.

6.2 Programs

6.2.1 Compilers

The DiVinE compiler is generally around twice as fast as the DiVinE 2 compiler (see figure 4). It did not matter what method was used for generating the DVE for compile times, as can be seen in figure 8 (figure for DiVinE 2 compiler is similar). Only the original model was often much faster. This is probably because all models generated from the same source model have similar transition lines. Only the ordering and grouping was different.

6.2.2 Enumerators

The DiVinE generators were always faster than `dve2lts-grey`, but quite loosely correlated (see figure 5). The difference between the DiVinE enumerator and the DiVinE 2 enumerator was not very big.

The enumerators are all fairly oblivious to what ISA is used. The running times for original (best case) and plain (worst case) are about equal.

Furthermore using the regrouping options on `dve2lts-grey` had no effect except for a very large spike with `fire_wire.4`, which could be explained by the large number of variables. Because even though it had no effect on enumeration, using the regrouping options still caused the regrouping to be calculated, which was a very expensive operation with such a large number of variables.

6.2.3 Reachability Analyzers

There is not much to say about the reachability analyzers in particular. There was practically no correlation between the analyzers (see graph 6). `dve-reach` averaged about 40 seconds for a run, and the DiVinE reachability analyzer averaged about 100 seconds.

6.3 The algorithms

Since both the enumerators as the compilers were fairly oblivious to the algorithm used, we will only compare the reachability analyzers.

6.3.1 Algorithm vs Original

When we check the running times of using a particular independent set algorithm and compare them to the running times of the original, there is not much of a correlation. The only real result is that the original is in almost all cases a lot faster than all the generated DVE models.

An example graph for the DiVinE Reachability Analyzer is in figure 7. The graph for `dve-reach` is similar.

6.3.2 Algorithm vs Plain

For the DiVinE Reachability Analyzer there was a strong correlation between the generated models (see figure 9). The ISA models (all except the originals and plain version) all performed about equally well, but they were all about 30-50% faster than the plain version.

For `dve-reach` however the correlation was very weak, as can be seen in figure 10 (with save and aggressive regrouping the graphs are a little bit more extreme, but otherwise similar). But we can see that on average the non-plain models were faster than the plain models. Compared to the plain version the span for no regrouping is from 6 times slower to about 30 times faster. The span for safe regrouping is from around 3 times slower to almost 150 times faster.

6.3.3 Weighted vs Non-weighted

For the DiVinE reachability analyzer it mattered very little whether a weighted or non-weighted version of the Independent Set Algorithm was used, as can be seen in figure 11.

For `dve-reach` the correlation is much less clear, as can be seen in figure 12. Especially for the complete search the results differ a lot. One explanation could be that the running times were too low, and that other factors (like initialisation) were too dominant. Another explanation could be that the weight function was a poor choice. Or maybe a combination of both.

6.3.4 *Regrouping on* `dve-reach` *on benchmarking step*

Using any of the `-rga` or `-rgs` options on `dve-reach` during benchmarking can make a lot of difference. Speeds can improve from around 36 seconds to under 1 second, or from 327 seconds to around 41, though most of the timing stayed around the same. See figure 13.

The difference between either regrouping option is however less pronounced. On average the speed increase of `-rga` over `-rgs` is less then 10%. See also figure 14.

7. CONCLUSIONS

The first conclusion is that the generated DVE seldom approach the efficiency of the originals. This could be because the translation is still relatively simple. You could for example try to detect */N transitions (transitions where the source value for a variable does not matter). Also synchronisation might make the model simpler.

Another conclusion is that parallelizing the generated DVE indeed results in a significant improvement of performance, from about 30-50% faster on the DiVinE Reachability Analyzer to up to 150 times faster on `dve-reach` with regrouping.

Another result is that, even though parallelizing is beneficial, the exact independent set algorithm used does not matter very much for the DiVinE reachability analyzer.

The results for `dve-reach` are still very inconclusive. Results vary wildly from model to model. More research might be needed on that.

