Applying Compiler Techniques to Reachability Analysis of High-Level Models

Marko Mäkelä[*]
Helsinki University of Technology,
Laboratory for Theoretical Computer Science,
P.O.Box 9700, 02015 HUT, Finland

Abstract

Using a tool for high-level Petri nets as an example, this article shows how techniques familiar from compilers can make reachability analysers more powerful.

Syntax transformations can be applied to extend the modelling language with convenient short-hand notations, such as universal and existential quantification and multi-set summation.

The process of reachability analysis can be dramatically sped up by compiling models to executable machine code that performs all model-dependent tasks, such as computing the successors of a state. This work describes a code generator implementation and the way the generated code is linked with the reachability analyser code.

Keywords. reachability analysis, compilers, syntax transformations, code generation

1 Introduction

One major problem in the reachability analysis of practical systems is the expressibility of the modelling formalism. The more expressive power a formalism offers, the easier it is to construct models in it. In the field of computer software or protocols, one particularly attractive class of formalisms is the one with user-definable data types. There is a clear analogy between programming and modelling: high-level languages make it possible to construct compact and easy-to-understand programs and models. Alas, care must be taken when equipping a formalism with expressive power, or the models might be difficult to analyse.

Using high-level languages involves a performance penalty. The underlying computing machinery must somehow be instructed to carry out the operations written in the high-level language. This is usually done by translating the high-level operations to sequences of lower-level operations, either one at a time (in an interpreter) or bigger units at a time (in a translator). An interpreter performs translations all the time, while a translator is executed only once, before the actual computations take place.

[*]This research was financed by the Helsinki Graduate School on Computer Science and Engineering, the National Technology Agency of Finland (TEKES), the Nokia Corporation, Elisa Communications and the Finnish Rail Administration.

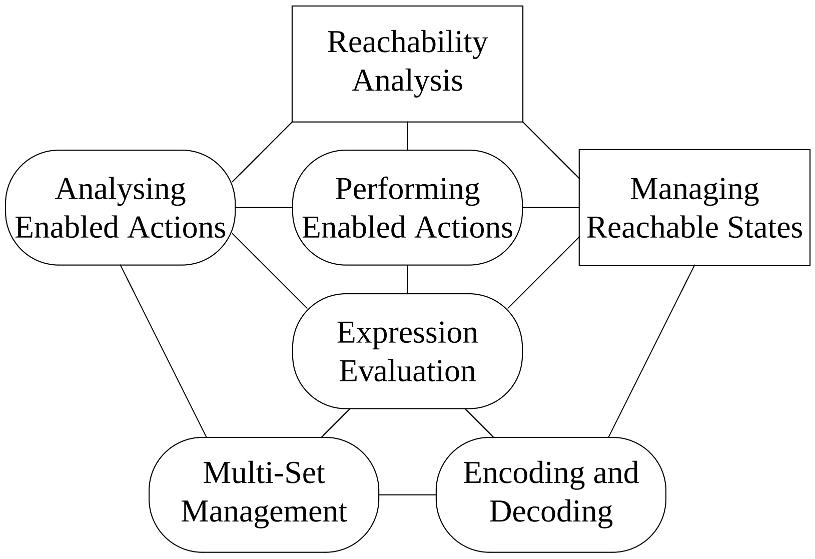

Figure 1: The Block Diagram of MARIA

A reachability analyser works by processing each reachable state of a model, determining which actions can be performed, and by extending the set of reachable states with the states generated by performing the actions. Thus, the actions defined in the model are likely to be performed several times in succession. Compared to interpreting, translating the model reduces the time spent analysing each state, but it takes more time before the first state can be analysed. In interactive simulations, it may make sense to interpret instead of translating.

This work is based on the implementation of MARIA [11, 12], an analyser for a class of high-level Petri nets known as Algebraic System Nets [10]. The functional blocks of this analyser are illustrated in Figure 1.

We describe two kinds of transformations. Some of them translate short-hand notations to the lower-level internal syntax of the tool, while others generate C [8] code for various algorithms and tasks—the blocks enclosed in rounded boxes in Figure 1.

2 Transforming Short-Hand Notations

Many high-level languages include constructs that save typing effort, such as macros, or the ability to declare variables in the middle of a block. Reachability analysers for high-level models are no different. For instance, while MARIA does not support lexical macros à la PROD [15], it implements global and local functions, a kind of type-safe macros for use in expressions.

