Inductive Theorem Proving by Program Specialisation: Generating proofs for Isabelle using Ecce

(Invited Talk)

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. In this paper we discuss the similarities between program specialisation and inductive theorem proving, and then show how program specialisation can be used to perform inductive theorem proving. We then study this relationship in more detail for the particular problem of verifying infinite state systems in order to establish a clear link between program specialisation and inductive theorem proving. Indeed, ECCE is a program specialisation system which can be used to automatically generate abstractions for the model checking of 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

Program specialisation aims at improving the overall performance of programs by performing source to source transformations. A common approach within functional and logic programming, known respectively as partial evaluation and partial deduction, is to exploit partial knowledge about the input. It is achieved through a well-automated application of parts of the Burstall-Darlington unfold/fold transformation framework.

The relation between program specialisation and theorem proving has already been raised several times in the literature [28, 10, 29, 27]. In this paper we will examine in closer detail at the relationship between partial deduction and inductive theorem proving.

Partial Deduction The heart of any technique for *partial deduction* is a program analysis phase. Given a program P and an (atomic) goal $\leftarrow A$, one aims to analyse the computation-flow of P for all instances $\leftarrow A\theta$ of $\leftarrow A$. Based on the results of this analysis, new program clauses are synthesised.

In partial deduction, such an analysis is based on the construction of finite and usually incomplete[1], SLD(NF)-trees. More specifically, following the foundations for partial deduction developed in [23] (see also [19] for an up-to-date overview), one constructs

- a finite set of atoms $S = \{A_1, \ldots, A_n\}$, and
- a finite (possibly incomplete) SLD(NF)-tree τ_i for each $(P \cup \{\leftarrow A_i\})$,

[1] As usual in partial deduction, we assume that the notion of an SLD-tree is generalised [23] to allow it to be incomplete: at any point we may decide not to select any atom and terminate a derivation.

such that:

1) the atom A in the initial goal $\leftarrow A$ is an instance of some A_i in S, and
2) for each goal $\leftarrow B_1, \ldots, B_k$ labelling a leaf of some SLD(NF)-tree τ_l, each B_i is an instance of some A_j in S.

The conditions 1) and 2) ensure that *together* the SLD(NF)-trees $\tau_1, \ldots, \tau_n$ form a *complete description* of all possible computations that can occur for all concrete instances $\leftarrow A\theta$ of the goal of interest. At the same time, the point is to propagate the available input data in $\leftarrow A$ as much as possible through these trees, in order to obtain sufficient accuracy. The outcome of the analysis is precisely the set of SLD(NF)-trees $\{\tau_1, \ldots, \tau_n\}$: a complete, and hopefully as precise as possible, description of the computation-flow. Finally, a code generation phase produces a *resultant clause* for each non-failing branch of each tree, which synthesises the computation in that branch. The approach has been generalised to specialising a set of *conjunctions* rather than just atoms in [5]. An overview of control techniques that are used in partial deduction, such as determinacy, homeomorphic embedding, and characteristic trees, can be found in [19].

Let us illustrate conjunctive partial deduction on the following simple program.

```
even(0).
even(s(X)) :- odd(X).                odd(s(X)) :- even(X).
```

Conjunctive partial deduction can specialise this program for the query $\leftarrow even(X) \wedge odd(X)$ by constructing the incomplete SLD-tree for it depicted in Fig. 1. The set S mentioned above would simply be $S = \{even(X) \wedge odd(X)\}$. The specialised program we obtain, is:

```
even_odd(s(X)) :- even_odd(X).
```

It is immediately obvious that *even_odd*(X) will never succeed, and hence that no number is even and odd at the same time. The partial evaluator Ecce [22, 5] will basically produce the same result (slightly more involved as it does not re-order atoms by default) and can also automatically infer the failure of *even_odd*(X) by applying its bottom up more specific program construction phase [24] in the post-processing.

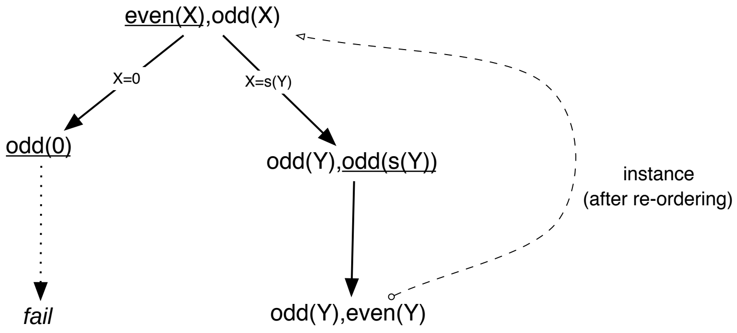

Fig. 1. Specialisation of even-odd

Inductive Theorem Proving Now, the above result corresponds to an inductive proof showing that no number can be both even and odd. The left branch of Fig. 1 corresponds to examining the base case $X = 0$, while the right branch corresponds to the induction step whereby

$even(s(Y)), odd(s(Y))$ is rewritten into the equivalent $odd(Y), even(Y)$ so that the induction hypothesis can be applied.

In a sense the conjunctive partial deduction has identified a working induction schema and the bottom-up propagation [24] has performed the induction proper. This highlights a similarity between partial deduction and *inductive theorem proving*. Indeed, in the induction step of an inductive proof one tries to transform the induction assumption(s) for $n+1$ using basic inference rules so as to be able to apply the induction hypothese(s) and complete the proof. In partial deduction, one tries to transform the atoms in A (or conjunctions for conjunctive partial deduction) by unfolding so as to be able to "fold" back all leaves. The set of atoms A thus plays the role of the induction hypotheses and resolution the role of classical theorem proving. In summary,

– there is a striking similarity between the control problems of partial deduction and inductive theorem proving. The problem of ensuring A-closedness is basically the same as finding induction hypotheses where the induction "goes through'.' Many control techniques have been developed in either camp (e.g., [2] for inductive theorem proving) and cross-fertilisation might be possible.

