Generating inductive verification proofs for Isabelle using the partial evaluator Ecce

Helko Lehmann and Michael Leuschel
mal@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-2002-2
September 2002

www.dsse.ecs.soton.ac.uk/techreports/

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

Generating inductive verification proofs for Isabelle using the partial evaluator Ecce

Helko Lehmann and Michael Leuschel

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
{hel99r,mal}@ecs.soton.ac.uk

Abstract. ECCE is a partial deduction system which can be used to automatically generate abstractions for the model checking of many infinite state systems. We show that to verify the abstractions generated by ECCE we may employ the proof assistant ISABELLE. Thereby ECCE is used to generate the specification, hypotheses and proof script in ISABELLE's theory format. Then, in many cases, ISABELLE can automatically execute these proof scripts and thereby verify the soundness of ECCE's abstraction. In this work we focus on the specification and verification of Petri nets.

1 Introduction

In recent work it has been shown that logic programming based methods can be applied to model checking of infinite state systems. One of the key issues of model checking of infinite systems is *abstraction* [2]. Abstraction allows to approximate an infinite system by a finite one, and if proper care is taken the results obtained for the finite abstraction will be valid for the infinite system. However, an important question when attempting to perform infinite model checking in practice is: How can one *automatically* obtain an abstraction which is finite, but still as precise as required to check a particular dynamic property? A potential solution to this problem is to apply existing techniques for the *automatic* control of *(logic) program specialisation*, [8] and many others. The aim of program specialisation is to improve the efficiency of a program by pre-evaluating it for particular parameters. In the context of logic programming program specialisation is also called *partial deduction*. Similar to model checking of infinite systems, in program specialisation in general and partial deduction in particular, one faces the following (quite extensively studied) problem: To be able to produce efficient specialised programs, *infinite* computation trees have to be abstracted in a *finite* but also as *precise* as possible way.

To apply partial deduction the system to be verified has to be modelled as a logic program by means of an interpreter [6, 11]. Thereby, the interpreter describes how the states of the system change by executing transitions. By applying partial deduction to the interpreter we expect a finite abstraction of the possibly infinite state space of the system to be generated. This abstraction may then be used to verify system properties of interest. This approach proved to be quite powerful as it was possible to obtain decision procedures for the coverability problem, if "typical" specialisation algorithms, as for example implemented in the ECCE system [14, 12], are applied to logic programs that encode Petri nets [10]. It is even possible to precisely mimic well known Petri net algorithms (by Karp–Miller [7] and by Finkel [3]) when the program specialisation techniques are slightly weakened. The results of [10] refer to *forward* algorithms only, i.e. algorithms which construct, beginning from some initial state, an abstract representation of the whole reachability tree of a Petri net. However, for some classes of systems such exhaustive

algorithms are not necessary or even not precise enough to decide coverability [1, 4, 5]. In such cases partial deduction may often be successfully applied as well [9], thereby mimicking well known *backward algorithms* [4].

Technically, the dynamic system specified in the input for the partial deduction algorithm can also be viewed as an inductive system describing the set of finite behaviours, i.e. the set of finite *paths*. Thereby, the set of initial states form the inductive base and each transition represents an inductive step. For the output of the partial deduction algorithm to be a sound abstraction each of the states reachable by a path must be contained in a state representation of the output. It is desirable to verify this property if we cannot ensure that the partial deduction algorithm is correctly implemented. The goal of this work is to show that such proofs can be generated and executed automatically. To this end we employ the partial deduction system ECCE for the automatic generation of the theory and the proof scripts. The proof assistant ISABELLE [15] is used to execute the proof scripts.

If we can use ISABELLE to verify the soundness of the output of the partial deduction method we may also ask whether it is possible to generate the hypotheses automatically and thereby use ISABELLE directly as a model checker of infinite systems. To this end, similar to the partial deduction system, ISABELLE needs to perform some kind of abstraction while searching for a proof of some dynamic property such as safety.

In this paper we focus on the specification and verification of Petri nets. This is due to their simple representation as a logic program as well as in a ISABELLE theory. The following section describes how we can specify Petri nets in ISABELLE. Then we discuss how such specifications are generated using ECCEand how ECCE output can be translated into ISABELLE. In Section 4 we demonstrate how proof scripts can be used in ISABELLE for automatic theorem proving. In Section 5 we demonstrate the complete verification process using an example specification. The above mentioned automatic generation of hypotheses and some efficiency issues are discussed in Section 6. The last section gives a conclusion and proposes some further work. All relevant source code of the ECCE system can be found in the appendix.

2 Specification of Petri nets in ISABELLE

The proof assistant ISABELLE has been developed as a generic system for implementing logical formalisms. Instead of developing an all new logic for our purposes we will use the specification and verification methods realised by the implementation of Higher Order Logic (HOL) in ISABELLE. HOL allows to express most mathematical concepts and, in contrast to, for example, First Order Logic, it allows the specification of and the reasoning about inductively defined sets. This latter feature is crucial for our purposes. Hence, strictly speaking, we will develop specifications in ISABELLE/HOL. Furthermore, the current ISABELLE system provides the language ISAR for the specification of theories and the development of proof scripts. In this work we will use ISAR instead of ISABELLE's implementation language ML since ISAR is much easier to use as it hides most implementation details of ISABELLE. However, the possibilities to develop proof tactics using ISAR only are very limited. Consequently we conjecture that for efficient automatic theorem proving the use of ISAR allone is insufficient (see also Section 7).

ISABELLE allows specifications as part of *theories*. A theory can be thought of as a collection of *declarations*, *definitions*, and *proofs*. ISABELLE/HOL is a typed logical language where the *base types* resemble those of functional programming languages such as ML. To specify new types ISABELLE provides *type constructors*, *function types*, and *type variables*. We will introdce the

particular concepts as we will use them and refer for additional information to the *Isabelle/Isar Reference Manual*[1].

Terms are formed by applying functions to arguments, e.g. if f is a function of type $\tau_1 \Rightarrow \tau_2$ and t a term of type τ_1 then ft is a term of type τ_2.

Formulas are terms of base type `bool`. Accordingly, the usual logical operators are defined as functions whose arguments and domain are of type `bool`.

We specify the Petri net theory `PN` as a successor of the theory `Main` which is provided by Isabelle/HOL. `Main` contains a number of basic declarations, definitions, and lemmas concerning often required basic concepts such as lists and sets. Thereby, every part of the theory `Main` becomes automatically visible in `PN`:

```
theory PN = Main:
```

To simplify the specification and to increase readability of the theory we define the type `state` which corresponds to a notion in Petri net theory: A *state* or *marking* is a vector of natural numbers representing the number of *tokens* on the *places* of a Petri net. The number of dimensions of the vector corresponds to the number of places of the particular net. In Isabelle we use the type constructor $\times$ to define the type `state` as a product over the base type `nat`:

```
types
 state = "nat × nat ×...× nat"
```

Based on the type `state` we declare the functions `paths`, `trans`, and `start`. The function `start` represents the *initial state* of the Petri net. Note that since we allow parameters in the definition of `state` it actually may represent a set of initial states. The function `trans` describes how the firing of a *transition* can change the state of a Petri net. The additional parameter of type `nat` is used to refer to a particular transition of the net. The set of finite possible sequences of transitons starting in the initial state is represented by `paths`. Note that the declaration of `trans` and *paths* is independent of the particular considered Petri net.

```
consts
 start :: "nat ⇒ ...⇒ nat ⇒ state"
 trans :: "(state × state × nat) set"
 paths :: "(state list) set"
```

By assigning a unique number the transition names are defined as a of enumeration type. Consequently, for each transition t we include a declaration of the following form:

```
consts
 t :: "nat"
```

The initial state `start` is defined by a term *term* of type state:

```
defs
 start_def [simp]: "start list of variables ≡ term"
```

The optional `[simp]` controls the strategy of Isabelle's built-in simplifier to apply this definition whenever possible. For our purposes *term* will be always a tuple of terms built using the unary successor function `Suc`, 0, and variables appearing in the *list of variables* (the number of variables in this list must correspond to the number of parameters in the declaration of `start`.

[1] Lawrence C. Paulson. The Isabelle Reference Manual. http://isabelle. in.tum.de/doc/ref.pdf.

The transition function is defined as a set of transitions of the Petri net. Thereby each transition is represented as a tuple `(x,y,n)`, where x and y are tuples of terms built by `Suc` and variables of the corresponding *list of variables*. The term n is the name of the transition.

```
defs
 trans_def: "trans ≡ {(x,y,n).
                    (∃ list1 of variables. (x,y,n)= transition1
                    ∨ (∃ list2 of variables. (x,y,n)= transition2
                    ⋮
                    ∨ (∃ listn of variables. (x,y,n)= transitionn}"
```

One of the important features of Isabelle/HOL is the possibility of inductive definitions. We define `paths` inductively using the following two rules:

```
inductive paths
intros
 zero: "[(start list of variables)] ∈ paths"
 step: "⟦(y,z,n) ∈ trans; y#l ∈ paths⟧ ⟹ z#(y#l) ∈ paths"
```

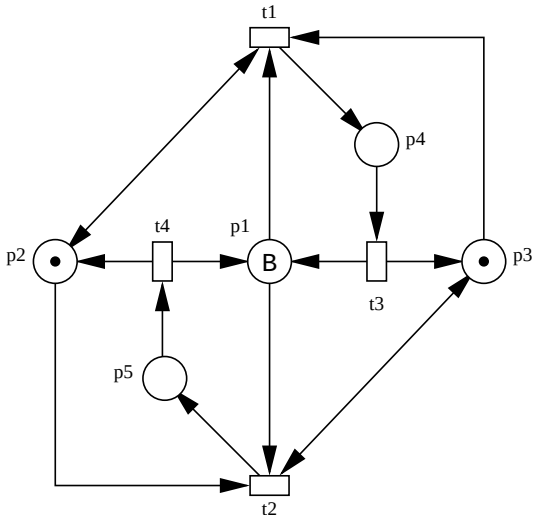
The first rule defines all initial states to be paths. The second rule allows the construction of new paths by extending an arbitrary path by a new state if there exists a transition from the state at the head of the path to the new state.

Finally, each transition t is defined as follows, where n is a unique natural number:

```
defs
 t_def [simp]: "t ≡ n"
```

The following example shows the the specification of a Petri net according to this scheme.

Example 1. We encode the Petri net depicted below in Isabelle/HOL. The initial state is defined by one token on each of the places $p2$ and $p3$, and the parameter `A` representing an arbitrary number of tokens on place $p1$ ($p1$, $p2$, $p3$ correspond to the first, second, and third dimension, respectively, of the state vector.



```
theory PN = Main:

types
 state = "nat × nat × nat × nat × nat"

consts
 start :: "nat ⇒ state"
 trans :: "(state × state × nat) set"
 paths :: "(state list) set"

 t1 :: "nat"
 t2 :: "nat"
 t3 :: "nat"
 t4 :: "nat"

defs
 start_def [simp]: "start ≡ (B,(Suc 0),(Suc 0),0,0)"

 trans_def: "trans ≡ {(x,y,n).
                  (∃ E D C B A. (x,y,n)=(((Suc A),(Suc B),(Suc C),D,E),
                                        (A,(Suc B),C,(Suc D),E),t1))
                  ∨ (∃ E D C B A. (x,y,n)=(((Suc A),(Suc B),(Suc C),D,E),
                                          (A,B,(Suc C),D,(Suc E)),t2))
                  ∨ (∃ E D C B A. (x,y,n)=((A,B,C,(Suc D),E),
                                          ((Suc A),B,(Suc C),D,E),t3))
                  ∨ (∃ E D C B A. (x,y,n)=((A,B,C,D,(Suc E)),
                                          ((Suc A),(Suc B),C,D,E),t4))}"

 t1_def [simp]: "t1 ≡ 0"
 t2_def [simp]: "t2 ≡ 1"
 t3_def [simp]: "t3 ≡ 2"
 t4_def [simp]: "t4 ≡ 3"

inductive paths
intros
 zero: "[(start B)] ∈ paths"
 step: "⟦(y,z,n) ∈ trans; y#l ∈ paths⟧ ⟹ z#(y#l) ∈ paths"
```

□

3 Generating Isabelle theories using Ecce

Since we aim to verify the partial deduction results of Ecce, we have integrated the generation of the Isabelle theory directly into Ecce. The generated Isabelle theory consists of three parts:

1. the specification of the Petri net,
2. the specification of the coverability graph as generated by Ecce,
3. the lemma to be verified together with a proof script.

