
Article:
Taylor, R., Foster, M. and North, S. (2022) An automated framework for verifying or refuting trace properties of extended finite state machines. International Journal on Software Tools for Technology Transfer, 24 (6). pp. 949-972. ISSN 1433-2779

https://doi.org/10.1007/s10009-022-00666-y

International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-022-00666-y

GENERAL

Regular

An automated framework for verifying or refuting trace properties of extended finite state machines

Ramsay G. Taylor[1] · **Michael Foster**[1] · **Siobhán North**[1]

Accepted: 30 June 2022

Abstract
Model checkers and interactive proof assistants are both used in the assurance of critical systems. Where theorem proving involves the use of axioms and inference rules to mathematically prove defined properties, model checkers can be used to provide concrete counterexamples to refute them. Thus, the two techniques can be thought of as complementary, and it is helpful to use both in tandem to take advantage of their respective strengths. However, this requires us to translate our system model and our desired properties between the two tools which is a time-consuming and error prone process if done manually. The key contribution of this work is a set of automated tools to translate between the Isabelle/HOL proof assistant and the Symbolic Analysis Laboratory (SAL) model checker. We focus on systems specified as extended finite state machines (EFSMs) and on properties specified in linear temporal logic (LTL). We present our representations in the two tools and demonstrate the applicability of our system with respect to an academic example and two realistic case studies. This is a significant contribution to broadening the applicability of these formal approaches, since it allows two powerful verification tools to be easily used in tandem without the risk of human error.

Keywords Software verification · Extended finite state machines · Model checking · Theorem proving

1 Introduction

For many types of critical system it is necessary to provide assurance that the system exhibits certain properties. This can be done by modelling the system in a formal setting (such as a state machine model), expressing the desired properties in a formal way (for example, as linear temporal logic [36] statements), and then providing a formal proof that the model possesses the properties. Model checkers [14] are a standard tool for this.

Many "theorems", in their initially stated form, do not actually hold [2]. This may be because the system violates the desired property, but is often due to a small mistake in the phrasing of the property, a missing assumption, or a flaw in the model. In these cases, either the model or the property must be changed and another verification attempt made. This means that the verification process is iterative. Model checkers are invaluable here because they provide clear information to show how the system violates the specified property in the form of a *counterexample*—a concrete execution of the model which does not satisfy the property. These counterexamples can often be found relatively quickly and with minimal effort on the part of the end user.

However, the automation and speed of model checkers is often based on optimisations. Most popular model checkers only support finite datatypes, and finite subranges of infinite types like the integers. Consequently, while a counterexample is inarguable proof of a violation, the fact that a model checker does not find one is not necessarily proof that the property holds universally. We may, for example, not be examining a large enough subrange.

Michael Foster is funded by the EPSRC CITCoM Project.

✉ Ramsay G. Taylor
r.g.taylor@sheffield.ac.uk

Michael Foster
m.foster@sheffield.ac.uk

Siobhán North
s.north@sheffield.ac.uk

[1] Department of Computer Science, The University of Sheffield, Regent Court, Sheffield S1 4DP, UK

Published online: 29 July 2022

Stronger verification can be provided by interactive theorem provers such as Isabelle/HOL [35] (henceforth referred to simply as Isabelle). The emphasis here is on deductive reasoning through the use of logical inference rules. Proofs judged to be correct by such tools are guaranteed to hold true subject to our trust in the tool's implementation, but the process of constructing such proofs can be very labour intensive and time-consuming. Some automation is provided through tools such as Sledgehammer [6], but the process still requires a great deal of time and skill on the part of the analyst, meaning that theorem provers are less amenable to the iterative nature of the verification process.

In contrast to model checkers, it is often unclear when a given property does not hold. It may be possible to make significant progress into a proof before the invalidity of the property becomes apparent, usually through a contradictory proof state. To minimise wasted proof effort, it is helpful if properties that do not hold (in their stated form) can be identified quickly. This is the motivation behind Isabelle's counterexample generators Nitpick [2] and QuickCheck [9], but the scope and reliability of these generators is limited.

This work aims to combine the best of both worlds, enabling users to leverage the counterexample-finding abilities of a model checker to facilitate iterative model and property development before moving to an automated theorem prover for stronger assurance. We focus specifically on the verification of linear temporal logic (LTL) properties of state transitions systems specified as extended finite state machines (EFSMs). This necessitates a compatible representation of models and properties in both systems, and a translation between the two. This is a time-consuming process if conducted manually and leaves us vulnerable to the introduction of inconsistencies through human error.

We present a framework of EFSM models and LTL properties that we have made available for both the SAL model checker and the Isabelle theorem prover. We also present a set of Java implemented tools to automatically translate between the two systems and also translate into the GraphViz DOT format to allow easy visualisation of the models during model refinement. The specific contributions of this work are:

- **Formal representations of EFSMs in Isabelle** that can be translated to the SAL input language while retaining important semantics.
- **A set of operators compatible with Isabelle's LTL framework** that can be translated to the SAL LTL system in such a way that they retain the semantics of the Isabelle statements.
- A justification as to the **equivalence of our Isabelle and SAL semantics**.

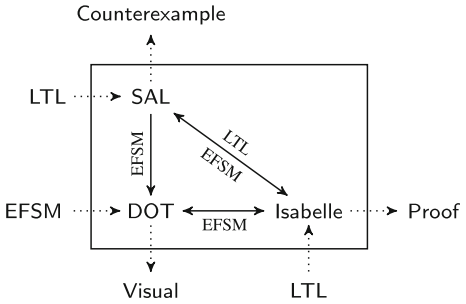


Fig. 1 The integrated analysis environment

- Tool support for automatically converting the **Isabelle representation into the SAL input language**
- Tool support for automatically converting the **Isabelle and SAL representations into the GraphViz DOT display format**
- Tool support for automatically converting **GraphViz DOT diagrams into the Isabelle EFSM formalism.**

Figure 1 shows the workflow that is enabled by the research described here. The solid arrows represent translations that are performed automatically by tools developed in this work. An EFSM model presented in a standardised GraphViz DOT file format can be converted both into a human-readable visual representation and an Isabelle formal representation that preserves the semantics of the system. The required properties of the system can then be added as LTL statements and the Isabelle system provides a semi-automated proof if the properties hold. Both the EFSM and the LTL statements can also be converted automatically to the input format for the SAL model checker, which can be used to automatically generate counterexamples if they exist. Further properties may be added and checked here, and both the model and the properties can be iterated before converting back to Isabelle to attempt a proof.