2.1 Quantification

One thing that distinguishes MARIA from many other reachability analysers is its data type system and the way it stores values in the reachability graph [13]. The tool defines a *total order for all predefined and user-defined types*, even deeply structured ones.

Total order makes it straightforward to implement e.g. an operator for nondeterministically choosing the values for the output variables of a transition: just start from the smallest value for the specified data type and apply the successor relation until all values of the type have been processed.

Our implementation handles nondeterminism in a similar way to *quantification*, iteration through all values d of a data type $\mathcal{D}$. In MARIA, operations based on iteration include multi-set summation and existential and universal quantification.

For instance, the multi-set summation

$$\sum_{\substack{d\in\mathcal{D}\\ f(d)}} \langle g(d), h(d)\rangle$$

over the items $h(d)$ of multiplicity $g(d)$ is equivalent to $\langle g(1), h(1)\rangle + \langle g(3), h(3)\rangle$, assuming that $\mathcal{D} = \{1,2,3\}$, $f(1) = f(3) = \top$ and $f(2) = \bot$.

It is worth noting that the summation condition $f(d)$ does not need to be a constant expression. If $f(d)$ refers to a variable, we may merge the condition with the summand by writing

$$\sum_{d\in\mathcal{D}} \langle [f(d)]g(d), h(d)\rangle$$

where the brackets map truth values to integers: $[\bot] \mapsto 0, [\top] \mapsto 1$.

This is the principle how MARIA treats multi-set summation expressions. They are expanded to multi-set additions of individual terms, simplified by means of constant folding and a search structure for the summands. For instance, if we had $h(1) = h(3) = 2$, $g(1) = 3$ and $g(3) = 4$ in the above example, the sum would be expanded to $\langle 7, 2\rangle$ by the parser. Originally we implemented quantified multi-set summation as an algebraic operation. It complicated the algorithms considerably, especially the algorithm that finds enabled actions. Expanding the summations in the parsing stage cut the analysis times of some models to less than half.

Similarly, the existential quantification

$$\exists d \in \{d' \in \mathcal{D} \| f(d')\} : i(d) \quad \text{or} \quad \exists d \in \mathcal{D} : f(d) \rightarrow i(d)$$

is equivalent to $i(1) \vee i(3)$ if $\mathcal{D}$ and f are defined as in the summation example. Since the logical conjunction and disjunction operators in MARIA have short-circuit semantics,[1] the quantification can be aborted as soon as a term evaluates to $\top$.[2] Universal quantification is handled analogously, using conjunction and $\bot$.

2.2 Variable Declarations

Some high-level Petri net analysers allow implicit variable declarations for transitions. That is, if a previously undeclared variable is referenced on an input arc, the variable will be silently declared.

We go a step further by supporting implicit declaration of groups of variables. Without this feature, the quantification operations described in the previous section would be of limited use.

Consider a net containing a place with a multi-set over $\mathcal{D} = \{1,\dots,n\} \times \{\bot, \top\}$ as its domain. The place could hold one proposition for each of the n entities $1,\dots,n$. Reading out all those propositions in a transition requires n variables.

[1]Short-circuit evaluation means that $\top \vee x$ evaluates to $\top$ and $\bot \wedge x$ to $\bot$, no matter what the x stands for.

[2]We cannot replace a term $a \vee \cdots \vee z \vee \top \vee \cdots$ directly with $\top$, since it would hide the possible side effects (evaluation errors) of the term $a \vee \cdots \vee z$.

```
typedef unsigned (1..3) entity;
place p struct { entity e; bool b; }:
  entity e: { e, false };
trans t in { p: entity e: {e, .b} }
gate entity e || .b;
```

Figure 2: Quantification and Variable Declarations in MARIA

```
typedef unsigned (1..3) entity;
place p struct { entity e; bool b; }:
  {1, false}, {2, false}, {3, false};
trans t in { p: {1, "b[1]"}, {2, "b[2]"}, {3, "b[3]"} }
gate "b[1]" || "b[2]" || "b[3]";
```