Finally the enumerators are not sensitive to how the model was put together, just how many states they had. For the compilers the size of the input model was most important. But since the size of the generated states was based solely on the transitions, it was independent of the way they were combined, and therefore the ISA used.

8. FURTHER RESEARCH

This research did not go really deep in the different independent set algorithms, nor the different weight functions for the weighted independent sets. More research could be done on that front. Also, the running times for some of the measurement times were too low. There were a lot of running times of less then a few seconds or even less than a second. Higher running times could mean more accurate results for comparison. The problem here was that there were too many dimensions too consider. Now that the interesting dimensions have been identified, one could dive deeper into them.

Something else which might still be interesting in benchmarking is comparing actual model checking on the different versions of the models. Model checking on DiVinE is done by combining LTS properties and a DVE model into a new DVE model. This is done by converting states to accepting states. However, this acceptance property is not preserved in the ETF format, it is not possible to check for the properties in the generated model. An alternative is to combine the properties on the generated models. But because of name and type translations on the variables there must be additional processing on the properties to make them valid for the new models.

Another potential performance increase could be the addition of synchronisation. This opens the possibility of having a maximum of two marked variables written per transition, which are synchronized on a channel. However, this would need a modified version of the independent set definition to work, complete with new algorithms.

9. ACKNOWLEDGEMENTS

I would like to thank Stefan Blom for his help and for writing the first versions of the Parser and Converter.

10. REFERENCES

[1] DiVinE project website. `http://divine.fi.muni.cz/`, per 22 March 2010.

[2] Enumerated table format. Available at `http://fmt.cs.utwente.nl/tools/ltsmin/etf.html`, per 22 March 2010.

[3] LTSmin. Available at `http://fmt.cs.utwente.nl/tools/ltsmin/`, per 22 March 2010.

[4] μCRL website. `http://homepages.cwi.nl/~mcrl/`, per 22 March 2010.

[5] Spin website. `http://spinroot.com/`, per 22 March 2010.

[6] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification*, volume 4144/2006 of *LNCS*, pages 278–281. Springer Berlin / Heidelberg, 2006.

[7] S. C. C. Blom, J. C. van de Pol, and M. Weber. Bridging the gap between enumerative and symbolic model checkers. Technical Report TR-CTIT-09-30, Centre for Telematics and Information Technology, University of Twente, 2009.

[8] M. Bozga, S. Graf, and L. Mounier. If-2.0: A validation environment for component-based real-time systems. In *In Proceedings of Conference on Computer Aided Verification, CAV.02*, pages 343–348. Springer Verlag, 2002.

[9] J. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, 1990.

[10] T. Margaria, R. Nagel, and B. Steffen. Remote integration and coordination of verification tools in jeti. In *ECBS '05: Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, pages 431–436, Washington, DC, USA, 2005. IEEE Computer Society.

[11] R. Pelánek. Beem: Benchmarks for explicit model checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[12] S. Sakai, M. Togasaki, and K. Yamazaki. A note on greedy algorithms for the maximum weighted independent set problem. *Discrete Applied Mathematics*, 126(2-3):313 – 322, 2003.

APPENDIX

A. DEFINITIONS

DVE Distributed Verification Environment. The high level model used in this paper.

ETF Enumerated Table Format. The low level labelled transition system dump used by LTSmin.

independent set A set of nodes on a graph such that for every two distinct nodes in the set, there exists no edge between them.

LTSmin A toolset for manipulating labelled transition systems.

main variable A variable playing a big role in a transition system. Usually a state variable.

maximal independent set An independent set such that there exists no node on the graph which can be added to the set so that the set is still independent (i.e.: a local maximum).

maximum independent set An independent set such that there exists no other independent set on the graph containing more nodes (i.e.: a global maximum).

secondary variable A variable playing a small role in a transition system. Usually a local variable.

state variable The variable saying what state a process is in.