In this section we deal with the first two parts while the third part is discussed in Section 4.

3.1 Generating Petri net specifications from logic programs

The Isabelle theory generator integrated in Ecce assumes that the transitions of a Petri net are specified by a set of clauses of a ternary predicate. The first parameter represents a transition name, the second represents the set of states where the transition can be applied, and the third how the state changes if the transition is executed. Technically, the second and the third parameter of each clause are lists of the length corresponding to the number of places. Relying on unification, conditions and changes can be easily expressed. For example, the condition that at least two tokens are on place $p3$ in a Petri net with five places is expressed by the term `[X0,X1,s(s(X2)),X3,X4]` (thereby `s` can be interpreted as the successor function on natural numbers). Similarly, the state change can be expressed: the removal of one token on place $p3$ and generation of two tokens on $p1$ is represented as `[s(s(X0)),X1,s(X2),X3,X4]`.

The initial state is simply represented as a single clause where the last parameter must be a list of the length corresponding to the number of places. Each element of the list can be constructed using `0`, the unary function `s`, and variables.

Example 2. The following logic program encodes the Petri net of Example 1.

```
trans(t1,[s(X0),s(X1),s(X2),X3,X4],[X0,s(X1),X2,s(X3),X4]).
trans(t2,[s(X0),s(X1),s(X2),X3,X4],[X0,X1,s(X2),X3,s(X4)]).
trans(t3,[X0,X1,X2,s(X3),X4],[s(X0),X1,s(X2),X3,X4]).
trans(t4,[X0,X1,X2,X3,s(X4)],[s(X0),s(X1),X2,X3,X4]).

start([B,s(0),s(0),0,0]).
```

□

The implementation of the theory generator is part of the file "code_generator.pro" and can be found in Appendix A. The generation is initiated by a call to the clause `print_specialised_program_isa`. In a user dialog the name of the file containing the Petri net specification, and the names of the predicates representing transitions and initial state, respectively are determined. The Isabelle specification is generated by the subsequent calls of `print_isa_header`, `print_isa_type_decl`, `print_isa_path_decl(Data)`, and `print_isa_path_def(Data)` in the body of `print_specialised_program_isa`. For example, the Isabelle theory of Example 1 has been generated from the logic program of Example 2.

3.2 Generating specifications of the coverability graph from logic programs

To use partial deduction techniques for model checking we need to specify also the verification task as a logic program. To this end we may implement the satisfiability relation of some temporal logic as a logic program. However, the generation of a coverability graph (by partial deduction or other techniques) is not effective for all tasks that can be expressed with a powerful temporal logic. However, one of the tasks where it is effective is the checking of *safety properties*. To express safety properties we only require the definition of the *EU* operator of the temporal logic *CTL*:

```
infinite_model_check(basic_safety,Formula) :- start(_,S),
     Formula = sat(S,eu(true,p(unsafe))).
```

```
sat(E,p(P)) :- prop(E,P).

sat(E,eu(F,G)) :-
    sat_eu(E,F,G).

sat_eu(E,_F,G) :-
    sat(E,G).
sat_eu(E,F,G) :-
   sat(E,F), trans(_Act,E,E2), sat_eu(E2,F,G).
```

Depending on the safety property we are interested in we define when a state is considered to be unsafe. For example the clause `prop([X0,X1,X2,s(X3),s(X4)],unsafe)` defines a state of a Petri net to be unsafe when there exist at least one token on each of the places $p4$ and $p5$.

Note that simply calling the clause `infinite_model_check(basic_safety,Formula)` in Prolog would force the system to explore an infinite derivation. Due to the potentially infinite state space of a Petri net also methods like tabeling would be in general insufficient to deal with this problem.

Before we apply the partial deduction system ECCE we will first perform a preliminary compilation of the particular Petri net and task. Thereby we will get rid of some of the interpretation overhead and achieve a more straightforward equivalence between markings of the Petri net and atoms encountered during the partial deduction phase. We will use the LOGEN offline partial deduction system [13] to that effect (but any other scheme which has a similar effect can be used). This system allows one to annotate calls in the original program as either reducible (executed by LOGEN) or non-reducible (not executed and thus kept in the specialised program).[2] In our case we will annotate all calls to `trans` and `start` as reducible. After that, the LOGEN system will (efficiently) produce a compiled version: As can be seen in Example 3, the compilation gives us a predicate `sat_eu__2` with one argument each for the transition name and the result, plus one argument per Petri net place. Observe that LOGEN (and ECCE as well) adds two underscores and a unique identifier to existing predicate names. `sat_eu__2` contains one clause per transition of the Petri net plus one fact (for the marking reached). The initial marking is encoded in the one clause for `ssat__0` which calls `sat__1`.

Example 3. Applying LOGEN to the Petri net specification of Example 2 and the above task implementation generates the following clauses:

```
sat_eu__2(B,C,D,s(E),s(F)).
sat_eu__2(s(G),s(H),s(I),J,K) :-
  sat_eu__2(G,s(H),I,s(J),K).
sat_eu__2(s(L),s(M),s(N),O,P) :-
  sat_eu__2(L,M,s(N),O,s(P)).
sat_eu__2(Q,R,S,s(T),U) :-
  sat_eu__2(s(Q),R,s(S),T,U).
sat_eu__2(V,W,X,Y,s(Z)) :-
  sat_eu__2(s(V),s(W),X,Y,Z).
sat__1(B,C,D,E,F) :-
  sat_eu__2(B,C,D,E,F).
```

[2] LOGEN is offline: the control decisions have been taken beforehand (and are encoded in the annotations).

```
ssat__0 :-
  sat__1(B,s(0),s(0),0,0).
```

□

After this precompilation we can apply ECCE to the resulting program. To this end we aim to specialise the predicate `ssat__0`. The result of applying ECCE to the program of Example 3 is given in Example 4:

Example 4.

```
ssat__0 :-
    ssat__0__1.

/* ssat__0__1 --> [ssat__0] */
ssat__0__1 :-
    sat__1__2(A).

/* sat__1__2(A) --> [sat__1(A,s(0),s(0),0,0)] */
sat__1__2(A) :-
    sat_eu__2__3(A).

/* sat_eu__2__3(A) --> [sat_eu__2(A,s(0),s(0),0,0)] */
sat_eu__2__3(s(A)) :-
    sat_eu__2__4(A).
sat_eu__2__3(s(A)) :-
    sat_eu__2__5(A).

/* sat_eu__2__4(A) --> [sat_eu__2(A,s(0),0,s(0),0)] */
sat_eu__2__4(A) :-
    sat_eu__2__3(s(A)).

/* sat_eu__2__5(A) --> [sat_eu__2(A,0,s(0),0,s(0))] */
sat_eu__2__5(A) :-
    sat_eu__2__3(s(A)).
```

□

From the output of ECCE we generate an ISABELLE theory representing the generated coverability relation. Independent of the particular domain this relation is declared as a set of pairs of states:

```
consts
 coverrel:: "(state × state) set"
```

For each predicate name of a clause in the specialised program, which represents a set of states we add a declaration of the form:

```
consts
 name :: nat ⇒ ...⇒ nat ⇒ state"
```

Thereby the number of parameters of type `nat` corresponds to the number of variables in the head of the clause. The definitions have the form:

```
defs
 name_def: "name list of variables ≡ term"
```

For our purposes *term* will be always a tuple of terms built using the unary successor function `Suc`, 0, and variables appearing in the *list of variables* (the number of variables in this list must correspond to the number of parameters in the declaration of *name*).

Finally, the coverability relation is defined as a set of pairs of states. In the specialised program every clause of the form $name_m(args_m)$ `:-` $name_n(args_n)$ corresponds to such a pair. Formally, in the ISABELLE theory each pair is represented as a tuple `(x,y)`, where x and y are tuples of terms built by `Suc` and variables of the corresponding *list of variables*:

```
defs
 coverrel_def: "coverrel ≡
                      {(x,y). ∃ list1 of variables. x= state11 ∧ y= state12}
                    ∪ {(x,y). ∃ list2 of variables. x= state21 ∧ y= state22}
                    ⋮
                    ∪ {(x,y). ∃ listm of variables. x= statem1 ∧ y= statem2}"
```

The theory generator of Appendix A produces automatically the specification of the coverability relation from the specialised program. To this end the predicate names characterising the coverability relation in the specialised program are determined by a user dialog (only the unspecialised names have to be provided, e.g. in the above example `sat__1` and `sat_eu__2`). In the body of `print_specialised_program_isa` the calls to `print_isa_cover_decl` and `print_isa_cover_def` generate the necessary declarations and definitions, respectively.

Example 5. The following theory was generated by the theory generator of Appendix A from the program of Example 4:

```
consts
 coverrel:: "(state × state) set"
 sat__1__2 :: "nat ⇒ state"
 sat_eu__2__3 :: "nat ⇒ state"
 sat_eu__2__4 :: "nat ⇒ state"
 sat_eu__2__5 :: "nat ⇒ state"
defs
 sat__1__2_def: "sat__1__2 A ≡ (A,(Suc 0),(Suc 0),0,0)"
 sat_eu__2__3_def: "sat_eu__2__3 A ≡ (A,(Suc 0),(Suc 0),0,0)"
 sat_eu__2__4_def: "sat_eu__2__4 A ≡ (A,(Suc 0),0,(Suc 0),0)"
 sat_eu__2__5_def: "sat_eu__2__5 A ≡ (A,0,(Suc 0),0,(Suc 0))"
 coverrel_def: "coverrel ≡ {(x,y). ∃ A. x=(sat__1__2 A)
                                      ∧ y=(sat_eu__2__3 A)}
                           ∪ {(x,y). ∃ A. x=(sat_eu__2__3 (Suc A))
                                      ∧ y=(sat_eu__2__4 A)}
                           ∪ {(x,y). ∃ A. x=(sat_eu__2__3 (Suc A))
                                      ∧ y=(sat_eu__2__5 A)}
```

```
∪ {(x,y). ∃ A. x=(sat_eu__2__4 A)
                ∧ y=(sat_eu__2__3 (Suc A))}
∪ {(x,y). ∃ A. x=(sat_eu__2__5 A)
                ∧ y=(sat_eu__2__3 (Suc A))}"
```

□

4 Proof Scripts

In this section we demonstrate how we can prove theorems using Isabelle/Isarand how we can write proof scripts for automatic execution. Thereby we focus only on some of the "execution style" proof commands of Isabelle/Isar. These commands can be considered to be the classical way of writing Isabelle proofs although the actual Isabelle proof methods are wrapped within the Isar language. Note however that Isar allows also a more "mathematical style" notation of proofs than the one we use here (see the *Isabelle/Isar Reference Manual* for details).

Furthermore we discuss only the proof methods we are going to apply in order to solve the verification task of Ecce. Keep in mind that Isabelle/Isar provides a much wider range of methods.

The proof mode of Isabelle/Isar is initiated by executing a `lemma`. When entering the proof mode Isabelle/Isar generates a single pending subgoal consisting of the lemma to be proven. The list of subgoals can be altered, mainly by executing *proof methods*. Proof methods are executed using the proof command `apply`. Thereby the list of subgoals defines the *proof state*. The proof mode can be left by executing `done` in the case that there are no pending subgoals (the proof state is the empty list of subgoals, in which case Isabelle/Isar prints `No subgoals!`).