The remainder of this paper is structured as follows. Section 2 introduces some necessary background in the context of a simple toy example and highlights the limitations of existing approaches. Section 3 describes how we represent EFSM models in both Isabelle and SAL and argues semantic equivalence such that model definitions are consistent between the two systems. Section 4 discusses how we represent LTL properties in both systems. Section 5 gives details of the implementation of our automated translation tool and the system's limitations. Section 6 shows the applicability of our system in the context of two small case studies. Section 7 provides a discussion on the correctness of our translation and of the scalability of our tools. Section 8 discusses related works. Finally, Sect. 9 concludes the paper and discusses possible future research directions.

2 Background

This section gives an overview of the key definitions and technologies relevant to this work in the context of a toy example of a simple vending machine.

2.1 Definitions

Extended Finite State Machines Formal models for verification are often expressed as some form of state machine. Here, systems are represented as a set of states with transitions between them representing the actions the system can perform. Systems which make use of data can be represented by *extended* finite state machines (EFSMs) [11]. While there are numerous definitions in the literature [11,33], our technique uses the definition from [24,30], which is formalised in Isabelle [24,27].

Definition 1 An EFSM is a tuple, (S, s_0, T) where S is a finite non-empty set of states, $s_0 \in S$ is the initial state, and T is the transition matrix $T : (S \times S) \rightarrow \mathscr{P}(L \times \mathbb{N} \times G \times F \times U)$ with rows representing origin states and columns representing destination states. In T, L is a set of transition labels. $\mathbb{N}$ gives the transition *arity* (the number of input parameters), which may be zero. G is a set of Boolean guard functions $G : (I \times R) \rightarrow \mathbb{B}$. F is a set of *output functions* $F : (I \times R) \rightarrow O$. U is a set of *update functions* $U : (I \times R) \rightarrow R$.

In G, F, and U, I is a tuple $[i_1, i_2, \ldots, i_m]$ of values representing the inputs of a transition, which is empty if the arity is zero. Inputs do not persist across states or transitions. R is a mapping from variables $[r_1, r_2, \ldots]$, representing each register of the machine, to their values. Registers are globally accessible and persist throughout the operation of the machine. All registers are initially undefined until explicitly set by an update expression. O is a tuple $[o_1, o_2, \ldots, o_n]$ of values, which may be empty, representing the outputs of a transition.

This differs from the traditional EFSM definition [11] in several ways. In [11], transitions take one literal input and produce one literal output. Our definition assigns each transition an explicit label and allows multiple inputs and outputs (or none at all). Transitions may also produce outputs as a function of input and register values, which allows transition behaviour to be *generalised.*

Definition 1 technically only affords each transition one guard, output, and update, but syntactic sugar allows a transition from state q_m to q_n to take the form

$$q_m \xrightarrow{label:arity[g_1,\ldots,g_g]/f_1,\ldots,f_f[u_1,\ldots,u_u]} q_n$$

in which guards $g_1, \ldots, g_g$ are implicitly conjoined, output functions $f_1, \ldots, f_f$ are evaluated to produce a list of outputs, and update functions $u_1, \ldots, u_u$ are executed simultaneously. We use this notation throughout this work.

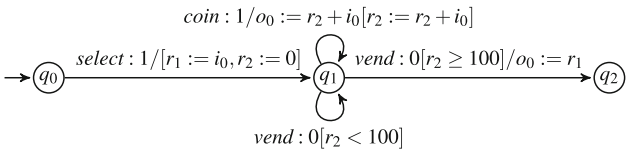


Fig. 2 An EFSM model of the drinks machine

Consider, for example, a simple vending machine which dispenses drinks. Users first *select* a drink. They then insert *coin*s, with the total balance being stored in a *register* r_2 and displayed as output on a small screen. Once sufficient payment has been inserted, the machine *vend*s the selected drink. If the user attempts to dispense their drink before inserting sufficient payment, no output is produced. The EFSM representing this system is shown in Fig. 2.

Executions and Traces EFSMs produce traces in response to executions. An execution is a sequence of (*label*, *inputs*) pairs called *actions* which correspond to method invocations and their arguments. A trace is an ordered sequence of (*label*, *inputs*, *outputs*) triples called *events*, in which *outputs* are the outputs produced by the model in response to invoking method *label* with the given *inputs.* Syntactic sugar allows us to write events as *label*(*inputs*)/*outputs* such that the event $coin(50)/[50]$ represents the *coin* action being called with the input 50 and producing the output 50.

Trace events may also include an extra element which gives information as to the values of the system's internal variables. Such traces are usually called *white box* traces, as they require us to look inside the system. Traces which only contain information visible from outside the system are called *black box* traces.

Trace Properties and Linear Temporal Logic Model checkers allow us to verify systems by specifying properties over their traces expressed in *temporal logics* such as LTL [36]. These logics allow us to express sequential behaviours of our system such as "event a is always followed by event b". In addition to the conventional logic operators of conjunction, disjunction, and negation, LTL provides the following temporal operators.

$G(P)$ for *globally*—the given property, P, is always true
$F(P)$ for *eventually*—the given property, P, becomes true at some point
$X(P)$ for *next*—the given property P is true in the next state
$P\ U\ Q$ for *until*—the given property, P, holds until the release property Q becomes true. The until operator has two flavours, strong and weak, both of which may be expressed in terms of the other. Strong until (sometimes denoted by S) is the default and is only true if the release condition eventually occurs. In contrast, weak until (denoted W) does not require the release

condition, Q, to ever become true if the property P holds globally. Isabelle, SAL, and our toolchain all support both flavours.