B. EXAMPLE MODEL IN ETF

Example model in ETF

```
begin state
X\.a:byte X\.b:byte X:X
end state
begin edge
end edge
begin init
0 0 0
end init
begin trans
 0/1 * 0/1
 0/0 * 0/1
 1/1 * 0/1
 1/0 * 0/1
end trans
begin trans
 * 0/1 1/2
 * 0/0 1/2
 * 1/1 1/2
 * 1/0 1/2
end trans
begin trans
 0/0 0/0 2/4
 0/0 1/1 2/3
 1/1 0/0 2/3
 1/1 1/1 2/4
end trans
begin trans
 * * 3/0
end trans
begin trans
 * * 4/0
end trans
begin sort byte
end sort
begin sort X
"x0"
"x1"
"x2"
"x3"
"x4"
end sort
```

C. MODELS

The following is a list of models used from the BEEM[11] site:

- bakery.4
- brp.3
- cyclic_scheduler.3
- cyclic_scheduler.4
- exit.5
- firewire_tree.4
- lamport.2
- lann.4
- lann.5
- leader_election.3
- leader_election.4
- mcs.3
- mcs.6
- msmie.3
- peterson.3
- phils.4
- phils.5
- production_cell.4
- protocols.4
- protocols.5

D. GRAPHS

On the next pages is a series of graphs extracted from the benchmarks. All values are in CPU seconds of user time. Some graphs feature one or two regression lines, together with regression formula's (in the form of $f(x) = Ax + b$) and a Pearson correlation coefficients (R^2).