Note that all proof methods described below only transform the first subgoal of the proof state. For finding a proof this may be inconvenient. Therefore, Isabelle/Isar provides commands to change the order of the subgoals. However, our aim in this paper is the automatic execution of proof scripts, not their interactive development.

4.1 Rewriting

To rewrite a subgoal using existing definitions and lemmas automatically we may execute Isabelle's simplifier: `apply(simp)`. For the simplifier to automatically attempt to use new defintions and lemmas they have to be accompanied by the option `[simp]`. Such defined simplification rules are then applied from left to right. However, we have to take care if we define simplification rules in such a way as they may slow the simplifier down considerably or even cause it to loop. Instead of defining a general simplification rule we may also use the simplifier to only apply certain, explicitely stated definitions. E.g., the execution `apply(simp only:` *r*`_def)` causes to rewrite using the definition of `r` only.

4.2 Introduction and Elimination

Based on reasoning using *natural deduction* there are two types of rules for each logical symbol, such as ∨: *introduction rules* which allow us to infer formulas containing the symbol (e.g. ∨),

and *elimination rules* which allow us to deduce consequences of a formula containing the symbol (e.g. $\vee$).

In ISABELLE an introduction rule is usually applied by `apply(rule r)`. Assume r being a rule of the form:

$$\frac{P_1, \ldots, P_n}{Q}$$

where Q is a formula containing the introduced logical symbol while the formulas P_1,...,P_n in the premise do not. Then, if r is applied as introduction rule the current first subgoal is unified with Q and replaced by the properly instantiated P_1,...,P_n.

An elimination rule is usually applied using `apply(erule r)`. Assume r being a rule of the above form and the current first subgoal of the form $A_1, \ldots, A_m \Longrightarrow S$. Then, if r is applied as elimination rule S is unified with Q and some A_i is unified with P_1. The old subgoal is replace by $n-1$ new subgoals of the form $A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m \Longrightarrow P_k$ with $2 \leq k \leq n$.

In our verification proofs we will use explicitely only the elimination rules `disjE` for disjunction and `paths.induct` for induction over the length of paths.

4.3 Automatic Reasoners

Most classical reasoning of even simple lemmas can require the application of a vast amount of rules. To simplify this task ISABELLE provides a number of automatic reasoners. Here we will make use of `blast` which is the most powerful of ISABELLE's reasoners. Additionally, we will employ `clarify` which performs obvious transformations which do not require to split the subgoal or render it unprovable. The method `clarify` and the explicit application of the elimination rule `disjE` (see above))) was necessary to tune the proof process. This tuning was necessary to complete the verification proofs of even very small Petri nets using the available computing resources.

Additionally to the two classical reasoners we also employ the simplifier `simp` as an automatic proof tool as it can also handle some arithmetics. Furthermore, for some cases in our verification task `simp` succeeded faster than `blast` if it was able to eliminate a subgoal at all.

4.4 Scripts

To improve the handling of large proofs and to allow a higher flexibility of a proof proof scripts can be extended by the following operators:

- *method*$_1$`,...,`*method*$_n$: a list of methods represents their sequential execution;
- `(`*method*`)`: mainly used to define the scope of another operator;
- *method*`?`: executes *method* only if it does not fail,
- *method*$_1$`|...|`*method*$_n$: attempts to execute *method*$_k$ only if each *method*$_i$ with $i < k$ failed;
- *method*`+`: *method* is repeatedly executed until it fails.

For our verification task the lemma and proof script are generated automatically by the theory generator of Appendix A (by calls to `print_isa_lemma` and `print_isa_proofscript` in the body of `print_specialised_program_isa`). The execution of the script in the example below is illustrated in the next section.

Example 6. The following lemma and script corresponds to the one automatically generated by ECCE for the Petri net specification of Example 1:

```
lemma "l ∈ paths ⟹ ∃ y. ((hd l),y) ∈ coverrel"

 apply(erule paths.induct)
  apply(simp only: start_def
                   coverrel_def)
  apply(simp only: sat__1__2_def
                   sat_eu__2__3_def
                   sat_eu__2__4_def
                   sat_eu__2__5_def)
  apply(simp)
  apply(blast)
 apply(simp only:trans_def)
 apply(clarify)
 apply(((erule disjE)?,
          simp only: coverrel_def,
          simp,
          ((erule disjE)?,
             simp only: sat__1__2_def
                        sat_eu__2__3_def
                        sat_eu__2__4_def
                        sat_eu__2__5_def,
             simp|blast)+)+)
```

□

5 Verifying ECCE

In this section we illustrate the automatic verification of the ECCE output by the ISABELLE system. To this end the theory, lemma and proof script as generated by ECCE for the Petri net of Example 1, are executed (the complete input consists of the ISABELLE specifications of Example 1, Example 5, and lemma and proof script of Example 6).

Example 7. For the line of the theory starting with `lemma`:

```
lemma "l ∈ paths ⟹ ∃ y. ((hd l),y) ∈ coverrel"
```

ISABELLE switches to the proof mode and creates a subgoal for the proposition of the lemma. Some very basic simplifications are performed (here the brackets surrounding `(hd l)` are removed):

```
goal (lemma, 1 subgoal):
l ∈ paths ⟹ ∃ y. (hd l, y) ∈ coverrel
 1. l ∈ paths ⟹ ∃ y. (hd l, y) ∈ coverrel
```

As the lemma makes a proposition for all paths and the set of paths is inductively defined, it is a reasonable approach to attempt induction as a proof method. For every inductively defined concept ISABELLE automatically generates the appropriate induction axiom, e.g. for `paths` this axiom is called `paths.induct`. We apply this axiom as an elimination rule (i.e. `erule`):

```
apply(erule paths.induct)
```

Accordingly Isabelle eliminates the above lemma by two subgoals. The first subgoal represents the initial induction condition. In our case the subgoal claims that for any substitution of `A` the lemma holds for the path consisting only of the initial marking. The second lemma represents the induction step. Here it claims, that whenever the lemma holds for some path then it also holds for the path extended by an additional transition.

```
 1. ⋀A. ∃ y. (hd [start A], y) ∈ coverrel
 2. ⋀l n y z.
       ⟦(y, z, n) ∈ PN.trans; y # l ∈ paths;
          ∃ ya. (hd (y # l), ya) ∈ coverrel⟧
       ⟹ ∃ ya. (hd (z # y # l), ya) ∈ coverrel
```

To solve the first subgoal we apply the definitions of the initial state and the coverability relation using Isabelle's simplifier. Restricting the simplifier to apply certain definitions only makes it often significantly faster and can prevent loops. However, less steps are handled automatically.

```
apply(simp only: start_def
                 coverrel_def)
```

As expected, Isabelle unfolds the given definitions. Since the second subgoal stays unchanged if `simp` is applied we do not represent it here.

```
 1. ⋀A. ∃ y. (hd [(A, Suc 0, Suc 0, 0, 0)], y)
             ∈ {(x, y). ∃ A. x = sat__1__2 A ∧ y = sat_eu__2__3 A} ∪
               {(x, y). ∃ A. x = sat_eu__2__3 (Suc A) ∧ y = sat_eu__2__4 A} ∪
               {(x, y). ∃ A. x = sat_eu__2__3 (Suc A) ∧ y = sat_eu__2__5 A} ∪
               {(x, y). ∃ A. x = sat_eu__2__4 A ∧ y = sat_eu__2__3 (Suc A)} ∪
               {(x, y). ∃ A. x = sat_eu__2__5 A ∧ y = sat_eu__2__3 (Suc A)}
 2. ...
```

Again, we have to unfold definitions, in this case the state descriptions.

```
apply(simp only: sat__1__2_def
                 sat_eu__2__3_def
                 sat_eu__2__4_def
                 sat_eu__2__5_def)

 1. ⋀A. ∃ y. (hd [(A, Suc 0, Suc 0, 0, 0)], y)
             ∈ {(x, y).
                ∃ A. x = (A, Suc 0, Suc 0, 0, 0) ∧
                     y = (A, Suc 0, Suc 0, 0, 0)} ∪
               {(x, y).
                ∃ A. x = (Suc A, Suc 0, Suc 0, 0, 0) ∧
                     y = (A, Suc 0, 0, Suc 0, 0)} ∪
               {(x, y).
                ∃ A. x = (Suc A, Suc 0, Suc 0, 0, 0) ∧
                     y = (A, 0, Suc 0, 0, Suc 0)} ∪
```

```
                {(x, y).
                 ∃ A. x = (A, Suc 0, 0, Suc 0, 0) ∧
                      y = (Suc A, Suc 0, Suc 0, 0, 0)} ∪
                {(x, y).
                 ∃ A. x = (A, 0, Suc 0, 0, Suc 0) ∧
                      y = (Suc A, Suc 0, Suc 0, 0, 0)}
 2. ...
```

Note that the above subgoal does not contain any further definitions specific to our theory. Hence, we may attempt to simplify the subgoal using the built-in simplifier:

```
apply(simp)
```

The resulting subgoal is a disjunction of three states potentially matching the intial state. The simplification was successful in two ways: 1) Two pairs have been eliminated since the first element of the first three pairs in the coverability relation (see previous subgoal) are equivalent. None of the second elements of the pairs is relevant to prove the claim, hence the new subgoal does not refer to these elements.

```
 1. ⋀A. ∃ a aa ab ac b.
           a = A ∧ aa = Suc 0 ∧ ab = Suc 0 ∧ ac = 0 ∧ b = 0 ∨
           A = Suc a ∧ aa = Suc 0 ∧ ab = 0 ∧ ac = Suc 0 ∧ b = 0 ∨
           A = Suc a ∧ aa = 0 ∧ ab = Suc 0 ∧ ac = 0 ∧ b = Suc 0
 2. ...
```

The remaining subgoal can be verified by using Isabelle's automatic proof tactic `blast`:

```
apply(blast)
```

Here `blast` succeeds in proving the first subgoal and we continue with the induction step:

```
 1. ⋀l n y z.
       ⟦(y, z, n) ∈ PN.trans; y # l ∈ paths;
         ∃ ya. (hd (y # l), ya) ∈ coverrel⟧
       ⟹ ∃ ya. (hd (z # y # l), ya) ∈ coverrel
```

As a first step, we unfold the definition of the transition function using Isabelle's simplifier:

```
apply(simp only:trans_def)
```

The resulting subgoal contains a disjunction of possible transitions. When a path is extended by any of these transitions it has still to comply with the lemma.

```
 1. ⋀l n y z.
       ⟦(y, z, n)
        ∈ {(x, y, n).
           (∃ E D C B A.
                (x, y, n) =
                ((Suc A, Suc B, Suc C, D, E), (A, Suc B, C, Suc D, E), t1)) ∨
           (∃ E D C B A.
```

```
          (x, y, n) =
          ((Suc A, Suc B, Suc C, D, E), (A, B, Suc C, D, Suc E), t2)) ∨
      (∃ E D C B A.
          (x, y, n) =
          ((A, B, C, Suc D, E), (Suc A, B, Suc C, D, E), t3)) ∨
      (∃ E D C B A.
          (x, y, n) =
          ((A, B, C, D, Suc E), (Suc A, Suc B, C, D, E), t4))};
    y # l ∈ paths; ∃ ya. (hd (y # l), ya) ∈ coverrel⟧
⟹ ∃ ya. (hd (z # y # l), ya) ∈ coverrel
```

Some simple non-destructive transformations can be achieved with the automatic proof tactic `clarify`:

```
apply(clarify)
```