– if basic resolution steps correspond to logical inference rules one may be able to perform inductive theorem proving directly by partial deduction. For example, Ecce can fully automatically prove associativity of addition [18] (see also[20]).. The only difference is that resolution is not guaranteed to decrease the induction parameter, so this is only guaranteed to work if the predicates to be specialised are inductively defined.

In the next sections we show how Ecce can be used to perform inductive theorem proving as applied to verification tasks and how the induction schemas produced by Ecce can be automatically translated for the proof assistant Isabelle [25].

2 Infinite Model Checking by Program Specialisation

In recent work it has been shown that logic programming based methods in general, and partial deduction in particular, can be applied to model checking [4] of infinite state systems. As this problem can also be tackled by inductive theorem proving [25] we choose this as the basis of a more formal comparison. Indeed, one of the key issues of model checking of infinite systems is *abstraction* [3]. 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. This is related to finding proper induction schemas for inductive theorem proving, which in turn is related to the control problem of partial deduction.

In earlier work we have tried to solve the abstraction problem by applying existing techniques for the *automatic* control of *(logic) program specialisation*, [13] and modelleing the system to be verified as a logic program by means of an interpreter [9, 16]. 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 [22, 17], are applied to logic programs that encode Petri nets [15]. It is even possible to precisely mimic well known Petri net algorithms (by Karp–Miller [11] and by Finkel [6]) when the program specialisation techniques are slightly weakened. The results of [15] 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, 7, 8]. In such cases partial deduction may often be successfully applied as well [14], thereby mimicking well known *backward algorithms* [7].

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 [26] 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 5 we demonstrate how proof scripts can be used in Isabelle for automatic theorem proving. In Section 6 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 7. The last section gives a conclusion and proposes some further work. All relevant source code of the Ecce system can be found in the technical report [12].

3 Specification of Petri nets in Isabelle

The proof assistant Isabelle [25] 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 8).

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*[2].

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 × 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`.

[2] 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).
                    (∃ $list_1$ of variables. (x,y,n)= $transition_1$
                    ∨ (∃ $list_2$ of variables. (x,y,n)= $transition_2$
                    ⋮
                    ∨ (∃ $list_n$ of variables. (x,y,n)= $transition_n$}"
```

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"
```

□

4 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 5.

4.1 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 [12]. 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.

4.2 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 [21] 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).[3] 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).
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.

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

```
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 ≡`

$$\begin{aligned} & \{(x,y).\ \exists\ list_1\ of\ variables.\ x= state_{11} \wedge y= state_{12}\} \\ \cup\ & \{(x,y).\ \exists\ list_2\ of\ variables.\ x= state_{21} \wedge y= state_{22}\} \\ & \vdots \\ \cup\ & \{(x,y).\ \exists\ list_m\ of\ variables.\ x= state_{m1} \wedge y= state_{m2}\}" \end{aligned}$$

The theory generator (cf. [12]) 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 [12] 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))}"
```

□

5 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.

5.1 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.

5.2 Introduction and Elimination

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

5.3 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.

5.4 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 [12] (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)+)+)
```

□

6 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). Full details can be found in the technical report [12]. After this, 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).

7 Automatic Generation of Hypotheses

Instead of defining the coverability as a relation as illustrated in Subsection 4.2 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 ≤ 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.

8 Conclusion and Further Work

We have shown the similarity between controlling partial deduction and inductive theorem proving. We have formally established a relationship between the program specialiser ECCE and the proof system ISABELLE when applied to verifying infinite state Petri nets. 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 6 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. Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: a heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.
3. 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.
4. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
5. Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.
6. A. Finkel. The minimal coverability graph for Petri nets. *Lecture Notes in Computer Science*, 674:210–243, 1993.
7. 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.
8. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 1999. To appear.
9. 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.
10. Robert Glück and Jesper Jørgensen. Generating transformers for deforestation and supercompilation. In Baudouin Le Charlier, editor, *Proceedings of SAS'94*, LNCS 864, pages 432–448, Namur, Belgium, September 1994. Springer-Verlag.
11. R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.
12. Helko Lehmann and Michael Leuschel. Generating inductive verification proofs for Isabelle using the partial evaluator Ecce. Technical Report DSSE-TR-2002-02, Department of Electronics and Computer Science, University of Southampton, UK, September 2002.
13. 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.
14. 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.
15. 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.
16. 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.
17. Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
18. Michael Leuschel. The ECCE partial deduction system. In German Puebla, editor, *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, Universidad Politécnica de Madrid, Tech. Rep. CLIP7/97.1, Port Jefferson, USA, October 1997.
19. Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
20. Michael Leuschel and Danny De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S.D. Swierstra, editors, *Proceedings of PLILP'96*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996. Springer-Verlag.

21. 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*, 2004. To appear.
22. 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.
23. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
24. K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990.
25. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. LNCS 2283. Springer-Verlag, 2002.
26. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
27. Alberto Pettorossi and Maurizio Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *The Journal of Logic Programming*, 41(2&3):197–230, 1999.
28. Valentin F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.
29. Valentin F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 482–509, Schloß Dagstuhl, 1996. Springer-Verlag.