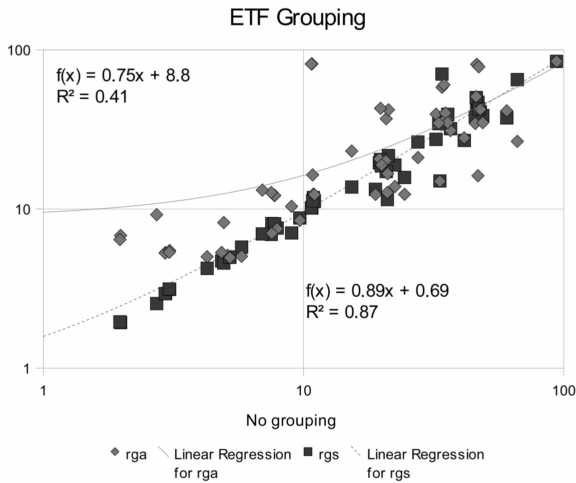

Figure 3. ETF grouping

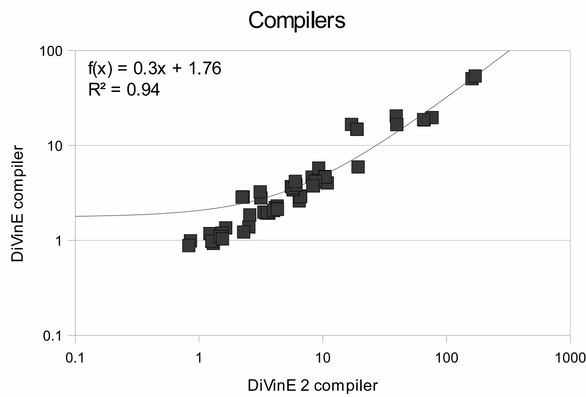

Figure 4. Compilers

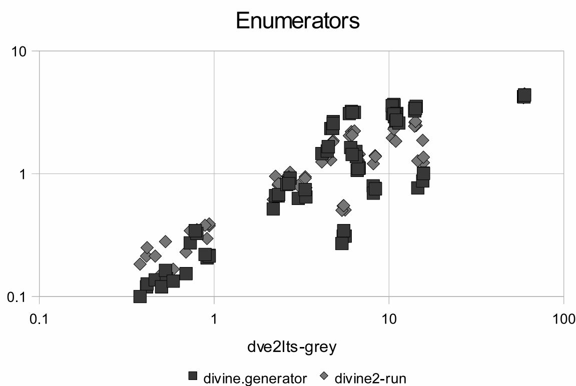

Figure 5. Enumerators

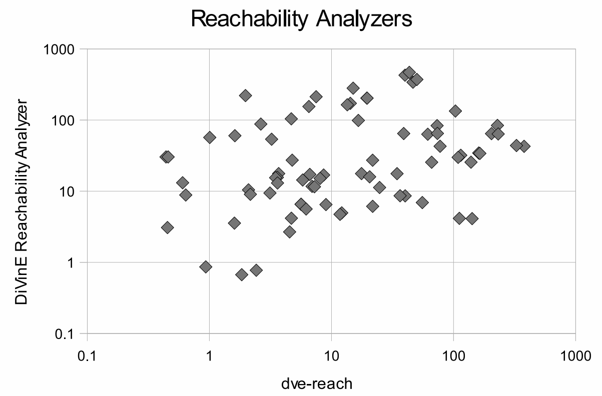

Figure 6. Reachability Analyzers

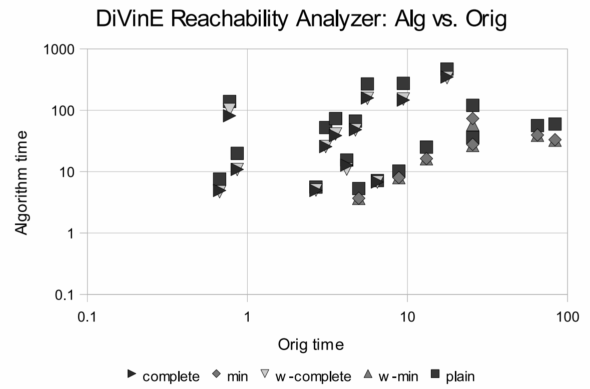

Figure 7. DiVinE Reachability Analyzer: Alg vs. Orig

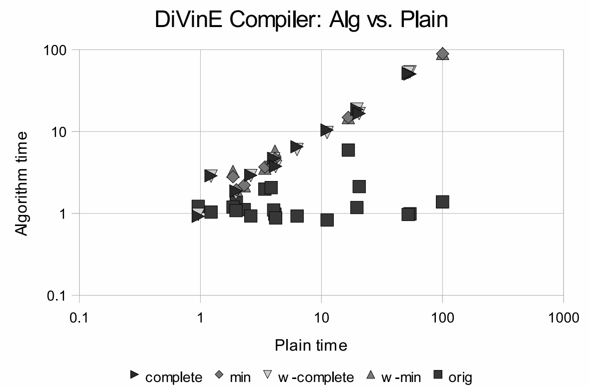

Figure 8. DiVinE compiler: Alg vs. Plain

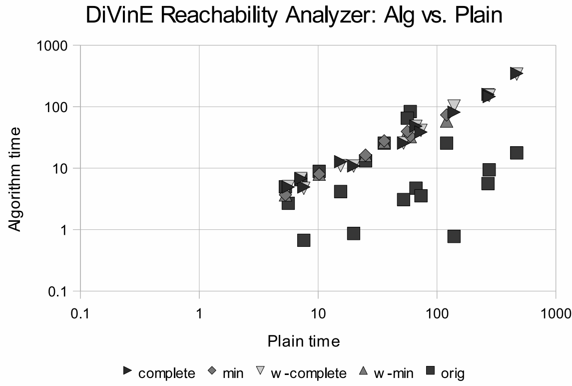

Figure 9. DiVinE Reachability Analyzer: Alg vs. Plain