In our case the quantifier for `ya` in the premise has been pulled to the front and the variables representing tuples are split:

```
1. ⋀l n a aa ab ac b ad ae af ag ba ah ai aj ak bb.
      ⟦(a, aa, ab, ac, b) # l ∈ paths;
          (∃ E D C B A.
              ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
              ((Suc A, Suc B, Suc C, D, E), (A, Suc B, C, Suc D, E), t1)) ∨
          (∃ E D C B A.
              ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
              ((Suc A, Suc B, Suc C, D, E), (A, B, Suc C, D, Suc E), t2)) ∨
          (∃ E D C B A.
              ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
              ((A, B, C, Suc D, E), (Suc A, B, Suc C, D, E), t3)) ∨
          (∃ E D C B A.
              ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
              ((A, B, C, D, Suc E), (Suc A, Suc B, C, D, E), t4));
          (hd ((a, aa, ab, ac, b) # l), ah, ai, aj, ak, bb) ∈ coverrel⟧
⟹ ∃ y. (hd ((ad, ae, af, ag, ba) # (a, aa, ab, ac, b) # l), y)
          ∈ coverrel
```

Using the elimination rule `disjE` we split the disjunction:

```
apply(erule disjE)
```

Thereby we perform a case distinction to prove the claim for each possible transition. Isabelle generates a new subgoal for the first transition. In the second subgoal the first transition has been eliminated from the disjunction.

```
1. ⋀l n a aa ab ac b ad ae af ag ba ah ai aj ak bb.
      ⟦(a, aa, ab, ac, b) # l ∈ paths;
          (hd ((a, aa, ab, ac, b) # l), ah, ai, aj, ak, bb) ∈ coverrel;
          ∃ E D C B A.
              ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
              ((Suc A, Suc B, Suc C, D, E), (A, Suc B, C, Suc D, E), t1)⟧
```

```
    ⟹ ∃ y. (hd ((ad, ae, af, ag, ba) # (a, aa, ab, ac, b) # l), y)
          ∈ coverrel

2. ⋀l n a aa ab ac b ad ae af ag ba ah ai aj ak bb.
    ⟦(a, aa, ab, ac, b) # l ∈ paths;
       (hd ((a, aa, ab, ac, b) # l), ah, ai, aj, ak, bb) ∈ coverrel;
       (∃ E D C B A.
            ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
            ((Suc A, Suc B, Suc C, D, E), (A, B, Suc C, D, Suc E), t2)) ∨
       (∃ E D C B A.
            ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
            ((A, B, C, Suc D, E), (Suc A, B, Suc C, D, E), t3)) ∨
       (∃ E D C B A.
            ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
            ((A, B, C, D, Suc E), (Suc A, Suc B, C, D, E), t4))⟧
    ⟹ ∃ y. (hd ((ad, ae, af, ag, ba) # (a, aa, ab, ac, b) # l), y)
          ∈ coverrel
```

Similar to the case of the initial we unfold the definition of the coverability relation and perform some simplifications.

```
apply(simp only: coverrel_def, simp)
```

The simplifier has substituted the set representation by a disjunction in both, the premise and the conclusion. Furthermore, only the relevant parts of the transition (with the name `t1`) are kept:

```
1. ⋀l n a aa ab ac b ad ae af ag ba ah ai aj ak bb.
    ⟦a = Suc ad ∧
     (∃ B. aa = Suc B ∧
           ab = Suc af ∧ ae = Suc B ∧ ag = Suc ac ∧ ba = b ∧ n = 0);
       (Suc ad, aa, ab, ac, b) # l ∈ paths;
       (∃ A. (Suc ad, aa, ab, ac, b) = sat_1_2 A ∧
             (ah, ai, aj, ak, bb) = sat_eu_2_3 A) ∨
       (∃ A. (Suc ad, aa, ab, ac, b) = sat_eu_2_3 (Suc A) ∧
             (ah, ai, aj, ak, bb) = sat_eu_2_4 A) ∨
       (∃ A. (Suc ad, aa, ab, ac, b) = sat_eu_2_3 (Suc A) ∧
             (ah, ai, aj, ak, bb) = sat_eu_2_5 A) ∨
       (∃ A. (Suc ad, aa, ab, ac, b) = sat_eu_2_4 A ∧
             (ah, ai, aj, ak, bb) = sat_eu_2_3 (Suc A)) ∨
       (∃ A. (Suc ad, aa, ab, ac, b) = sat_eu_2_5 A ∧
             (ah, ai, aj, ak, bb) = sat_eu_2_3 (Suc A))⟧
    ⟹ ∃ a aa ab ac b.
             (∃ A. (ad, ae, af, ag, ba) = sat_1_2 A ∧
                   (a, aa, ab, ac, b) = sat_eu_2_3 A) ∨
             (∃ A. (ad, ae, af, ag, ba) = sat_eu_2_3 (Suc A) ∧
                   (a, aa, ab, ac, b) = sat_eu_2_4 A) ∨
             (∃ A. (ad, ae, af, ag, ba) = sat_eu_2_3 (Suc A) ∧
                   (a, aa, ab, ac, b) = sat_eu_2_5 A) ∨
             (∃ A. (ad, ae, af, ag, ba) = sat_eu_2_4 A ∧
```

```
                (a, aa, ab, ac, b) = sat_eu__2__3 (Suc A)) ∨
          (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__5 A ∧
                (a, aa, ab, ac, b) = sat_eu__2__3 (Suc A))
```

2. ...

We have to show that for every pair covering a path the path extended by transition `t1` is covered by some pair of the coverability relation in the conclusion. Hence, we apply again the elimination rule `disjE` on the disjunction in the premise of the subgoal.

```
apply((erule disjE)?)
```

Thereby we perform a case distinction on the possible pairs of the coverability relation and generate a new subgoal for the first pair while this pair is removed from the second subgoal:

```
1. ⋀l n a aa ab ac b ad ae af ag ba ah ai aj ak bb.
    ⟦a = Suc ad ∧
     (∃ B. aa = Suc B ∧
           ab = Suc af ∧ ae = Suc B ∧ ag = Suc ac ∧ ba = b ∧ n = 0);
        (Suc ad, aa, ab, ac, b) # l ∈ paths;
        ∃ A. (Suc ad, aa, ab, ac, b) = sat__1__2 A ∧
             (ah, ai, aj, ak, bb) = sat_eu__2__3 A⟧
    ⟹ ∃ a aa ab ac b.
          (∃ A. (ad, ae, af, ag, ba) = sat__1__2 A ∧
                (a, aa, ab, ac, b) = sat_eu__2__3 A) ∨
          (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__3 (Suc A) ∧
                (a, aa, ab, ac, b) = sat_eu__2__4 A) ∨
          (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__3 (Suc A) ∧
                (a, aa, ab, ac, b) = sat_eu__2__5 A) ∨
          (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__4 A ∧
                (a, aa, ab, ac, b) = sat_eu__2__3 (Suc A)) ∨
          (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__5 A ∧
                (a, aa, ab, ac, b) = sat_eu__2__3 (Suc A))

2. ⋀l n a aa ab ac b ad ae af ag ba ah ai aj ak bb.
    ⟦a = Suc ad ∧
     (∃ B. aa = Suc B ∧
           ab = Suc af ∧ ae = Suc B ∧ ag = Suc ac ∧ ba = b ∧ n = 0);
        (Suc ad, aa, ab, ac, b) # l ∈ paths;
        (∃ A. (Suc ad, aa, ab, ac, b) = sat_eu__2__3 (Suc A) ∧
              (ah, ai, aj, ak, bb) = sat_eu__2__4 A) ∨
        (∃ A. (Suc ad, aa, ab, ac, b) = sat_eu__2__3 (Suc A) ∧
              (ah, ai, aj, ak, bb) = sat_eu__2__5 A) ∨
        (∃ A. (Suc ad, aa, ab, ac, b) = sat_eu__2__4 A ∧
              (ah, ai, aj, ak, bb) = sat_eu__2__3 (Suc A)) ∨
        (∃ A. (Suc ad, aa, ab, ac, b) = sat_eu__2__5 A ∧
              (ah, ai, aj, ak, bb) = sat_eu__2__3 (Suc A)) ⟧
    ⟹ ∃ a aa ab ac b.
          (∃ A. (ad, ae, af, ag, ba) = sat__1__2 A ∧
                (a, aa, ab, ac, b) = sat_eu__2__3 A) ∨
          (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__3 (Suc A) ∧
```

```
                    (a, aa, ab, ac, b) = sat_eu__2__4 A) ∨
                (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__3 (Suc A) ∧
                    (a, aa, ab, ac, b) = sat_eu__2__5 A) ∨
                (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__4 A ∧
                    (a, aa, ab, ac, b) = sat_eu__2__3 (Suc A)) ∨
                (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__5 A ∧
                    (a, aa, ab, ac, b) = sat_eu__2__3 (Suc A))
 3. ...
```

Again, we unfold the definitions of the state representations:

```
apply(simp only: sat__1__2_def
                 sat_eu__2__3_def
                 sat_eu__2__4_def
                 sat_eu__2__5_def)
```

```
 1. ⋀l n a aa ab ac b ad ae af ag ba ah ai aj ak bb.
       ⟦a = Suc ad ∧
        (∃ B. aa = Suc B ∧
              ab = Suc af ∧ ae = Suc B ∧ ag = Suc ac ∧ ba = b ∧ n = 0);
          (Suc ad, aa, ab, ac, b) # l ∈ paths;
          ∃ A. (Suc ad, aa, ab, ac, b) = (A, Suc 0, Suc 0, 0, 0) ∧
               (ah, ai, aj, ak, bb) = (A, Suc 0, Suc 0, 0, 0)⟧
       ⟹ ∃ a aa ab ac b.
               (∃ A. (ad, ae, af, ag, ba) = (A, Suc 0, Suc 0, 0, 0) ∧
                     (a, aa, ab, ac, b) = (A, Suc 0, Suc 0, 0, 0)) ∨
               (∃ A. (ad, ae, af, ag, ba) = (Suc A, Suc 0, Suc 0, 0, 0) ∧
                     (a, aa, ab, ac, b) = (A, Suc 0, 0, Suc 0, 0)) ∨
               (∃ A. (ad, ae, af, ag, ba) = (Suc A, Suc 0, Suc 0, 0, 0) ∧
                     (a, aa, ab, ac, b) = (A, 0, Suc 0, 0, Suc 0)) ∨
               (∃ A. (ad, ae, af, ag, ba) = (A, Suc 0, 0, Suc 0, 0) ∧
                     (a, aa, ab, ac, b) = (Suc A, Suc 0, Suc 0, 0, 0)) ∨
               (∃ A. (ad, ae, af, ag, ba) = (A, 0, Suc 0, 0, Suc 0) ∧
                     (a, aa, ab, ac, b) = (Suc A, Suc 0, Suc 0, 0, 0))
 2. ...
 3. ...
```

To resolve the remaining subgoal we can rely on the automatic proof tactic `blast` of Isabelle.

```
apply(blast)
```