Figure 3: The Model of Figure 2 After Expansion

Figure 2 shows how this can be arranged in MARIA, and Figure 3 shows how the parser expands the quantification in the case $n = 3$. The net illustrated in the figures has one place and one transition. The place is initialised with pairs whose first component is an entity identifier and second component is a Boolean constant $\bot$. The initial marking is the multi-set sum

$$\sum_{e \in \{1,\ldots,n\}} \langle e, \bot \rangle.$$

The transition removes all $\langle e, b_e \rangle$ pairs from the place, provided that

$$\exists e : b_e = \top \text{ (Figure 2)} \quad \text{or} \quad b_1 \vee b_2 \vee b_3 \text{ (Figure 3)}.$$

This fails to hold in the initial marking, and the transition is disabled. This construct could be applied to modelling a distributed service. The propositions bound to the variables could identify e.g. the server nodes that have responded to a query. Then the transition is enabled if at least one server has responded to the query.

3 Generating Executable Code

Once the first version of MARIA was up and running, we made some performance measurements. We could immediately identify two bottlenecks in the system: expression evaluation and transition instance analysis—the procedure that determines the set of enabled actions.

The expression evaluator in MARIA is written in C++ [7] just like the rest of the analyser. Expressions, data types and values are objects of classes derived from abstract base classes. The expression evaluator—an abstract method of the expression class—constructs a new value object for each subexpression evaluated.

To get rid of this performance bottleneck, we decided to translate expressions to machine code that evaluates them. As we shall see later, expression evaluation is not the only task that can be translated. In principle, any program that processes input written in a certain language can be translated to an equivalent program that handles a narrower set of input.

A program specialised for a very narrow set of input can be made more efficient than a fully generic program. Replacing variables with constants does not merely reduce the number of memory references caused by executing the program. For instance, if the sizes of some data structures can be statically determined from the input, a program specialised for a particular input can use static data storage, while the generic program has to use dynamic memory allocation. Another example is arithmetic operations. Multiplication or division by a power of two can be reduced to bit-shifting, a much cheaper operation. Comparisons between variable data structures can be simplified to comparisons between variables and constants, or eliminated altogether.

In order to keep the translation somewhat maintainable and portable, we chose to use C [8] as an intermediate language. The generated code is compiled to a shared library that is dynamically loaded into the reachability analyser. In a broad sense, we have constructed a self-modifying program—something deprecated and discouraged by purists, but something that dramatically saves computing resources in practice.

3.1 Interfacing with the Generated Code

The idea of implementing a reachability analyser using generated machine code is not new. Tools that translate models to C programs that generate reachability graphs include SPIN [5], an analyser for concurrent processes communicating via message queues, and PROD [15], an analyser for Predicate/Transition nets [4].

The new aspects in our code generator are that the static part of the analyser is written in a different language from the generated code, and that the generated code is executed by the same program that produced it, making use of dynamic linking.

At first, one might think that dynamic linking is less efficient than static linking. It does not need to be the case. The code we generate contains only a handful of globally visible symbols. The addresses of these symbols are looked up only once, when the shared library has been loaded. In static linking, the linker would need to resolve all symbols in the various modules of the reachability analyser as well.

Many generated functions are related to data types. Our analyser has two representations for states: an expanded representation—in this case, the representation used by the generated code—and a compact representation that is used for permanently storing generated states. Since the way how the data structures of a C program are laid out in memory depends on the C compiler used, the generated code must carry out the conversions between the expanded and compacted representations.

The rest of the generated functions deal with the dynamic parts of the model, the transitions: arc expressions, gates (additional enabling conditions) and nondeterminism (values for output variables). In the following, we shall briefly describe the generated code for expression evaluation and transition instance analysis.

3.2 A Closer Look at the Expression Evaluator

Implementing an expression evaluator in generated code is not as simple as it seems at first sight. The expression evaluator in MARIA has two features that rule out the idea of directly translating each expression to a function consisting of a single `return` statement that computes the value.

In the C programming language, functions cannot return structured values. We cannot return a pointer to a value either, since it would violate our design decisions: the expression evaluator must be re-entrant, and it must not perform dynamic memory allocation. Instead, the caller of the expression evaluator must pass a pointer to a memory area large enough to hold the result.

The other major problem we had were evaluation errors. When verifying models of concurrent systems, we want to detect all errors. Some errors, such as division by zero or trying to write to a full buffer, can cause program termination or undefined behaviour. We certainly do not want that the analyser crashes or works unpredictably if there is an error in the model—instead, we must catch these errors and report them. In principle, this is simple: every time an operation is to be performed, we check its arguments, and abort the expression evaluation with an appropriate error code if the check fails.