These temporal operators only look forward into the future—they are not concerned with past states. Additionally, they only really make sense when applied to infinite traces. While there are adaptations of LTL which work with finite traces [38], most model checkers (including SAL) work with the more conventional operators.

Example 1 Say that we want to verify, for the drinks machine in Fig. 2, that a customer will only receive a drink if they have paid for it. We can phrase this property in LTL as $G(label = vend \wedge X(output = [d]) \implies r_2 \geq 100)$. This states that, if the user performs a *vend* action and the subsequent output is an arbitrary drink d, register r_2 must hold a value greater than or equal to 100 (the price of a drink).

2.2 Tools

Symbolic Analysis Laboratory The Symbolic Analysis Laboratory (SAL) [17] combines tools for abstraction, model checking, theorem proving, and program analysis. Alternative tools such as SPIN [31] and NuSMV [13] provide similar functionality, but our choice of SAL allows us to reuse code from previous work [18]. Additionally, SAL's input language is very conducive to EFSMs. The only real difference is that there is no inherent notion of control flow state, as there is in Definition 1. Instead, this can be modelled by a local variable in the exact same way as the registers. EFSMs are then expressed as a set of "`if` *conditions* `then` *updates*" rules. This is explained in detail in Sect. 3.2.

SAL features three LTL model checkers for different purposes. The *symbolic model checker* uses a binary decision diagram for finite state systems. The finite and infinite *bounded model checkers*, for finite and infinite state systems, respectively, are based on SAT and SMT solving and can perform verification by k-induction.

As in Example 1, LTL properties in SAL are expressed over *models*. When we say "property P holds for model m", what this really means is "property P holds for every feasible trace of model m", but traces are handled under the hood. Counterexamples are then presented as valid traces of the model which violate the specified property. For example, if we were to encode Fig. 2 in SAL, but made our drinks half price such that the transition $q_1 \xrightarrow{vend:0[r_2 \geq 100]/o_0:=r_1} q_2$ instead had the guard $r_2 \geq 50$, then attempting to verify the property from Example 1 leads SAL to produce the counterexample shown in Fig. 3.

```
Counterexample:
=======================
Path
=======================
Step 0:
--- Input Variables (assignments) ---
label = select
i(0) = Str(String__d)
--- System Variables (assignments) ---
ba-pc!1 = 2
cfstate = State__0
r__1 = None
r__2 = None
o(0) = OptionBB
-----------------------
Transition Information:
((label SELECT transition at
  [Context: drinks_machine,
   line(39), column(10)]))
-----------------------
Step 1:
--- Input Variables (assignments) ---
label = coin
i(0) = Num(90)
--- System Variables (assignments) ---
ba-pc!1 = 2
cfstate = State__1
r__1 = Some(Str(String__d))
r__2 = Some(Num(0))
o(0) = OptionBB
-----------------------
Transition Information:
((label COIN transition at
  [Context: drinks_machine,
   line(48), column(10)]))
-----------------------
Step 2:
--- Input Variables (assignments) ---
label = vend
i(0) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 2
cfstate = State__1
r__1 = Some(Str(String__d))
r__2 = Some(Num(90))
o(0) = Some(Num(90))
-----------------------
Transition Information:
((label VEND transition at
  [Context: drinks_machine,
   line(60), column(10)]))
-----------------------
Step 3:
--- Input Variables (assignments) ---
label = coin
i(0) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 1
cfstate = State__2
r__1 = Some(Str(String__d))
r__2 = Some(Num(90))
o(0) = Some(Str(String__d))
```

Fig. 3 A counterexample produced by SAL

Although Fig. 3 is a little verbose, it depicts the trace $\langle select(`d`)\rangle, coin(90)/[90], vend()/[`d`], coin()/[]$.[1] Since we have only inserted one coin with value 90 before receiving our drink, this trace clearly violates the property that r_2 must be greater than or equal to 100 before a drink is dispensed. Armed with this trace, we now know the transitions involved in violating the property, so can inspect these to investigate where the problem is. Since this example is very small, it is easy to locate and fix the problem by changing the guards on the *vend* transition back to 100.

Isabelle/HOL Isabelle/HOL [35] is a proof assistant based on higher-order logic. Its features include complex pattern matching and the ability to define datatypes and functions as in a functional programming language. Proofs over these functions are then expressed as *lemmas* to be proven via the application of a series of inference rules. This process is semi-automated through use of *sledgehammer* [4].

Alternative theorem provers such as Coq [12], Agda [8], and Lean [16] offer similar functionality. We here use Isabelle since for compatibility with previous work [30]. Our work builds on this, and the formalisation has evolved significantly during the work presented here, and in concurrent work [24,25]. Most notably, the ability to define and prove LTL properties over EFSMs is a key development of this work over [30]. Our final definition is openly available in the Archive of Formal Proofs [27].

Proofs of LTL properties generally involve beginning in the initial state and recursively proving that the property holds in each reachable state. To prove that the property in Example 1 holds of the model in Fig. 2, we begin in the initial state q_0, move to q_1 by *select*ing a drink, show that *vend* produces no output unless r_2 holds value 100 or above, at which point we are done.

Graphviz DOT Graphviz [21] is a well established set of open source graph visualisation tools. It allows graphs to be specified in a simple text format in terms of nodes and edges which can then automatically rendered to images like Fig. 2, in which nodes represent states and edges represent transitions. This provides a much more intuitive way to work with EFSMs than the pure text representations of either Isabelle or SAL, especially when done using a WYSIWYG editor. From a practical point of view, it is therefore desirable to be able to translate EFSMs specified using DOT to Isabelle and SAL as well to facilitate more intuitive editing of EFSM models.

[1] Note that this trace is four elements long. The final $coin()/[]$ event allows us to observe the final output [`d`]. Since output is set by transitions as they execute, its value can only be observed in the next state.