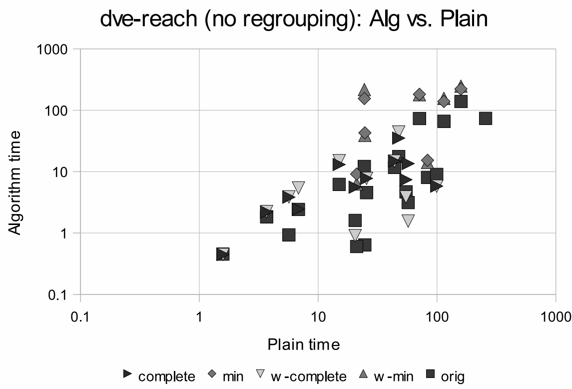

Figure 10. dve-reach (no regrouping): Alg vs. Plain

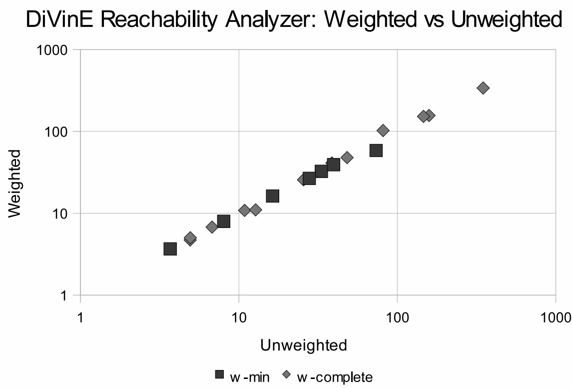

Figure 11. DiVinE Reachability Analyzer: Weighted vs. Unweighted

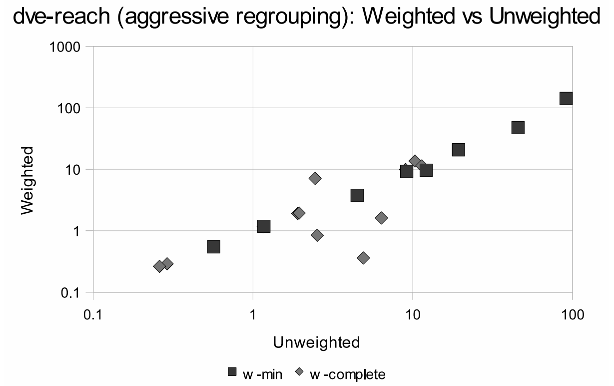

Figure 12. dve-reach (aggressive regrouping): Weighted vs. Unweighted

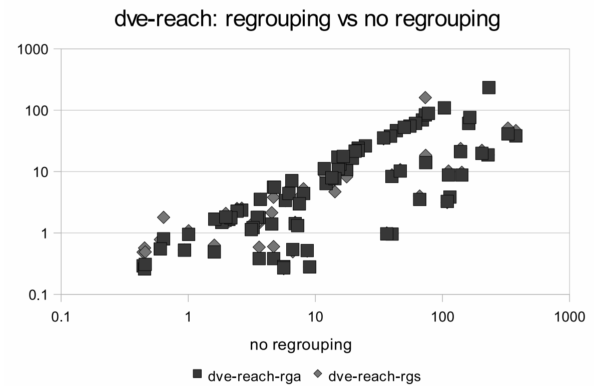

Figure 13. dve-reach: regrouping vs. no regrouping

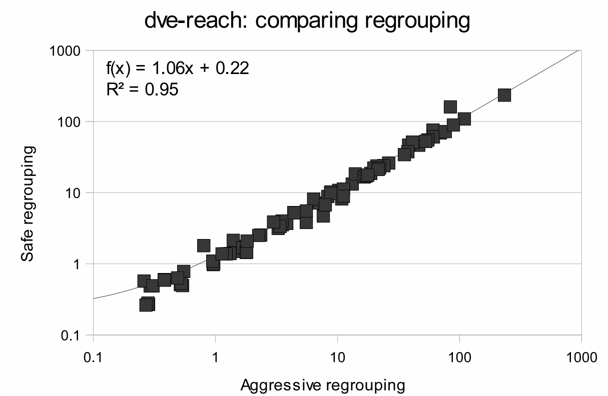

Figure 14. dve-reach: comparing regrouping