3.2.1 First Approach: Mapping Expressions to Expressions

Initially, our expression evaluator generator mapped each subexpression in the MARIA language to an expression in C. Sometimes we had to generate a separate function for evaluating the subexpression, and we generated numerous auxiliary functions e.g. for computing successors and predecessors.

Our mechanism for detecting and reporting errors made use of three features of the C programming language: the if-then-else operator ?:, the comma operator, and the longjmp library function. For instance, we translated the unsigned expression a/b to something like ((!b?longjmp(*jmp,errDiv0):(void)0),a/b). This looks ugly, but it is a valid C expression that can be used as a subexpression.

This translation worked, but the performance of the generated code was unsatisfactory. A jump context buffer had to be initialised every time an expression was evaluated—a very expensive operation, especially on a RISC machine having a large register file. Also, many of the generated auxiliary routines were never used.

3.2.2 Second Approach: Splitting Expressions to Statements

If C compilers treated statements as void expressions, the longjmp calls in our initial translation could have been replaced with computationally much cheaper return statements. Because the flexibility of the C programming language has a limit here, we had to figure out something else.

We decided to introduce a temporary variable for each subexpression, and to evaluate each subexpression using a sequence of statements. Possible errors are checked in if(...)return... statements right before making assignments. The common subexpression elimination performed by the MARIA parser is useful here. For instance, if an expression makes several references to a variable, the generated evaluation code checks only once if it is defined.

In addition, we managed to eliminate all auxiliary functions. Eliminating the auxiliary functions for comparisons was the most difficult part. Comparisons for equality, against a constant or between simple values (values that can be represented in one machine word) are translated to simple C expressions. But what about the remaining cases?

Our translation for the remaining case, comparisons between non-constant structured values, is amazingly simple. Instead of translating these comparisons to lengthy expressions,[3] we

[3] Our data type system includes a variable-length buffer type. For a buffer of maximum length n, the length of a less-than comparison expression is in the order of n^2 times the length of the less-than comparison expression for the buffer item type.

```
struct s { int a; char b; };
int cmp3 (struct s* left, struct s* right)
{
  if (left->b < right->b) goto less;
  if (left->b > right->b) goto greater;
  if (left->a < right->a) goto less;
  if (left->a > right->a) goto greater;
  /* equal */
  return 0;
less:
  return -1;
greater:
  return 1;
}
```

Figure 4: Three-Way Comparison of Structured Values

decided to split the comparisons to several statements, comparing the components of the structured type one at a time, starting from the lexicographically most significant component. When a component of the left-hand value is smaller than the corresponding component of the right-hand value, we abort the comparison with a goto statement. Likewise, when the right-hand component is smaller, we jump to another location. Only if the most significant components match, the less significant components are compared. If the values are equal, the control flow falls through the whole chain of comparisons. In this way, we perform a three-way comparison. Figure 4 illustrates this for a very simple structured data type.

3.2.3 Managing Multi-Sets

Initially, we did not generate any code for maintaining multi-sets, the highest-level constituents of the model state. Instead, we relied on the map container template covered by the C++ standard [7] and maintained the multi-sets in the static part of the analyser.

This turned out to be inefficient in many aspects. Searching for items in multi-sets involve calls to two-way comparison functions—in our case, calls via function pointers. Maintaining the multi-sets in generated C code using balanced binary trees allows us to embed the comparisons directly in the code. Generating three-way comparisons speeds up equality testing of structured values, since one comparison can determine whether the two items are equal or which one is greater.

The C++ Standard Template Library replaces pointers with heavier-to-use iterator objects. Normally the iterators are bidirectional; also the special end() iterator can be iterated backwards to obtain the last element in the collection. Since our algorithms make no use of this, we can use pointers and a simple NULL pointer as the end marker.

Generating code for managing multi-sets is not as difficult as one might think at first. A multi-set over the items of a set A can be represented as a binary search tree whose keys are those items of A that have nonzero multiplicity in the multi-set. Generic operations, which are independent of the actual set A, such as iterating through all items contained in the multi-set, or balancing a tree after an insertion, can be implemented as C preprocessor macros or as functions that input the tree as a generic pointer. This allows us to avoid one problem of C++ templates: we do not instantiate these common routines for each type of tree, but share them as such for all trees.