2.3 Limitations of current technologies

The goal here is to be able to take advantage of the strong assurance we get from proving a property in Isabelle, but also to use SAL to quickly and easily generate counterexamples to untrue properties. The most notable existing approaches here are Isabelle's built-in counterexample generators, Nitpick [2] and QuickCheck [9]. While these can both be very effective, they are simply not intended to work in what is essentially a model checking context. When run on the property in Example 1, Nitpick reports that "The conjecture either trivially holds for the given scopes or lies outside Nitpick's supported fragment ", and QuickCheck "No type arity value : : full_exhaustive".

Looking in the other direction, towards model checkers, many of these tools are not entirely satisfactory either. While a proof in Isabelle is, subject to our trust in the implementation, irrefutable, we cannot say the same of model checkers. To mitigate the state space explosion problem, model checkers often take certain shortcuts, most notably working with finite subsets of infinite datatypes like integers and strings. This is a problem because properties may appear to hold, even if they do not, if the model is checked using an unsuitable subset of data values. For example, SAL can be made to think that the property in Example 1 holds of the half price drinks machine if we limit integers to be between 0 and 49 since r_2 can never hold a value greater than or equal to 50, meaning that no drink is ever dispensed. Section 3.3 discusses the measures we take to assist the user in determining the values to use, but, ultimately, this is a non-trivial problem which falls outside the intended scope of this work. By contrast, Isabelle is not subject to this limitation as it uses the full infinite datatypes from mathematical theory.

Another technique worth mentioning is SMT solving. By encoding (finite) paths through the model as sets of constraints, solvers such as Z3 [15] can be used to determine whether a trace exists which violates a given property, assuming said property can also be expressed as a set of constraints. This is analogous to the process of symbolic model checking, and SAL does make use of a solver (Yices [19]) under the hood. Unfortunately, the translation of models, properties, and traces to suitable constraints is non-trivial, meaning that counterexamples would be rather unintuitive. Further, without the optimisations performed by SAL such as placing bounds on infinite datatypes, it is also likely to be extremely slow to run, meaning that it is not a viable option.

3 Representing models

This paper aims to support the use of Isabelle to prove LTL properties through the use of a model checker, SAL, to generate counterexamples to untrue properties and facili-

tate subsequent development of the model or property. The key challenge here is formulating a bidirectional translation between the two representations which is semantically consistent. This means that counterexamples produced by SAL must be valid and that SAL must find counterexamples to untrue properties if it is run with a suitable subset of input values (as discussed in Sect. 2).

Our approach works by parsing an input file (either an Isabelle theory or a SAL context) and converting it, including any LTL properties, to the other representation. This section lays out the details of our representation of EFSM models as per Definition 1 in both Isabelle and SAL. The details of how we represent LTL properties are discussed in the following section and a justification for equivalence between the two representations is presented in Sect. 7.

3.1 EFSMs

Being a general purpose prover, Isabelle has no inherent notion of or support for EFSMs out of the box, so we must provide our own formalisation of them by defining datatypes and functions. As described in Definition 1, an EFSM is characterised by three things: a set of states S, an initial state s_0, and a set, T, of transitions between the states in S. Our Isabelle formalisation indexes control flow states using natural numbers, i.e. $S = \mathbb{N}$. We represent transitions between them as a tuple $((s_1, s_2), t)$, where s_1 is the origin state, s_2 is the destination state, and t is a transition as will be discussed in Sect. 3.2. Introducing the convention that the initial state, s_0, is always indexed by zero allows us to represent an EFSM entirely by its set of transitions.

Since the natural numbers is an infinite datatype, this technically violates Definition 1, which states that EFSMs have a *finite* number of states. We could instead define a dedicated finite datatype for each EFSM containing one element for each state, and indeed this is what we do in SAL. However, our Isabelle formalisation forms part of a wider framework [24,25,28] for the inference of EFSMs from their execution traces. This requires the arbitrary addition and subtraction of states from EFSMs, for which an infinite indexing datatype is useful.

In practice, the use of infinite natural numbers to index states is easily resolved in Isabelle by enforcing that any given EFSM only has finitely many transitions. This means that there can only ever be a finite number of states with an incoming or outgoing transition. The rest are "orphaned" and can be ignored such that, rather than having $S = \mathbb{N}$, we have $S = \{s | \exists s'. \exists\, t. ((s, s'), t) \in T \vee ((s', s), t) \in T\}$.

In contrast to Isabelle, SAL is specifically designed to work with EFSM-like models, although the representation is more similar to abstract state machines [7]. These are simply a collection of `if` *condition* `then` *updates* rules, which correspond to EFSM transitions. The main challenge to tackle here is that a *state* in SAL refers to a program state, i.e. a mapping from variables to their values at a particular point in time. There is no inherent notion of a separate control flow state here, but this is easily resolved by representing states as an additional local variable `cfstate` to be checked along with the rest of the transition guards and updated to the new value along with the rest of the updates.

A theme which runs throughout the representation of EFSMs in SAL is a necessity for finite datatypes. SAL cannot check any model which makes use of a datatype that cannot be exhaustively enumerated. Consequently, we cannot use natural numbers to represent states in SAL. Instead, we use the finite enumeration of all states with an incoming or outgoing transition.

Example 2 The simple vending machine in Fig. 2 has three states q_0, q_1, and q_2. In SAL, this becomes the following datatype.

```
STATE : TYPE = {State__0, State__1,
  State__2, NULL_STATE};
```

Here, we also have an additional `NULL_STATE`. This will be explained in Sect. 4.2.1.

type-synonym *guard* = *vname gexp*

type-synonym *output-function* = *vname aexp*

record *transition* =
  Label :: *String.literal*
  Arity :: *nat*
  Guards :: *guard list*
  Outputs :: *output-function list*
  Updates :: *update-function list*

Fig. 4 Isabelle type definitions for transitions

3.2 Transitions

As described in Sect. 2, EFSM transitions have five components: label, arity, guards, outputs, and updates. To implement this in Isabelle, we make use of the built-in `record` type such that each component can be easily accessed by its name. The type definitions for these components are shown in Fig. 4.