Here, `blast` has successfully removed the first subgoal:

```
 1. ⋀l n a aa ab ac b ad ae af ag ba ah ai aj ak bb.
       ⟦a = Suc ad ∧
        (∃ B. aa = Suc B ∧
              ab = Suc af ∧ ae = Suc B ∧ ag = Suc ac ∧ ba = b ∧ n = 0);
          (Suc ad, aa, ab, ac, b) # l ∈ paths;
          (∃ A. (Suc ad, aa, ab, ac, b) = sat_eu__2__3 (Suc A) ∧
```

```
            (ah, ai, aj, ak, bb) = sat_eu__2__4 A) ∨
       (∃ A. (Suc ad, aa, ab, ac, b) = sat_eu__2__3 (Suc A) ∧
            (ah, ai, aj, ak, bb) = sat_eu__2__5 A) ∨
       (∃ A. (Suc ad, aa, ab, ac, b) = sat_eu__2__4 A ∧
            (ah, ai, aj, ak, bb) = sat_eu__2__3 (Suc A)) ∨
       (∃ A. (Suc ad, aa, ab, ac, b) = sat_eu__2__5 A ∧
            (ah, ai, aj, ak, bb) = sat_eu__2__3 (Suc A)) ⟧
  ⟹ ∃ a aa ab ac b.
            (∃ A. (ad, ae, af, ag, ba) = sat__1__2 A ∧
                  (a, aa, ab, ac, b) = sat_eu__2__3 A) ∨
            (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__3 (Suc A) ∧
                  (a, aa, ab, ac, b) = sat_eu__2__4 A) ∨
            (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__3 (Suc A) ∧
                  (a, aa, ab, ac, b) = sat_eu__2__5 A) ∨
            (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__4 A ∧
                  (a, aa, ab, ac, b) = sat_eu__2__3 (Suc A)) ∨
            (∃ A. (ad, ae, af, ag, ba) = sat_eu__2__5 A ∧
                  (a, aa, ab, ac, b) = sat_eu__2__3 (Suc A))


2. ⋀l n a aa ab ac b ad ae af ag ba ah ai aj ak bb.
     ⟦(a, aa, ab, ac, b) # l ∈ paths;
       (hd ((a, aa, ab, ac, b) # l), ah, ai, aj, ak, bb) ∈ coverrel;
       (∃ E D C B A.
            ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
            ((Suc A, Suc B, Suc C, D, E), (A, B, Suc C, D, Suc E), t2)) ∨
       (∃ E D C B A.
            ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
            ((A, B, C, Suc D, E), (Suc A, B, Suc C, D, E), t3)) ∨
       (∃ E D C B A.
            ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
            ((A, B, C, D, Suc E), (Suc A, Suc B, C, D, E), t4))⟧
  ⟹ ∃ y. (hd ((ad, ae, af, ag, ba) # (a, aa, ab, ac, b) # l), y)
          ∈ coverrel
```

The above steps of performing a case distinction on the premise, unfolding the state definitions and attempting to apply `blast` can be automatised using the script below. Thereby the symbol `+` denotes the iteration of the sequence of proof commands within preceding brackets until the sequence fails. For the last case of the case distinction the application of the elimination rule `disjE` is not necessary and will fail. Consequently, we apply this rule only if it is possible, denoted by `?`. Finally, although `blast` is often more powerful than `simp`, it is also slower. By writing `simp|blast` we attempt first to prove the subgoal by simplification. Only if `simp` fails to resolve the goal we attempt the slower `blast` method.

```
apply(((erule disjE)?,
       simp only: sat__1__2_def
                  sat_eu__2__3_def
                  sat_eu__2__4_def
                  sat_eu__2__5_def,
```

```
        simp|blast)+)
```

Applying this script resolves the whole first subgoal concerning transition `t1`:

```
 2. ⋀l n a aa ab ac b ad ae af ag ba ah ai aj ak bb.
       ⟦(a, aa, ab, ac, b) # l ∈ paths;
           (hd ((a, aa, ab, ac, b) # l), ah, ai, aj, ak, bb) ∈ coverrel;
           (∃ E D C B A.
                ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
                ((Suc A, Suc B, Suc C, D, E), (A, B, Suc C, D, Suc E), t2)) ∨
           (∃ E D C B A.
                ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
                ((A, B, C, Suc D, E), (Suc A, B, Suc C, D, E), t3)) ∨
           (∃ E D C B A.
                ((a, aa, ab, ac, b), (ad, ae, af, ag, ba), n) =
                ((A, B, C, D, Suc E), (Suc A, Suc B, C, D, E), t4))⟧
       ⟹ ∃ y. (hd ((ad, ae, af, ag, ba) # (a, aa, ab, ac, b) # l), y)
                ∈ coverrel
```

Now we can also apply the steps required to prove the lemma for transition `t1` in a similar fashion to the remaining transitions. The following proof script attempts precisely this. Again, the elimination rule `disjE` is not applicable for the last transition. Hence, we perform a test using `?` before applying this method in the first line.

```
apply(((erule disjE)?,
        simp only: coverrel_def,
        simp,
        ((erule disjE)?,
          simp only: sat__1__2_def
                     sat_eu__2__3_def
                     sat_eu__2__4_def
                     sat_eu__2__5_def,
          simp|blast)+)+)
```

For our example all cases could be verified, hence Isabelle answers:

```
No subgoals!
```

□

Consequently, the coverability relation generated by Ecce for the Petri net of Example 1 covers indeed all states reachable by any path (under the condition that the theory generated by the automatic theory generator as implemented in Ecce is correct).

6 Automatic Generation of Hypotheses

Instead of defining the coverability as a relation as illustrated in Subsection 3.2 we may view the coverability graph as an inductive definition of a set of states which covers the actual state space of the Petri net. For our example a corresponding Isabelle/Isar definition could look as follows:

```
consts
 coverstates:: "state set"
inductive coverstates
intros
 zero : "(sat__1__2 A) ∈ coverstates"
 step1 : "⟦∃ A. (sat_eu__2__3 (Suc A)) ∈ coverstates⟧ ⟹
                                          (sat_eu__2__4 A) ∈ coverstates"
 step2 : "⟦∃ A. (sat_eu__2__3 (Suc A)) ∈ coverstates⟧ ⟹
                                          (sat_eu__2__5 A) ∈ coverstates"
 step3 : "⟦∃ A. (sat_eu__2__4 (Suc A)) ∈ coverstates⟧ ⟹
                                          (sat_eu__2__3 A) ∈ coverstates"
 step4 : "⟦∃ A. (sat_eu__2__5 (Suc A)) ∈ coverstates⟧ ⟹
                                          (sat_eu__2__3 A) ∈ coverstates"
```

Similarly, instead of using the concept of paths, we may directly specify the set of reachable states inductively in Isabelle/Isar. For our example the following specification would fit the purpose:

```
consts
 reachstates:: "state set"
inductive reachstates
intros
 zero : "(start B) ∈ reachstates"
 step1 : "⟦∃ A B C D E. ((Suc A),(Suc B),(Suc C),D,E) ∈ reachstates⟧ ⟹
                                    (A,(Suc B),C,(Suc D),E) ∈ reachstates"
 step2 : "⟦∃ A B C D E. ((Suc A),(Suc B),(Suc C),D,E) ∈ reachstates⟧ ⟹
                                    (A,B,(Suc C),D,(Suc E)) ∈ reachstates"
 step3 : "⟦∃ A B C D E. (A,B,C,(Suc D),E) ∈ reachstates⟧ ⟹
                                    ((Suc A),B,(Suc C),D,E) ∈ reachstates"
 step4 : "⟦∃ A B C D E. (A,B,C,D,(Suc E)) ∈ reachstates⟧ ⟹
                                    ((Suc A),(Suc B),C,D,E) ∈ reachstates"
```

Then, the lemma to be verified to show the soundness of the coverability relation is

```
lemma "x ∈ reachstates ⟹ x ∈ coverstates"
```

However, lets assume that the specification of `coverstates` is unknown and has to be generated by Isabelle. To this end we may attempt to prove the following lemma:

```
lemma "∃ coverstates. x ∈ reachstates ⟹ x ∈ coverstates"
```

Thereby it is not important to find a proof, since there are many sets which fulfill this criterion (e.g. the (minimal) set `reachstates` and the (maximal) set of all states). Instead it is important to find a proof, which generates the induction steps of the above specification of `coverstates` as (or as parts of) subgoals. In other words, the question is whether Isabelle's proof methods can imitate the behaviour of Ecce (or other model checkers for Petri nets).

The most important elements of Ecce's partial deduction method to generate the coverability graph are: *coverability test*, *unfolding*, *whistling*, *abstraction*. The coverability test can easily be defined in Isabelle/Isar, e.g.:

```
⟦ x∈ state; y∈ state; x≤y⟧ ⟹ covers(y,x)
```

where $\leq$ is defined as an order on the set of states. We may also check whether a set of states is covered by another set of states, e.g.:

```
∀ B. ∃ A. covers((0,0,0,A,0),(0,0,0,(Suc B),0))
```

Similarly, we may define *whistling* for two states (state sets) or even for the states on a path (a whistle blows if a newly encontered state is (in some sense) bigger than any of its predecessors on the path, thereby it indicates a potentially infinite growth).

The *unfolding* corresponds in Isabelle simply to the rewriting of a subgoal using a definition, in case of Petri nets the definition of the transition function.

The most difficult element to imitate seems to be the *abstraction*. Given a certain subgoal Isabelle's proof method has to replace this subgoal by a more general one. E.g., if unfolding of a transition has led to a subgoal containing the state `(0,0,0,(Suc 0),0)` and the whistle has blown due to a preceding state of the form `(0,0,0,0,0)`, then we have to replace the subgoal by a new one containing a state of the form `(0,0,0,A,0)` (where `A` is all quantified). The only proof rule which is capable of introducing an all quantified variable in Isabelle/Isar is `spec`:

$$\frac{\forall x.P}{P[t/x]}$$

And indeed, by applying `spec` as an introduction rule we may indeed introduce perform a generalisation. For example, assume the following subgoal:

```
1. "(0,0,0,0,0) ∈ coverstates"
```

Executing `apply(rule spec)` and backtracking (using the proof command `back`) generates as the 30th possibility (out of 38):

```
1. ∀ x. (0, 0, 0, x, 0) ∈ coverstates
```

However, we did not succeed yet in implementing a complete proof script using this rule as the search for the appropriate alternative subgoal has to be controlled by the proof script. Within the execution oriented proof style we have focused on Isabelle/Isar does not seem to provide enough control without implementing new proof tactics on Isabelle's ML-implementation level.

7 Conclusion and Further Work

In this work we have shown that verification of Ecce output using the proof system Isabelle can be achieved for small nets. The execution of the proof script of Section 5 on a Pentium II/400 needed about 90s and the underlying PolyML required 80MB of memory. However, as further experiments with a net containing 14 places and 13 transitions reveiled, more specific proof methods have to be employed as the use of the method `blast` required more than the available 200MB of main memory and therefore had to be canceled. One way of tuning the proof process further is by restricting the number of rules potentially applied by `blast`. However, while rules can easily be removed from and added to the list of simplification rules in Isabelle/Isar, a similar simple manipulation of the "`blast` rules" without rewriting underlying Isabelle proof tactics seems not possible. An indirect way of restricting the search space of `blast` could also be to derive the theory `PN` not from `Main` but from (sets of) more basic theories.

A way of improving the readability of the proof script could be to employ the mathematical proof style instead of the execution oriented style. In the mathematical proof style higher-order pattern matching can be used to control the proof. This may also increase the flexibility of

the proof significantly, in particular if the results have to be generalised for other specifications than those of Petri nets.

Finally, for ISABELLE to automatically generate the coverability relation from the specification of the Petri net we believe that it is necessary to implement a new proof rule/proof method at ISABELLE's implementation level which allows to automatically backtrack over potential hypotheses which are more general than the subgoal to be shown. Another option worth exploring might be to attempt to define a proof scheme using the higher-order pattern matching of ISABELLE/ISAR, which performs the abstraction on proof level: E.g., if a state description matches a certain pattern we attempt to prove a lemma concerning a similar pattern where a constant is replaced by some variable.