Managing the multi-sets in the generated code makes the code fully self-contained: all calls to procedures in the static part of the analyser can be eliminated. The static part merely controls the whole play by invoking the generated code to analyse enabled actions, to perform them, to evaluate state formulae, and to encode or decode states and events.

Extending the generated code to a stand-alone reachability graph generator program would be easy, unless we wanted the user to be able to interrupt the analysis at any time and to examine the generated reachability graph. It is difficult to support the evaluation of arbitrary formulae without implementing a decent parser and keeping a syntax tree of the model in memory.

3.3 Transition Instance Analysis

In reachability analysers for high-level Petri nets, one of the most time-consuming tasks is determining which actions can be performed. An action in a high-level Petri net—a transition instance—consists of a transition and an assignment for its variables. A transition instance can be performed if it is enabled—if the assignment fulfills some conditions. It must be possible to evaluate all expressions of the transition using the assignment, all gate expressions must hold, and there must be enough tokens in the input places of the transition.

There are many ways to construct the set of enabled instances of a transition in a specified state of a model. One way is to construct all possible assignments for the transition variables, and to prune those for which the input condition or a gate expression does not hold. Since the transitions in high-level models typically have numerous variables with large domains, it pays off to take into account the gate expressions and the contents of the input places already while constructing the assignments.

The approach presented by Ilié and Rojas [6] incrementally constructs the set of enabled instances, starting with a single instance that leaves all variables undefined. The algorithm processes the input places of the transition in one pass. For each input place, it iterates through all arcs running from the place to the transition, and all tokens contained in the place. Each token is matched against the current arc expression, and instances are pruned or augmented accordingly.

Unfortunately this technique is hard to apply if the multiplicities on the input arc expressions are allowed to depend on variables. For instance, if a transition inputs a token x from place A, x copies of a token y from place B, and $x+y$ tokens z from the place A, the above algorithm will fail, since the multiplicity of the token z will be unknown.

The basic transition instance analysis algorithm implemented in MARIA iterates through the input places, input arcs and tokens in a different order than the algorithm presented by Ilié and Rojas. It performs a depth-first search of transition instances by iterating through the input arcs in the order they appear in the net description, until the current transition instance cannot be augmented further. For each input arc with known multiplicity, it iterates through all tokens in the corresponding input place, or—if the arc can be evaluated—searches the input place for the token the arc evaluates to. If the multiplicity of an arc depends on a variable that has not been assigned a value yet or if an expression evaluation error occurs, the current assignment is declared erroneous.

The main drawback of the C++ implementation of our algorithm is that the expression evaluator is repeatedly invoked on the same input arcs and gate expressions, even if nothing has happened to the variables they depend on. One tangible optimisation would be to evaluate only those expressions or subexpressions that are affected by changing the assignments of the variables. Another thing that could be done is to keep a cache and to return the already generated set of assignments if the marking of the input places of a transition has not changed.

Such optimisations can be implemented also without generating code, but code generation has further advantages. One is that the search stack required by our depth-first search algorithm can be replaced by a static structure—nested iteration loops. Another reason to generate a monolithic instance analysis procedure for each transition is the overhead caused by procedure invocations. The transition instance analysis algorithm needs to evaluate expressions, to find assignment candidates and to check whether assignments agree with input arc expressions or gates. Instead of generating procedure calls for all these tasks, we perform the whole instance analysis of a transition in one generated procedure.

3.4 Some Experiences

3.4.1 Debugging the Generated Code

In order to ease debugging, we decided to format the generated code nicely. It is rather pleasing to read the code—the only inconvenience is that all identifiers are mapped to numbers.[4]

With a good debugger, it is possible to debug the generated code at source level. When debugging, one may want to examine the contents of a multi-set. We do not generate auxiliary functions for displaying entire multi-sets. Instead, our run-time library—a set of header files—contains two functions for in-order iteration through binary search trees. These functions, together with the convenience variables of the GNU debugger [2], make it possible to dump any multi-set by repeatedly invoking a debugger command.