Transition labels are strings, and the arities natural numbers. Guards have a defined expression type `gexp` (detailed in Sect. 3.5) and the output and update functions are defined using another datatype `aexp` (detailed in Sect. 3.4). Outputs are simply a list of expressions to be evaluated. Updates are a list of pairs, the first element being the index of the register to be updated, and the second element being an arithmetic expression to be evaluated.

Example 3 The definition below shows how the transition $coin : 1/o_0 := r_2{+}i_1[r_2 := r_2{+}i_0]$ from Fig. 2 is represented in Isabelle.

definition *coin* :: *transition* **where**

```
coin ≡ (|
    Label = STR ''coin'',
    Arity = 1,
    Guards = [],
    Outputs = [Plus (V (R 2)) (V (I 0))],
    Updates = [
            (1, V (R 1)),
            (2, Plus (V (R 2)) (V (I 0)))
           ]
  |)
```

As mentioned above, SAL models are a collection of `if` *condition* `then` *updates* rules, each of which represents a "transition". This maps well to Definition 1, in which transitions also have guards and updates, and most transition fields from Fig. 4 have an obvious mapping.

Example 4 Figure 5 shows how the same *coin* transition as in Example 3 is represented in SAL. A variable followed by a prime indicates its posterior state, and the unprimed version is its anterior value. Note that this transition is tied to a particular control flow state (`cfstate`) where the Isabelle version is not. This is because in SAL we must encode the control flow as a local variable which the transitions update. This is discussed further in Sect. 5.

The `output_sequence ! insert` operation used in the assignment of `o'` is analogous to the `Cons` operation on lists except that, as will be discussed in Sect. 3.3, lists must have a fixed maximum length. In the case of our simple drinks machine, a maximum length of one is sufficient to represent every output produced by the model. The `insert` construction is not as aesthetically pleasing as the standard comma separated lists used by Isabelle, but SAL does not appear to provide a way of defining such custom syntaxes.

To represent transition labels in SAL, we define a special `INPUT` variable. Our Isabelle formalisation represents transition labels as string literals. This is convenient for the wider inference framework [24,28], but falls foul of SAL's requirement for enumerable datatypes. Indeed, SAL has no native support for strings out of the box. To resolve this, we make use of the fact that EFSMs have a finite number of transitions, and thus a finite number of transition labels. This enables us to use a similar trick as for control flow states and simply form an enumeration of the transition labels which occur in the model. For example, the transition labels of our simple drinks machine in Fig. 2 are *select, coin,* and *vend.*

```
  COIN :
    cfstate = State__1 AND
    label = coin AND
    input_sequence!size?(i) = 1 AND
    check_bounds(value_plus(r__2, Some(i(1))))
  -->
    cfstate' = State__1;
    r__1' = r__1;
    r__2' = value_plus(r__2, Some(i(1)));
    o' = output_sequence!
         insert(value_plus(r__2, Some(i(1))),
         output_sequence!empty)
[]
```

Fig. 5 A SAL representation of the *coin* transition

In Isabelle, transitions have an `Arity` field to record the number of inputs the transition expects to receive. This is then checked when the EFSM processes executions. In SAL, we represent this as a guard on each transition with no difference in behaviour between the two representations.

The other slight difference between Isabelle and SAL is that the output `o` behaves just like any another variable. A consequence of this is that the outputs of the current transition are not observable until the next state because they are set along with the register updates. This has no significant effect on the functionality of the models, but it is important to bear in mind when specifying LTL properties, as will be discussed in Sect. 4.

3.3 Input and data values

Definition 1 deliberately does not restrict the types of inputs, outputs, and register values; however, both Isabelle and SAL require a concrete datatype to be specified. For the purposes of this paper, we limit ourselves to integer and string inputs as these are relatively straightforward to work with in both Isabelle and SAL. To tie integers and strings into a single datatype, we define a "sum type" `value`. This tags its members as either a number (`Num`) or a string (`Str`) and is defined as follows.

datatype *value = Num int | Str String.literal*

We use a similar representation in both Isabelle and SAL, although, as will be discussed below, our SAL representation must contain an extra "bottom element".

Since register values are strictly undefined until they are first assigned, the data state is formalised as a function from the register index (a natural number) to a `value option`. The Isabelle `option` type is used to make partial functions total. It takes a type argument and is defined as being either `None` (the bottom element used to represent an undefined value) or `Some x`, where `x` is an element from the specified type, in this case a `value`. Since the number of registers used by any EFSM is known to be finite [24,27], we make use of the theory of finite functions (FinFun [32]) from the HOL library. Here, a FinFun is a function which is constant except for finitely many points. This corresponds to a *map* (or *dictionary*) in conventional programming languages.

In SAL, registers can be represented a similar way. While SAL does not natively support the `option` datatype, it is straightforward to define one. Each register can then be defined as a local variable of type `value option`. Representing inputs in SAL is more challenging due to the fact

that all datatypes need to be enumerable and both integers and strings have an infinite number of inputs. SAL has a native integer type, but models involving this cannot be checked. To tackle this, we define a type `BOUNDED_INT` in SAL which takes two parameters, provided by the user at runtime, which specify the minimum and maximum integer values to consider. This means that when SAL declares a property to hold, we can only be certain that it holds for integers within the specified range.

As mentioned in Sect. 2, this is not just a limitation of SAL, but applies to most established model checkers (e.g. SPIN [31], NuSMV [13]) in one form or another, so is basically unavoidable. The way to mitigate this limitation is to use a suitable subset of integers when checking properties. To assist the user in determining this, our translation tool outputs a comment at the top of the output SAL file informing the user of the largest literal integer that appears in the EFSM. Depending on the structure of the EFSM, this may not be sufficient, especially in situations where the EFSM is entirely symbolic, i.e. where guards and updates are phrased entirely in terms of inputs and registers rather than literal constants. Ultimately, the set of integers needed to verify a given property is highly dependent on both the property and the model. Determining this requires some expertise on the part of the modeller and falls outside the scope of this paper.