References

1. P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *11th IEEE Symposium on Logic in Computer Science*, pages 313–321, 1996.
2. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.
3. A. Finkel. The minimal coverability graph for Petri nets. *Lecture Notes in Computer Science*, 674:210–243, 1993.
4. A. Finkel and P. Schnoebelen. Fundamental structures in well-structured infinite transition systems. In *Proceedings of LATIN'98*, LNCS 1380, pages 102–118. Springer-Verlag, 1998.
5. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 1999. To appear.
6. R. Glück and M. Leuschel. Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification. In *Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, LNCS 1755, pages 93–100, Novosibirsk, Russia, 1999. Springer-Verlag.
7. R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.
8. M. Leuschel. Logic program specialisation. In J. Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory*, LNCS 1706, pages 155–188, Copenhagen, Denmark, 1999. Springer-Verlag.
9. M. Leuschel and H. Lehmann. Coverability of Reset Petri Nets and other Well-Structured Transition Systems by Partial Deduction. In J. Lloyd, editor, *Proceedings of the International Conference on Computational Logic (CL'2000)*, LNCS 1861, London, UK, 2000. Springer-Verlag.
10. M. Leuschel and H. Lehmann. Solving Coverability Problems of Petri Nets by Partial Deduction. In Maurizio Gabbrielli and Frank Pfenning, editors, *Proceedings of PPDP'2000*, pages 268–279, Montreal, Canada, 2000. ACM Press.
11. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In Annalisa Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, September 1999.
12. Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
13. Michael Leuschel, Jesper Jørgensen, Wim Vanhoof, and Maurice Bruynooghe. Offline specialisation in prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, page 52pp, 2003. To appear.
14. Michael Leuschel, Bern Martens, and Danny De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.
15. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

Appendix A: Isabelle theory generator for Ecce (code_generator.pro)

```
/* the location of the last written isabelle theory */

:- dynamic last_isa_file/1.
last_isa_file('~/demo/ecce.gtree.thy').

/* the location of the last read file containing transition clauses */

:- dynamic last_trans_file/1.
last_trans_file('~/ecce/ecce_examples/petri-nets/basicME.P').

/* the last clause name representing transitions */

:- dynamic last_trans_pred/1.
last_trans_pred(trans).

/* the last clause name representing the initial state */

:- dynamic last_initial_pred/1.
last_initial_pred(start).

/* the last clause names of the specialised original clauses */

:- dynamic last_cover_pred/1.
last_cover_pred([sat__1,sat_eu__2]).

/* global clauses to represent the size of a state vector and the arguments of
   the clause representing the initial state
*/

:- dynamic last_state_size/1.
:- dynamic last_initial_args/1.

/* some convenient global variables to simplify the printing */

:- dynamic trans_print_started/0.
:- dynamic cover_print_started/0.
:- dynamic trans_print_N/1.

/* this calls all other clauses required for printing an isabelle theory,
   includes:
   1. User dialogs
   2. Declaration of new clause database (transitionsystem)
   3. Load transitions
   4. Write theory
   5. Undeclare database (transitionsystem)
*/
```

```
print_specialised_program_isa :-
        dialog_isa(last_trans_file,'read transitions from file'),
        dialog_isa(last_trans_pred,'predicate describing transitions'),
        dialog_isa(last_initial_pred,'predicate describing initial state'),
        dialog_isa(last_cover_pred,'list of predicates describing coverability
                                        (names in unspecialised program)'),
        dialog_isa(last_isa_file,'write theory to file'),
        last_trans_file(TF),
        Data=transitionsystem,
        declare_database(Data),
        see(TF),read_database(Data),seen,
        last_initial_pred(IP), /* only the last argument is considered to be a state */
        claus(Data,_,P,_),P=..[IP|Args],last_el(Args,Arg),
        assert(last_initial_args(Arg)),
        length_list(Arg,NS),assert(last_state_size(NS)),
        last_isa_file(IF),
        tell(IF),
        print_isa_header,
        print_isa_type_decl,
        print_isa_path_decl(Data),
        print_isa_path_def(Data),
        print_isa_cover_decl,
        print_isa_cover_def,
        print_isa_lemma,
        print_isa_proofscript,
        told,
        clear_database(Data),
        undeclare_database(Data),
        retract(last_initial_args(_)),
        retract(last_state_size(_)).

/* creates generic dialog, prints string S, removes and adds new clause P(T)
   depending on user input T,
   P,S must be instantiated
*/

dialog_isa(P,S):-
        C=..[P,L],C,
        beginner_print('Type a dot (.) and hit return at the end.'),
        beginner_nl,
        print(S),print(' (1 for '),print(L),print(') =>'),
        read(N),
        ((N=1) -> (T = L);  (T = N,retract(C),CN=..[P,T], assert(CN))).

/* prints theory header */

print_isa_header:-
        print('theory PN = Main:'),nl,nl.
```

```
/* prints type declaration of "state" depending on size of state */

print_isa_type_decl:-
        print('types'),nl,
        nl,
        print(' state = "'),
        last_state_size(N),
        print_isa_prod(N,'nat',' \\<times> '),
        print('"'),nl,nl.

/* prints type declarations for predicate "paths", the predicate representing
   transitions, and the predicate representing the initial state(s),
   Data must be instantiated by appropriate clause database
*/

print_isa_path_decl(Data):-
        print('consts'),nl,
        nl,
        print(' paths:: "(state list) set"'),nl,
        nl,
        last_trans_pred(T),
        printtdecl(Data,T),
        nl,
        print(' '),print(T),
        print(':: "(state \\<times> state \\<times> nat) set"'),nl,nl,
        last_initial_pred(I),last_initial_args(IP),
        collect_vars_isa(IP,V),sort(V,V2),length_list(V2,NI),
        print(' '),print(I),print(' :: "'),
        print_isa_prod(NI,'nat',' \\<Rightarrow> '),
        (NI \== 0 -> print(' \\<Rightarrow> '); true),print('state"'),nl,nl.

/* prints definitions for "paths", the predicate representing transitions,
   Data must be instantiated by appropriate clause database
*/

print_isa_path_def(Data):-
        print('defs'),nl,
        nl,
        last_trans_pred(T),
        printtdef(Data,T),
        nl,
        last_initial_pred(I),last_initial_args(IP),
        print(' '),print(I),print('_def [simp]: "'),print(I),
        collect_vars_isa(IP,V),sort(V,V2),length_list(V2,NI),
        numbervars(V2,0,_),printvlist_isa(V2), print(' \\<equiv> '),
        print_state_isa(IP),print('"'),nl,
        nl,
        print(' '),print(T),print('_def: "'), print(T),
```

```
        print(' \\<equiv> {(x,y,n). '),
        print_trans_isa(Data,T),
        print('}"'),nl,
        nl,
        print('inductive paths'),nl,
        print('intros'),nl,nl,
        print(' zero: "['),print(I),(NI \== 0 -> print(' '); true),
        printvlist_isa(V2),print(')] \\<in> paths"'),nl,
        print(' step: "\\<lbrakk> (y,z,n) \\<in> '),print(T),
        print('; y#l \\<in> paths \\<rbrakk> \\<Longrightarrow> z#(y#l)
                                                        \\<in> paths"'),
        nl,nl.

/* prints type declarations for predicate "coverrel" and the predicates used to
   represent the coverability relation
*/

print_isa_cover_decl:-
        print('consts'),nl,
        nl,
        print(' coverrel:: "(state \\<times> state) set"'),nl,
        nl,
        last_cover_pred(List),
        member(L,List),
        cg_filter_goal(_NodeID,[Goal|R],_MSVGoal,FGoal),
        Goal=..[L|_],
        numbervars(FGoal,0,NrOfVars),
        N2 is NrOfVars + 26,
        numbervars([Goal|R],N2,_), /* these are redundant arguments */
        print(' '),
        filter_print_atom_name_isa(FGoal),
        print(' :: "'),
        print_isa_prod(NrOfVars,'nat',' \\<Rightarrow> '),
        print(' \\<Rightarrow> state"'),
        nl,
        fail.
print_isa_cover_decl:- nl.

/* prints definitions for the predicates used to represent the coverability
   relation
*/

print_isa_cover_def:-
        print('defs'),nl,
        nl,
        last_cover_pred(List),
        member(L,List),
        cg_filter_goal(_NodeID,[Goal|R],MsvGoal,FGoal),
        Goal=..[L|_],
```

```
        numbervars(FGoal,0,NrOfVars),
        N2 is NrOfVars + 26,
        numbervars([Goal|R],N2,_), /* these are redundant arguments */
        print(' '),
        filter_print_atom_name_isa(FGoal),
        print('_def: "'),
        filter_print_atom_isa(FGoal),
        MsvGoal=[C|_],
        print(' \\<equiv> '),
        C=..[_|A],
        print_state_isa(A),
        print('"'),
        nl,
        fail.

/* prints definition for predicate "coverrel" to represent the coverability
   relation
*/

print_isa_cover_def:-
        nl,nl,
        print(' coverrel_def: "coverrel \\<equiv>'),
        spec_clause(SpecClauseNr,FGoal,Body),
        SpecClauseNr\==filter_comment,
        last_cover_pred(List),
        member(L,List),
        cg_filter_goal(_NodeID,[Goal|_],_MsvGoal,FGoal),
        Goal=..[L|_],
        (cover_print_started -> print('\\<union> ');
        assert(cover_print_started)),
        print_isa_coverel(FGoal,Body),
        fail.
print_isa_cover_def :-
        cg_filter_goal(_NodeID,_Goal,_MsvGoal,FGoal),
        not(spec_clause(_SpecClauseNr,FGoal,_Body)),
        print_isa_coverel(FGoal,[fail]),
        fail.
print_isa_cover_def:- print('"'),nl,nl,retract(cover_print_started).

/* prints lemma to be verified */

print_isa_lemma :-
        print('lemma "l \\<in> paths \\<Longrightarrow> \\<exists> y.
                                        ((hd l),y) \\<in> coverrel"'),
        nl.

/* prints proofscript */

print_isa_proofscript :-
```

```
        print(' apply(erule paths.induct)'),nl,
        print('   apply(simp only: '),
        last_initial_pred(IP),print(IP),print('_def'),nl,
        print('                    coverrel_def)'),nl,
        print('   apply(simp only:'),print_coverpred,
        print(')'),nl,
        print('   apply(simp)'),nl,
        print('   apply(blast)'),nl,
        print('  apply(simp only: '),
        last_trans_pred(TP),print(TP),print('_def)'),nl,
        print('  apply(clarify)'),nl,
        print('  apply(((erule disjE)?,'),nl,
        print('           simp only: coverrel_def,'),nl,
        print('           simp,'),nl,
        print('           ((erule disjE)?,'),nl,
        print('              simp only:'),print_coverpred,
        print(','),nl,
        print('              simp|blast)+)+)'),nl.


/* slightly modified atom printing for isabelle,
   note: does not produce correct output for Non-atoms
*/

filter_print_atom_isa(':'(Module,Pred)) :- !,
        filter_print_atom_isa(Module),print(':'),
               /* this is not the proper output for isa */
        filter_print_atom_isa(Pred).
filter_print_atom_isa(X) :- X='$VAR'(_),!,
        print_red(X). /* this is not the proper output for isa */
filter_print_atom_isa(Atom) :-
        Atom =.. [Pred],!,
        print_faithful_functor(Pred).
filter_print_atom_isa(Atom) :-
        Atom =.. [Pred|Args],
        Args = [_|_],!,
        print_faithful_functor(Pred),
        print(' '),
        l_filter_print_isa(Args).
filter_print_atom_isa(Atom) :- print_atom(Atom).

/* slightly modified atom name printing for isabelle */

filter_print_atom_name_isa(Atom) :-
        Atom =.. [Pred],!,
        print_faithful_functor(Pred).
filter_print_atom_name_isa(Atom) :-
        Atom =.. [Pred|Args],
        Args = [_|_],!,
```

```
        print_faithful_functor(Pred).
filter_print_atom_name_isa(Atom) :- print_atom(Atom).

/* slightly modified argument list printing for isabelle */

l_filter_print_isa([H|T]) :-
        filter_print_arg_isa(H),
        l_filter_print1_isa(T).

l_filter_print1_isa([]).
l_filter_print1_isa([H|T]) :-
        print(' '),
        filter_print_arg_isa(H),
        l_filter_print1_isa(T).

/* slightly modified argument printing for isabelle
   note: does not produce correct output for generic variables
*/

filter_print_arg_isa(X) :- var(X),!,
        print_red(X).
filter_print_arg_isa(X) :- X='$VAR'(_),!,
        print_red(X). /* this is not the proper output for isa */
filter_print_arg_isa(X) :-
        print_faithful_isa(X). /* print_green */

/* slightly modified argument printing for isabelle
   note: does not produce correct output for generic variables
*/

print_faithful_isa(X) :- var(X),!,print_red(X).
print_faithful_isa(X) :- X='$VAR'(_),!,
                      /* this is not the proper output for isa */
        print_red(X). /* remove  to faithfully print $VAR variables */
print_faithful_isa(Struct) :-
        Struct =.. [_F],!,
        print_faithful_functor(Struct).
print_faithful_isa([H|T]) :- !,
        print('('),
        print_faithful_isa(H),
        print('#'),
        print_faithful_isa(T),print(')').
print_faithful_isa(Struct) :-
        Struct =.. [s,Arg1], !,
        print('('),
        print('Suc '),
        print_faithful_isa(Arg1),
        print(')').
print_faithful_isa(Struct) :-
```

```
        Struct =.. [F,Arg1|Args],
        print('('),
        print_faithful_functor(F),
        print_faithful_isa(Arg1),
        l_print_faithful_isa(Args,63),
        print(')').

/* slightly modified argument list printing for isabelle
   note: does not produce correct output for more than 62 elements
*/

l_print_faithful_isa([],_).
l_print_faithful_isa([H|T],Nr) :-
        print(' '),
        ((Nr>1, Nr\==[])
        -> (print_faithful_isa(H), Nr1 is Nr - 1,
            l_print_faithful_isa(T,Nr1)
            )
        ;  (print('xtra('), /* this is not the proper output for isa */
            print_faithful_isa(H),
            l_print_faithful_isa(T,63),
            print(')')
           )
        ).


/* prints a single subset of the coverability relation,
   Head, Body must be instantiated
*/

print_isa_coverel(Head,Body) :-
        collect_vars_isa([Head|Body],V),
        sort(V,V2),
        print('{(x,y). \\<exists> '),
        numbervars(clause(Head,Body),0,_),
        printvlist_isa(V2), print('. '),
        print('x='),print('('),
        filter_print_atom_isa(Head),print(')'),
        NewBody = Body, /* remove_redundant_calls(Body,NewBody,[]), */
        print_isa_coverel2(NewBody),
        newlinebreak,
        fail.
print_isa_coverel(_Head,_Body).

/* prints a the goal state description of a subset of the coverability
   relation,
   Arg must be instantiated
*/
```

```
print_isa_coverel2([]) :- print('.').
print_isa_coverel2([Call|T]) :-
        print(' \\<and> '),print('y='),print('('),
        print_call_isa(Call),
        print_body1_with_nl_isa(T),print(')'),
        print('}').

/* note that the second clause does not produce isabelle output */

print_body1_with_nl_isa([]).
print_body1_with_nl_isa([Call|T]) :-
        print(', '),nl,
        print('    '),
        print_call(Call),
        print_body1_with_nl_isa(T).

/* prints a predicate or function call, note that negative literals are not
   supported in this version for isabelle
   Call must be instantiated by a call
*/

print_call_isa(Call) :-
        (is_negative_literal(Call,NegatedAtom)
        -> (print_bold('\\+('),print_atom(NegatedAtom),print_bold(')'))
              /* this does not print properly for isa */
        ;  ((nonvar(Call),infix_predicate(Call),Call =.. [Pred,Arg1,Arg2])
            -> (filter_print_atom_isa(Arg1),print(' '),print(Pred),print(' '),
                filter_print_atom_isa(Arg2))
            ; filter_print_atom_isa(Call)
           )
        ).


/* filters the list of Arg1 to obtain only the variables in Arg2
   Arg1 must be instantiated by a list
*/

collect_vars_isa([],[]).
collect_vars_isa([H|T],[H|L]) :- var(H), collect_vars_isa(T,L).
collect_vars_isa([H|T],L) :- nonvar(H), H =.. [_|N],
                                  collect_vars_isa(N,L1),
                                  collect_vars_isa(T,L2),
                                  appendl_isa(L1,L2,L).


/* append two lists
   Arg1, Arg2 must be instantiated by lists
*/
```

```
appendl_isa([],L2,L2).
appendl_isa([H|T],L2,[H|L]) :- appendl_isa(T,L2,L).

/* prints a list as space seperated sequence
   Arg must be a list
*/

printvlist_isa([]).
printvlist_isa([H]) :- print(' '), print(H).
printvlist_isa([H1|[H2|T]]) :- print(' '), print(H1), printvlist_isa([H2|T]).

/* prints sequence (ES)^(N-1)E
   N,E,S must be instantiated, N to a number
*/

print_isa_prod(0,_,_).
print_isa_prod(1,E,_) :- print(E).
print_isa_prod(N,E,S) :- N>1, M is N-1,
                  print(E), print(S),
                  print_isa_prod(M,E,S).

/* calculates length of list if Arg1 is instantiated,
   generates list of length Arg2 if Arg1 is uninstantiated
*/

length_list([],0).
length_list([_|T],N) :- (ground(N)-> (N>0,M is N-1);true),
                        length_list(T,M), N is M+1.

/* picks the last element of a list
   Arg1 must be instantiated by list
*/


last_el([],[]).
last_el([A],A).
last_el([H|[H1|T]],A) :- last_el([H1|T],A).

/* prints a state vector, including variables and conversion of function
   s(X) into (Suc X)
   X must be instantiated to list
*/

print_state_isa(X) :- print('('), print_state_isa1(X), print(')').
print_state_isa1([]).
print_state_isa1([H]) :- print_state_isa2(H).
print_state_isa1([H1|[H2|T]]) :-
        print_state_isa2(H1), print(','),
        print_state_isa1([H2|T]).
```

```
print_state_isa2(X) :- var(X), print(X).
print_state_isa2(X) :- atomic(X), print(X).
print_state_isa2(X) :- X='$VAR'(_),!,print(X).
print_state_isa2(S) :-
        nonvar(S),
        S =.. [s,Arg1], !,
        print('('),
        print('Suc '),
        print_state_isa2(Arg1),
        print(')').

/* print transition relation
   Data must be instantiated to clause database, T to clause name
*/

print_trans_isa(Data,T) :-
        TH=..[T,N,S1,S2],
        claus(Data,_,TH,_),
        (trans_print_started -> print('\\<or> '); assert(trans_print_started)),
        appendl_isa(S1,S2,S),
        collect_vars_isa(S,V),
        sort(V,V2),
        print('(\\<exists> '),
        numbervars(TH,0,_),
        printvlist_isa(V2), print('. '),
        print('(x,y,n)=('),
        print_state_isa(S1),print(','),
        print_state_isa(S2),print(','),
        print(N),
        print('))'),
        nl,
        fail.
print_trans_isa(_,_) :- retract(trans_print_started).

/* print transition number definition
   Data must be instantiated to clause database, T to clause name
*/

printtdef(Data,T) :-
        TH=..[T,N,_,_],
        claus(Data,_,TH,_),
        (trans_print_N(Num) ->
            (retract(trans_print_N(Num)),M is Num+1,assert(trans_print_N(M)));
            M is 0, assert(trans_print_N(M))),
        print(' '),print(N),print('_def [simp]: "'), print(N),
        print(' \\<equiv> '),print(M),print('"'),nl,
        fail.
printtdef(_,_) :- retract(trans_print_N(_)).
```

```
/* print transition name declaration
   Data must be instantiated to clause database, T to clause name
*/

printtdecl(Data,T) :-
        TH=..[T,N,_,_],
        claus(Data,_,TH,_),
        print(' '),print(N),print(' :: "nat"'),nl,
        fail.
printtdecl(_,_).

/* print space seperated list of cover predicate definition names */

print_coverpred:-
        last_cover_pred(List),
        member(L,List),
        cg_filter_goal(_NodeID,[Goal|_],_MsvGoal,FGoal),
        Goal=..[L|_],
        print(' '),filter_print_atom_name_isa(FGoal),print('_def'),
        fail.
print_coverpred.
```