The only construct that the author considers a programming trick concerns reporting evaluation errors in the case when the generated code is embedded into another function. When an evaluation error occurs, we cannot `return` from the entire function; we just want to pass an error code to the enclosing function and to stop further evaluation of the expression. We accomplish this by enclosing the generated evaluation code in `do{...}while(0)` blocks and by generating `continue` statements instead of `return` statements for the errors.

3.4.2 Performance of Code Generation

Generating executable machine code is surprisingly cheap. When testing our implementation, we found out that generating code for a model and invoking a C compiler pays off already on models with a few thousand reachable states.

When developing a big model, it is often infeasible to wait some minutes for the compiler to finish and then notice that there is a mistake in the initial state. Fortunately, one can always use MARIA without code generation, and even generate the reachability graph interactively. That straightforward, unoptimised implementation is also practical when one suspects a bug in the code generator.

[4]One reason for this is that the modelling language allows everything except the NUL character in identifiers, while C compilers use a more restricted alphabet.

Secondly, we divide the generated code into modules and recompile only those parts which have changed. If one modifies only one transition in the model, we need to compile only one module, and to relink the modules together. Since generating the C code is much faster than invoking a C compiler, we always generate all code, and use checksums to find out which modules need to be recompiled. Only if data type definitions are changed or places are added or deleted, we may need to recompile everything.

Tables 1 and 2 show the performance of our implementation on a model of a distributed data base management system. This model , originally presented by Genrich, Lautenbach and Jensen [3, 9], is defined in our earlier article [13]. We vary two of its parameters: the number of servers n, and the presence or absence of a redundant place **unused**, which contains either $(n-1)^2$ or $(n-1)n$ tokens in all reachable states. Other places contain at most n tokens. Capacity constraints are defined in the model, but place invariants are not.

We have constructed the full reachability graph of this model using breadth-first search and five different approaches:

1. an interpreter implemented in C++ code
2. generated code
3. generated code with balanced trees
4. generated code with transition instance caches
5. generated code with balanced trees and transition instance caches

The time columns in Tables 1 and 2 correspond to these five approaches. The times were obtained on a 266 MHz Pentium II system with the C and C++ compilers from GCC 2.95.2 [1]. The time to compile and link the generated code ranges from about 7 to 10 seconds, and it is included in the figures.

Table 1: Exhaustive Analysis Times for the Distributed Data Base Model

n	States	Arcs	Seconds to Generate the Graph				
1	2	2	0.00	7.25	7.35	7.41	7.68
2	7	8	0.02	8.51	8.72	8.81	8.96
3	28	42	0.04	8.73	8.98	9.05	9.24
4	109	224	0.17	8.96	9.08	9.15	9.39
5	406	1,090	0.96	9.61	9.79	9.79	10.05
6	1,459	4,872	5.60	10.93	11.13	11.22	11.48
7	5,104	20,426	30.42	16.72	16.77	17.57	17.60
8	17,497	81,664	156.00	43.34	41.19	46.37	44.35
9	59,050	314,946	752.61	165.67	150.35	183.64	164.52
10	196,831	1,181,000	3,548.29	760.48	644.66	832.02	711.90

The figures reveal many interesting things. It is rather surprising to see that balancing the trees containing the multi-sets yields a slower result on models containing few items in the places. Since our implementation uses lazy deletion, only one operation is slower on balanced

Table 2: Exhaustive Analysis Times for the Model Excluding the Place **unused**

n	States	Arcs	Seconds to Generate the Graph				
1	2	2	0.02	6.90	6.95	7.06	7.17
2	7	8	0.02	7.91	8.11	8.13	8.20
3	28	42	0.04	8.05	8.19	8.25	8.37
4	109	224	0.11	8.14	8.35	8.31	8.39
5	406	1,090	0.51	8.61	8.79	8.91	8.88
6	1,459	4,872	2.30	9.46	9.67	9.72	9.82
7	5,104	20,426	10.05	12.93	13.04	13.27	13.45
8	17,497	81,664	41.63	25.86	26.38	26.80	27.28
9	59,050	314,946	168.68	77.31	78.86	81.04	81.84
10	196,831	1,181,000	654.74	272.62	277.08	283.79	290.43

trees: insertion. Because the state encoder processes the tokens in places in order, unbalanced trees will almost always degenerate to linked lists. However, this does not start to show up until the places have tens of tokens.