While working with a finite subset of integers is very much a necessity, it does introduce a semantic difference from the Isabelle formalisation: that of *arithmetic overflow*. when the value of an expression exceeds the maximum integer, it "loops around" to the beginning of the range. For example, if `BOUNDED_INT` spans the range $-10 \ldots 10$ and we have an expression that should evaluate to 12, it actually evaluates to -9. To stop SAL producing spurious counterexamples that exploit this, we must explicitly guard against it. This is discussed further in Sect. 3.5.

Representing strings is even more challenging since SAL has no native support for them whatsoever. To tackle this, we define strings as an enumerated type, taking the values from the model. We also include a dummy value in our enumeration both to guarantee that our string datatype is never empty, and to mitigate the risk of missing counterexamples due to an insufficiency of data values. As with integers, choosing an appropriate subset of values must ultimately be left up to the user, and the string datatype would ideally be a runtime parameter to SAL, like the integer range, but SAL does not support this. The user is, however, free to modify the enumerated type representing strings as they wish. Because of our efforts to maintain human readability between representations, this is fairly straightforward.

While our Isabelle formalisation defines action arguments as lists of values, this again conflicts with SAL's requirement for finite datatypes. Because lists can be of arbitrary length, there are infinitely many possible lists, so the type is not enumerable. To get around this, we make use of the implementation of finite *sequences* presented in [18]. These are of a fixed size specified in the type declaration and are implemented as finite functions of fixed domain from natural numbers to elements, like an *array*.

To represent inputs as fixed-length SAL sequences, we must apply a lifting such that all inputs are the same length. To do this, a *bottom element* $\perp$ is required for any data type we would like to form a sequence of. This is used to pad out sequences with a length less than the maximum specified in the type declaration, such that the function from indices to elements is total (i.e. every element has a value). The "length" of the sequence is then defined as the minimum index for which the corresponding element is $\perp$. When representing EFSMs, the length of the input sequences used in the traces is set to the maximum arity of any transition in the EFSM. Similarly, the length for the output sequences is the maximum number of outputs produced by any transition in the EFSM.

Example 5 Consider the execution $\langle$f(1, 2), g(1), h(4, 5, 6)$\rangle$. To represent this in SAL, we need to use input sequences with a maximum length of three or more, since action h has three inputs. We then have the execution $\langle f(1, 2, \perp), g(1, \perp, \perp), h(4, 5, 6)\rangle$.

The necessity to represent lists as fixed-length sequences in SAL leads to a slight difference in semantics between the Isabelle and SAL representations. While EFSMs defined in Isabelle can process inputs of arbitrary length, this is not so in SAL since input sequences of length longer than the specified maximum are not members of the datatype. Since the main purpose of using a model checker is to generate *counterexamples*, that is, traces of the model which violate a given property, we do not need to consider actions which take more inputs than the maximum arity of the model since any trace involving such actions is not a valid trace of the model. Invalid traces cannot serve as counterexamples, so the fact that we cannot generate them does not affect our ability to refute untrue properties.

Example 6 Consider again the simple drinks machine EFSM from Fig. 2. Here, the maximum arity of any transition in the model is one. Thus, any trace containing an event which takes more than one input is not a valid trace of the model, even if the action label is *select*, *coin*, or *vend*.

Because of the necessity for a bottom value, our definition of the `value` type in SAL must have three cases where in Isabelle it has only two.

```
B_value : TYPE = DATATYPE
  ValueBB,
  Str(stringOf: STRING),
  Num(intOf: BOUNDED_INT)
END;
```

In `B_value`, the atomic element `ValueBB` represents the bottom element used to pad out sequences. Clearly we do not want this element to occur in the `value` type, but we do want the other two. Consequently, we define the `value` type as `{g : B_value | g /= valueBB}` meaning that `value` is a subtype of `B_value` which does not include `ValueBB`.

3.4 Arithmetic and outputs

Definition 1 allows transitions to perform arithmetic as part of transitions guards, outputs, or updates. While there is no technical restriction on the supported operations, the wider inference context of our Isabelle formalisation [24,25] means that it is necessary to recognise and transform arithmetic expressions. Consequently, we use a *deep embedding* to represent arithmetic expressions. This means that a dedicated expression data type is declared, and the semantics of various cases is defined as a function on top of this. The converse of this, a *shallow embedding*, uses existing syntax to define purely semantic operations which cannot be explicitly checked for. The arithmetic expression (`aexp`) datatype is defined as follows.

datatype *'a aexp* = *L value* | *V 'a* | *Plus 'a aexp 'a aexp* | *Minus 'a aexp 'a aexp* | *Times 'a aexp 'a aexp*

There are five cases here. The two base cases are literal constants (tagged `L`) and variable values (tagged `V`). Here, the variable type is an argument `a'` to `aexp`. This will be explained in Sect. 4. There are then three recursive cases to support the basic numeric operations of addition, subtraction, and multiplication. Division is not supported as this is intuitively a floating point operation, and we only support integer numeric `values`.

Further, the recursive cases `Plus`, `Minus`, and `Times` are only supported over numeric values. Most notably, we do not support the concatenation of strings with the `Plus` operator. This is mostly because the Isabelle formalisation was built to support the inference technique in [24,25], which is numerically focussed, but also because the expression of such an operation in SAL is impossible when representing strings as a finite enumeration.

Because our arithmetic expressions in Isabelle are defined as a separate datatype, we must manually specify how to evaluate them. By default, the expression `Plus (V (R 1)) (V (I 0))` has no inherent meaning. We therefore define an arithmetic evaluation function (`aval`) as follows. This takes an arithmetic expression and a mapping from variables to values and returns a value.

fun *aval* :: *'a aexp* ⇒ *'a datastate* ⇒ *value option* **where**
aval (*L x*) *s* = *Some x* |
aval (*V x*) *s* = *s x* |
aval (*Plus a1 a2*) *s* = *value-plus* (*aval a1 s*)(*aval a2 s*) |
aval (*Minus a1 a2*) *s* = *value-minus* (*aval a1 s*) (*aval a2 s*) |
aval (*Times a1 a2*) *s* = *value-times* (*aval a1 s*) (*aval a2 s*)

In the above definition `aval` actually returns a `value option` rather than just a `value`. This is because Definition 1 is not strongly typed, so we must account for badly typed expressions. Consider, for example, the output expression $r_2 + i_0$ from the *coin* transition in Fig. 2. As discussed above, addition is only supported for numerical values, but there is nothing to stop us from calling the *coin* transition with input "hello", in which case the result of evaluating this expression is undefined. Since Isabelle functions must be total, we define an arithmetic for the `value` datatype in terms of `options` to allow the bottom element, `None`, to represent undefined values. In general, a binary function $f : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ is lifted to the optional arithmetic to become $f' : \mathbb{Z}$ `option` $\to \mathbb{Z}$ `option` $\to \mathbb{Z}$ `option`. The full details of this are described in [24].