Appendix B: Multiple clause databases for Ecce (bimtools/claus_database.pro)

```
/* ------------------------------------------- */
/* (C) COPYRIGHT MICHAEL LEUSCHEL 1995,1996,1997 */
/* ------------------------------------------- */

/* New version by Helko Lehmann 2002 */
/* extended to allow the use of multiple clause databases
   simultaneously
*/

/* ------------------- */
/* DATABASE OF CLAUSES */
/* ------------------- */
/* file: claus_database.pro */

:- multifile pre_condition/1, post_condition/1, type/2.
:- dynamic pre_condition/1, post_condition/1, type/2.


:- ensure_consulted('bimtools/makeflat.pro').
:- ensure_consulted('bimtools/makeiff.pro').


/* ---------------------------------------- */
/* REPRESENTING THE PROGRAM TO BE SPECIALISED */
/* ---------------------------------------- */

/* new predicate to store database names */

:- dynamic declared_database/1.
declared_database(compat).

:- dynamic claus/4.
:- dynamic next_free_clause_nr/2.

:- dynamic mode_declaration/6.
        /* mode(Nr,p(In,Any,Out),[In],[Any],[Out]). */
:- dynamic next_free_mode_nr/2.

next_free_clause_nr(compat,N) :- next_free_clause_nr(N).
next_free_mode_nr(compat,N) :- next_free_mode_nr(N).

claus(compat,A,B,C) :- claus(A,B,C).

/* declare a new database */
```

```
declare_database(Name) :- Name \== compat,
                          (declared_database(Name)
                          -> true
                          ; (assert(declared_database(Name)),
                             assert(next_free_clause_nr(Name,1)),
                             assert(next_free_mode_nr(Name,1)))).

/* undeclare a database */

undeclare_database(Name) :- Name \== compat,
                            (declared_database(Name)
                            -> (clear_database(Name),
                                retract(declared_database(Name)),
                                retract(next_free_clause_nr(Name,1)),
                                retract(next_free_mode_nr(Name,1)))
                            ; true).

/* new versions extended by database parameter
   fails if database undeclared
*/

print_claus_database_status(Name) :-
        declared_database(Name),
        print('database name: '),print(Name),nl,
        next_free_clause_nr(Name,Nr),
        Total is Nr - 1,
        print('clauses stored in database: '),print(Total),nl,
        next_free_mode_nr(Name,MNr),
        ((MNr > 1)
        -> (MT is MNr - 1,
            print('mode declarations: '),print(MT),nl)
        ; (true)
        ),
        (claus(Name,_,_,[])
        -> true
        ; print('database contains *no* facts')),nl.


/* -------------- */
/* clear_database */
/* -------------- */

/* reset the clause database to empty */

cd(Name) :- clear_database(Name).

/* succeeds even if database undeclared */
```

```
clear_database(Name) :-
        Name \== compat,retract(claus(Name,_Nr,_Head,_Body)),fail.
clear_database(Name) :-
        Name \== compat,retract(next_free_clause_nr(Name,_X)),fail.
clear_database(Name) :-
        Name \== compat,
        retract(mode_declaration(Name,_Nr,_Call,_In,_Any,_Out)),fail.
clear_database(Name) :-
        Name \== compat,retract(next_free_mode_nr(Name,_X)),fail.
clear_database(Name) :-
        Name \== compat,
        assert(next_free_clause_nr(Name,1)),
        assert(next_free_mode_nr(Name,1)).


/* ------------- */
/* read_database */
/* ------------- */

/* reads clauses from standard input and asserts them as claus/4 facts */
/* be sure to issue ?-see('FILE') before using the call and ?-seen after
*/
/* fails if database undeclared */

read_database(Name) :-
        declared_database(Name),
        next_free_clause_nr(Name,Nr),
        read_database(Name,Nr,NewNr),
        (Name = compat
        -> (retract(next_free_clause_nr(Nr)),
           assert(next_free_clause_nr(NewNr)))
        ; (retract(next_free_clause_nr(Name,Nr)),
          assert(next_free_clause_nr(Name,NewNr)))
        ),
        Delta is NewNr - Nr,
        Total is NewNr - 1,
        print('clauses read:   '),print(Delta),nl,
        print('clauses stored: '),print(Total),nl,
        next_free_mode_nr(Name,MNr),
        ((MNr > 1)
        -> (MT is MNr - 1,
            print('mode declarations stored: '),print(MT),nl)
        ; (true)
        ).

read_database(Name,Nr,Res) :-
        ((read(RTerm),not(RTerm = end_of_file),
          transform_dcg_term(RTerm,DTerm),
          transform_clause(DTerm,Term))
```

```
        -> (Term =.. [Functor|Args],
            ((Functor=(':-'))
            -> (/* we have a clause */
                ((Args = [Head,BodyCommaList])
                 -> (comma_to_list(BodyCommaList,BodyList),
                     add_clause(Name,Nr,Head,BodyList),Nrp1 is Nr + 1)
                 ;  (treat_query(Name,Args), Nrp1 = Nr)
                )
               )
            ;  ((Functor=('-->'))
                -> (/* we have a DCG rule */
                        expand_term(Term, ':-'(Head,BodyCommaList)),
                        comma_to_list(BodyCommaList,BodyList),
                        add_clause(Name,Nr,Head,BodyList),Nrp1 is Nr + 1

                   )
                ;  (/* we have a fact */
                    (special_fact(Term) /* e.g. unfold annotations */
                     -> (treat_special_fact(Term), Nrp1 = Nr)
                     ;  (add_clause(Name,Nr,Term,[]),Nrp1 is Nr + 1)
                    )
                   )
               )
            ),
            read_database(Name,Nrp1,Res)
           )
        ;  (Res = Nr)
        ).

/* add a clause to the database */
add_new_clause(Name,Head,Body) :-
        (Name = compat
        -> (retract(next_free_clause_nr(Nr)),
            NewNr is Nr + 1,
            assert(next_free_clause_nr(NewNr)),
            assert(claus(Nr,Head,Body)))
        ;  (retract(next_free_clause_nr(Name,Nr)),
            NewNr is Nr + 1,
            assert(next_free_clause_nr(Name,NewNr)),
            assert(claus(Name,Nr,Head,Body)))
        ).

treat_query(Name,[mode(ModedCall)]) :-
        !,add_new_mode(Name,ModedCall).
treat_query(_Name,[op(Priority,Infix,Op)]) :-
        !,op(Priority,Infix,Op).
treat_query(_Name,Query) :-
        print('### encountered query'),nl,
        print('### '), print(Query),nl.
```

```
        /* add_new_clause(q__u__e__r__y,Query). */

/* add a mode declaration to the database */

add_new_mode(compat,ModedCall) :- !,
        retract(next_free_mode_nr(Nr)),
        NewNr is Nr + 1,
        assert(next_free_mode_nr(NewNr)),
        ModedCall =.. [P|MArgs],
        make_mode_declaration(MArgs,VArgs,InArgs,AnyArgs,OutArgs),!,
        VCall =.. [P|VArgs],
        debug_print(mode(VCall,InArgs,AnyArgs,OutArgs)),debug_nl,
        assert(mode_declaration(Nr,VCall,InArgs,AnyArgs,OutArgs)).

add_new_mode(Name,ModedCall) :-
        retract(next_free_mode_nr(Name,Nr)),
        NewNr is Nr + 1,
        assert(next_free_mode_nr(Name,NewNr)),
        ModedCall =.. [P|MArgs],
        make_mode_declaration(MArgs,VArgs,InArgs,AnyArgs,OutArgs),!,
        VCall =.. [P|VArgs],
        debug_print(mode(VCall,InArgs,AnyArgs,OutArgs)),debug_nl,
        assert(mode_declaration(Name,Nr,VCall,InArgs,AnyArgs,OutArgs)).

add_clause(Name,Nr,Head,BodyList) :-
        Head =.. [Predicate|Args],
        length(Args,Arity),!,
        (dont_assert(Predicate,Arity)
         -> (true)
         ;  (strip_body(BodyList,StrippedBodyList),
             (Name = compat
             -> assert(claus(Nr,Head,StrippedBodyList))
             ; assert(claus(Name,Nr,Head,StrippedBodyList)))
            )
        ).

/* other predicates */

/* transform elements seperated by commas into a real prolog list */
comma_to_list(Com,Lst) :-
        ((nonvar(Com),Com =.. [ ',' , Arg1 , Arg2 ])
         ->     (comma_to_list(Arg1,L1),
                 comma_to_list(Arg2,L2),
                 append(L1,L2,Lst)
                )
         ;      (var(Com) -> (Lst = [call(Com)]) ; (Lst = [Com]))
        ).

make_mode_declaration(X,[],[],[],[]) :- var(X),!,
```

```
        print('### error in mode declaration'),nl.
make_mode_declaration([],[],[],[],[]) :- !.
make_mode_declaration(['i'|T],[V|VT],[V|IT],AT,OT) :- !,
        make_mode_declaration(T,VT,IT,AT,OT).
make_mode_declaration(['-'|T],[V|VT],[V|IT],AT,OT) :- !,
        make_mode_declaration(T,VT,IT,AT,OT).
make_mode_declaration(['o'|T],[V|VT],IT,AT,[V|OT]) :- !,
        make_mode_declaration(T,VT,IT,AT,OT).
make_mode_declaration(['+'|T],[V|VT],IT,AT,[V|OT]) :- !,
        make_mode_declaration(T,VT,IT,AT,OT).
make_mode_declaration(['?'|T],[V|VT],IT,[V|AT],OT) :- !,
        make_mode_declaration(T,VT,IT,AT,OT).
make_mode_declaration([_X|T],[V|VT],IT,[V|AT],OT) :-
        print('### error in mode declaration'),nl,
        make_mode_declaration(T,VT,IT,AT,OT).



/* ================== */
/* transform_clause/2 */
/* ================== */

:- dynamic make_iff_when_reading_clauses/1.
make_iff_when_reading_clauses(off).

transform_clause(Clause,TClause) :-
        make_iff_when_reading_clauses(on),!,
        makeflat(Clause,FClause),
        makeiff(FClause,TClause).
transform_clause(Clause,Clause).

/* ---------------------------------- */
/* set_make_iff_when_reading_clauses/0 */
/* ---------------------------------- */

set_make_iff_when_reading_clauses :-
        print('Transform clauses into prop-iff form when reading in:'),nl,
        print('on: '),nl,
        print('off:'),nl,
        print('Current choice: '),
        make_iff_when_reading_clauses(Cur),
        print(Cur),nl,
        print('choice =>'),
        read(NewValue),
        ((not(NewValue=on),not(NewValue=off))
         -> (print('Illegal value, assuming off'),nl,
             set_make_iff_when_reading_clauses(off))
         ;  (set_make_iff_when_reading_clauses(NewValue))
        ).
```

```
/* ---------------------------------- */
/* set_make_iff_when_reading_clauses/1 */
/* ---------------------------------- */

set_make_iff_when_reading_clauses(_NewVal) :-
        retract(make_iff_when_reading_clauses(_Cur)),
        fail.
set_make_iff_when_reading_clauses(NewVal) :-
        asserta(make_iff_when_reading_clauses(NewVal)).

/* ================= */

/* ============================ */
:- dynamic using_ml_typechecker/1.
        /* Using Michael Leuschel Type Checker */
        /* Should be set to true if the object program uses the type
           checker
        */
        /* developed by Michael Leuschel */

using_ml_typechecker(yes).

dont_assert(pre_condition,1) :- using_ml_typechecker(yes).
dont_assert(post_condition,1) :- using_ml_typechecker(yes).
dont_assert(type,2) :- using_ml_typechecker(yes).

strip_body([],[]).
strip_body([H|T],Res) :-
        strip_literal(H,SH),
        strip_body(T,ST),
        append(SH,ST,Res).

strip_literal(X,[X]) :- var(X),!.
strip_literal('C'(X,Y,Z),['='(X,[Y|Z])]). /* for DCGs */
strip_literal('\+'(X),[not(SX)]) :-
        strip_literal(X,[SX]).
strip_literal(not(X),[not(SX)]) :-
        strip_literal(X,[SX]).
strip_literal(when(_Condition,X),[SX]) :-
        strip_literal(X,[SX]).
strip_literal(prepost_call(X),[X]) :- using_ml_typechecker(yes),!.
strip_literal(prepost_mnf_call(X),[X]) :- using_ml_typechecker(yes),!.
strip_literal(mnf_call(X),[X]) :- using_ml_typechecker(yes),!.
strip_literal(verify_pre(_X),[]) :- using_ml_typechecker(yes),!.
strip_literal(verify_post(_X),[]) :- using_ml_typechecker(yes),!.
strip_literal(X,[X]).

/* ============================ */
```

```
:- dynamic using_special_facts/1.

using_special_facts(yes).

/* special facts: unfold annotations */
special_fact(X) :- var(X),!,fail.

treat_special_fact(Fact) :-
        var(Fact),!,fail.
treat_special_fact(Fact) :-
        assert(Fact).


/* ==================== */

/* for backwards compatibility */

:- dynamic claus/3.
:- dynamic next_free_clause_nr/1.

:-dynamic mode_declaration/5.
        /* mode(Nr,p(In,Any,Out),[In],[Any],[Out]). */
:- dynamic next_free_mode_nr/1.

next_free_clause_nr(1).
next_free_mode_nr(1).

print_claus_database_status :- print_claus_database_status(compat).

cd :- clear_database.

clear_database :-
        retract(claus(_Nr,_Head,_Body)),fail.
clear_database :-
        retract(next_free_clause_nr(_X)),fail.
clear_database :-
        retract(mode_declaration(_Nr,_Call,_In,_Any,_Out)),fail.
clear_database :-
        retract(next_free_mode_nr(_X)),fail.
clear_database :-
        assert(next_free_clause_nr(1)),
        assert(next_free_mode_nr(1)).

rd :- read_database.

read_database :- read_database(compat).

read_database(Nr,Res) :- read_database(compat,Nr,Res).
```

```
add_new_clause(Head,Body) :- add_new_clause(compat,Head,Body).

treat_query(A) :- treat_query(compat,A).

add_new_mode(ModedCall) :- add_new_mode(compat,ModedCall).

add_clause(Nr,Head,BodyList) :- add_clause(compat,Nr,Head,BodyList).
```