We can also see that using the transition instance cache slows down the analysis. The reason is simple: all cache look-ups always fail on this model. The cache could work better if some transitions were independent of each other. We experimented with a simple model consisting of two independent transitions to see how well the cache performs. In our test setting, one transition has *i* constantly enabled instances that generate self-loops to the reachability graph, while the other transition has one enabled instance that changes the state. With $i = 25$, there is a slight advantage of using the cache. With $i = 121$, the difference is more noticeable but still subtle. However, since managing the cache takes only a few lines of generated code, we decided to leave the code there, so that it can be enabled as a compile-time option.

4 Conclusion

Compiler techniques have precious applications in reachability analysers for high-level models. Compact notations, such as the multi-set summation and universal and existential quantification we have implemented in MARIA, can be invaluable e.g. when modelling multi-point communications [14].

Adding expressibility to a formalism should never have a negative effect on the applicability of different analysis techniques. Translating powerful short-hand notations to a lower-level internal representation keeps the core semantics of the formalism simple and allows existing analysis techniques to be used. The user can work with a compact model; he does not need to be aware of the transformations.

Exhaustive reachability analysis of high-level models with millions of states can be sped up considerably by transforming the models to machine code that analyses them. Alas, compiling large collections of code sometimes consumes much more time than executing the code. This practical problem—faced by everyone who is developing big models in an iterative process—can be solved by incremental compilation and simulation in an interpreter-based environment.

Acknowledgements

The author would like to thank Dr. Kimmo Varpaaniemi, who implemented the initial version of the transition instance analysis algorithm in MARIA, for fruitful discussions and ideas for improving the performance of the implementation.

References

[1] Free Software Foundation, Boston, MA, USA. *Using and Porting GNU CC, for Version 2.95*, 1999.

[2] Free Software Foundation, Boston, MA, USA. *Debugging with GDB: The GNU Source-Level Debugger, for Version 5*, 2000.

[3] Hartmann J. Genrich and Kurt Lautenbach. The analysis of distributed systems by means of Predicate/Transition-Nets. In Gilles Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 123–146, Evian, France, July 1979. Springer-Verlag, Berlin, Germany, 1979.

[4] Hartmann J. Genrich. Predicate/Transition Nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and their Properties—Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247, Bad Honnef, Germany, September 1986. Springer-Verlag, Berlin, Germany, 1987.

[5] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[6] Jean-Michel Ilié and Omar Rojas. On well-formed nets and optimizations in enabling tests. In Marco Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993: 14th International Conference*, volume 691 of *Lecture Notes in Computer Science*, pages 300–318, Chigago, IL, USA, June 1993. Springer-Verlag, Berlin, Germany.

[7] *Information Technology—Programming Languages—C++*. ISO/IEC 14882. International Organization for Standardization, Geneva, Switzerland, 1998.

[8] *Information Technology—Programming Languages—C*. ISO/IEC 9899. International Organization for Standardization, Geneva, Switzerland, 1999.

[9] Kurt Jensen. Coloured Petri Nets and the invariant method. *Theoretical Computer Science*, 14(3):317–336, June 1981.

[10] Ekkart Kindler and Hagen Völzer. Flexibility in algebraic nets. In Jörg Desel and Manuel Silva, editors, *Application and Theory of Petri Nets 1998: 19th International Conference, ICATPN'98*, volume 1420 of *Lecture Notes in Computer Science*, pages 345–364, Lisbon, Portugal, June 1998. Springer-Verlag, Berlin, Germany.

[11] Marko Mäkelä. *Maria—Modular Reachability Analyzer for Algebraic System Nets*. On-line documentation, ⟨URL:http://www.tcs.hut.fi/maria/⟩.

[12] Marko Mäkelä. *A Reachability Analyser for Algebraic System Nets*. Licentiate's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland, March 2000.

[13] Marko Mäkelä. Condensed storage of multi-set sequences. In *Workshop on the Practical Use of High-Level Petri Nets*, Århus, Denmark, June 2000.

[14] Leo Ojala, Nisse Husberg and Teemu Tynjälä. Modelling and analysing a distributed dynamic channel allocation algorithm for mobile computing using high-level net methods. In *Workshop on the Practical Use of High-Level Petri Nets*, Århus, Denmark, June 2000.

[15] Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen and Tino Pyssysalo. PROD reference manual. Technical Report B13, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, Espoo, Finland, August 1995.