We could, in theory, define arithmetic in SAL in exactly the same way as in Isabelle, with a corresponding `aexp` datatype and `aval` function over it. Unfortunately, the `aexp` datatype is defined recursively so, like with the lists used to represent input sequences, SAL cannot handle models specified this way. Instead, we must use a shallow embedding to define arithmetic. What this means is that arithmetic expressions are represented using SAL's existing logical and arithmetic operations rather than with a dedicated datatype and evaluation function.

Because the output expressions of each transition are fixed, we achieve this by essentially unfolding the definition of `aval` during translation instead of during model execution. This means that SAL does not have to deal with the possibility of infinite arithmetic expressions, even though, for obvious reasons, only a finite number could ever actually be used in a real EFSM. To handle arithmetic addition, subtraction, and multiplication in terms of `value options`, we define the auxiliary functions `value_plus`, `value_minus`, and `value_times`, each of which lifts its respective function as above. These are then used in arithmetic expressions in place of their corresponding `aexp` constructors in Isabelle.

3.5 Guards

Transition guards place restrictions on the input and current data state such that the transition can only be taken under certain circumstances. Definition 1 places no restriction on the complexity of these expressions, but the wider inference context of our Isabelle formalisation again requires a deep embedding. Thus, guards are defined as a datatype `gexp` in terms of arithmetic expressions, and a corresponding `gval` function is defined similarly to (and on top of) `aval`.

datatype *'a gexp = Bc bool | Eq 'a aexp 'a aexp | Gt 'a aexp 'a aexp | In 'a value list | Nor 'a gexp 'a gexp*

Here, the only terminal cases are Boolean constants (`Bc`) *true* and *false*. We can compare the results of two arithmetic expressions either for equality (`Eq`), inequality in the form of the "greater than" operator (`Gt`), or set membership (`In`). Finally, we have the logical connective `Nor`, which is equivalent to "not or". This is used instead of the more conventional logical operations (conjunction, disjunction, and negation) because it allows all three to be represented with a single operator. This makes the datatype smaller and thus shrinks proofs over it. For ease of expression, we also define functions to express logical AND, OR, and NOT in terms of NOR, as well as functions for the $<$, $\leq$, $\geq$, and $\neq$ operators.

As with arithmetic expressions, we cannot rely on conventional Boolean logic here, as the result of evaluating a register or arithmetic expression may be undefined. For example, consider the guard $r_1 < i_0$. If r_1 is undefined here, or if either value is non-numeric, the expression cannot evaluate to *true*. It must therefore evaluate to *false*, but this then means that $\neg(r_1 < i_0)$ would evaluate to *true*, which is clearly undesirable. Instead, we use a three-valued Bochvar logic [5], in which *true* and *false* behave like their binary counterparts and we can only take a transition if its guards evaluate to *true*. A third value *invalid* is used to signify that something has gone wrong. Any operation involving *invalid* also evaluates to *invalid* such that we cannot take the transition.

As with arithmetic expressions, it is infeasible to use a deep embedding representation for guards in SAL. Again, we resolve this by essentially expanding the definition of `gval` during translation and representing guards with a shallow embedding using functions like `maybe_nor` in guard expressions instead of datatype constructs.

4 Representing properties

Our work is motivated by the fact that Isabelle's built-in counterexample generators do not work for non-trivial LTL properties of realistic models. Indeed, the only way to discover the falsehood of such a property in Isabelle is to either prove its negation or reach a contradictory proof state, both of which may take a great deal of time and effort. By contrast, model checkers like SAL are designed to find counterexamples quickly and automatically for arbitrarily complex models and properties. Having laid out our respective formalisations of EFSMs in Isabelle and SAL in the previous section we now do the same for LTL properties.

4.1 LTL in SAL

Model checkers are designed to investigate temporal properties of models. It therefore makes sense that SAL has good support for LTL out of the box, supporting all of the temporal operations defined in Sect. 2 as well as the standard logical operators of conjunction, disjunction, negation, and implication. We can define properties over any variable defined in our model, in our case control flow states, registers, inputs, and outputs. SAL also supports global and existential quantification of variables in expressions.

Example 7 Consider again the property $G(label = vend \wedge X(output = [d] \implies r_2 \geq 100))$ from Example 1. This is defined in SAL as follows.

```
output_vend: THEOREM drinks |-
FORALL (d: VALUE):
  G((
      label = vend AND
      X(
        o=output_sequence ! insert(
            Some(d), output_sequence ! empty)
      )
    ) => gval(value_ge(r__2, Some(Num(100)))));
```

Here, we give our property a name, `output_vend`, declare that it concerns the `drinks` machine, and state our property in LTL. SAL does not support free variables, but the arbitrary drink d can be universally quantified without changing the semantics of the formula.

We can then call either the symbolic or bounded model checker to prove or refute our claim. In the above expression, `o=output_sequence ! insert(Some(d), output_sequence ! empty)` corresponds to the output `[d]`. Recall that we must use an optional semantics for evaluating outputs. Thus, if we get a drink `d`, this will be represented as `Some(d)`.

4.2 LTL in Isabelle

Since Isabelle is a general purpose theorem prover, it is not explicitly designed or optimised to work with LTL. Having said that, several formalisations of LTL exist [37,40] within the HOL library and AFP. For this work, we use the `Linear_Temporal_Logic_on_Streams` formalisation [37] as this supports variables of arbitrary datatypes and predicates over them (e.g. $x > 0$) in expressions where [40] only supports Boolean valued variables.