Appendix C: Added and changed clauses in Ecce front-end (front_end.pro)

```
action(87) :- /* W for Write to Isabelle file */
        print_specialised_program_isa,
        front_end.

action(104) :- /* h for help */
        print(' ECCE 1.1'),nl,
        print(' The Partial Evaluator based on Characteristic Atoms and
                                                        Global Trees'),
        nl,nl, print(''),
        print_claus_database_status,nl,
        print(' Command Summary:'),nl,
        expert_print('  b: execute Benchmark (from special file)'),expert_nl,
        print('  c: Clear clause database'),nl,
        print('  e: set dEbugging on/off'),nl,
        print('  f: choose File for output'),nl,
        print('  h: Help (also ?)'),nl,
        print('  i: Insert specialised program into clause database'),nl,
        expert_print('  k: execute benchmarK Set(from special file)'),
        expert_nl,
        print('  l: List clause database'),nl,
        print('  o: turn type checking Off'),nl,
        print('  p: Partially evaluate an atom or goal'),nl,
        print('  r: Read clauses into database'),nl,
        print('  s: Set Parameters'),nl,
        print('  v: set user expert leVel on/off'),nl,
        print('  w: Write specialised program to file'),nl,
        print('  W: Write specialised program as Isabelle theory file'),nl,
        print('  x: eXit (also a and q)'),nl,
        print('  ---------------------------------- '),nl,
        expert_print('  a: toggle treatment of open predicates'),expert_nl,
        print('  d: Determinate (post-)unfolding'),nl,
        expert_print('  g: print Global tree'),expert_nl,
        expert_print('  j: generate ic-checking code'),expert_nl,
        print('  m: Msv analysis'),nl,
        print('  n: eNable abstract partial deduction'),nl,
        expert_print('  t: iff clause Transformation'),expert_nl,
        expert_print('  u: manual Unfold'),expert_nl,
        expert_print('  y: redundant argument filtering (RAF) analYsis'),
        expert_nl,
        expert_print('  z: FAR (reversed RAF) analysis'),expert_nl,
        front_end.
```