Rather than defining properties directly over models, properties, temporal operators, and logical connectives are all defined as functions which take a stream (i.e. an infinite trace) and return Boolean true or false. This gives Isabelle a much greater expressivity than SAL, most notably the ability to express *hyperproperties* [23], which SAL cannot do. Thus, the properties which are expressible in SAL form a subset of

those which are expressible in Isabelle. It is upon this subset that this work focusses.

In SAL, we express properties in terms of *models.* To verify these properties, SAL explores every possible *trace* of the model in search of one which violates the property. If no such trace is found, the property is reported to hold of the model. When we express an LTL property in Isabelle, it is phrased as "property φ holds on stream π", where π represents a particular trace of the model. This is a problem for us as we would like to express properties over particular models like in SAL.

To achieve this, we must find a way to express our EFSM models as streams. Thinking of each action as a step forward in time, there are five components which characterise a given point in the execution of an EFSM. At each point, the model has a current *control state* and *data state*. Each action has a *label* and possibly some *input* parameters, and its execution may produce some observable *output* and update the data state. It is therefore sufficient to provide a white box trace in the form of a stream of 5-tuples containing these values at each step of the execution.

Simply quantifying a property over all arbitrary traces is likely to lead to a lot of spurious counterexamples as this does not take individual models into account. Instead, we must codify the fact that EFSMs generate traces in response to executions. To do this, we define the coinductive function `watch` which takes an EFSM and an infinite execution (a stream of (label, inputs) pairs) and executes it, starting in the initial state, resulting in a stream of 5-tuples as defined above. Properties over a particular model *m* then take the form φ(`watch m t`), where `t` represents an arbitrary (infinite) execution. Full details of this can be found in [24].

4.2.1 Making EFSMs complete

As discussed in Sect. 2, LTL works in terms of infinite traces. This means that, to define LTL properties over EFSMs, they must be able to process every action from every state, regardless of validity. Models which do this are referred to as *complete,* but this is not a requirement of Definition 1. As with our simple drinks machine in Fig. 2, there may be certain states from which certain actions do not represent valid behaviour so do not have corresponding transitions. To be able to meaningfully express LTL properties over EFSM models, we need a way of making them complete without trivially making all behaviour valid.

To resolve this, we apply the standard procedure of adding an additional "sink state" to models which is entered at the point when the model has no corresponding transition for a given input. This represents an error state from which it is impossible to escape. Once entered, the model will trivially process any action without updating any registers or producing any output. In Isabelle, this is handled implicitly by the `watch` function. In SAL, we make use of the `ELSE` keyword to define a transition which may be taken only when no others can be. This takes the model into the explicit `NULL_STATE` mentioned in Example 2.

4.3 Translating LTL

Since the semantics of LTL are well defined and consistent between Isabelle and SAL, the process of translation is largely a matter of transforming between the two syntaxes. The two main challenges here are the potential for arithmetic overflow, as discussed in the previous section, and fact that Isabelle defines LTL properties over streams (i.e. traces) where SAL defines them over models.

We tackle the problem of arithmetic overflow in the same way as for transition guards and updates: by looking for the arithmetic operations of addition, subtraction, and multiplication and adding overflow checks to the property to account for this. Any arithmetic expression, when translated into SAL, is followed by the same expression surrounded by some boilerplate text to perform the overflow check.

Tackling the problem of Isabelle's representation is more challenging since Isabelle has a much greater expressivity than SAL. Since LTL properties are defined as functions over streams, we can define any such function, and even form anonymous lambda functions within expressions. This means that we cannot translate arbitrary LTL properties from Isabelle to SAL. We therefore restrict ourselves to a particular subset consisting of the temporal operators and logical connectives defined in [37] and the following named predicate functions. Translating this restricted set of predicates between Isabelle and SAL is then a reasonably straightforward mapping between the two syntaxes.

state_eq	takes a natural number representing a control flow state index and returns *true* if this is the control flow state at the head of the stream.
label_eq	takes a string and returns *true* if this is equal to the label at the head of the stream.
input_eq	takes a `value` list and returns *true* if it equals the input at the head of the stream.
output_eq	takes a `value option` list and returns true if this is equal to the output at the head of the stream.
check_exp	takes a guard expression and returns *true* if it holds at the head of the stream.

Of these, it is `check_exp` which is the most interesting. Here, we can supply an arbitrary guard expression as per Sect. 3.5 to be evaluated. Here, though, we may wish to express properties over the *outputs* of the EFSM as well as over its inputs and registers. We can do this easily by defining a new `vname` datatype, `ltl_vname`, as follows.

datatype *ltl-vname* = *Ip nat* | *Op nat* | *Rg nat*

Here, we have inputs represented as `Ip`, outputs as `Op`, and registers as `Rg`. As mentioned in Sect. 3.3, the `gexp` datatype takes a type parameter which is used to represent variables, so we simply use `ltl_vname gexps` in place of the `vname gexps` used for transition guards, and we can then define expressions in terms of inputs, outputs, and registers.

The above functions allow us to define the property from Example 1 as the following.

lemma *LTL-output-vend*:
alw ((((*label-eq* ″*vend*″) *aand* (*nxt* (*output-eq* [*Some d*]))) *impl*
(*check-exp* (*Ge* (*V* (*Rg* 2)) (*L* (*Num* 100))))) (*watch drinks t*)

Here, the logical operations $\wedge$ and $\longrightarrow$ are respectively represented by the `aand` and `impl` operators, which are themselves syntactical constructs that allow us to integrate logical operators into stream expressions. For example, $(p$ `aand` $q)\ s$ is equivalent to $(\lambda s.\ ps \wedge qs)$.

5 Implementation

This section describes the implementation and use of our translation tools, all of which are currently implemented in Java and openly available at [29]. While our implementation takes inspiration from Z2SAL [18], a significant amount of implementation work was still required since Isabelle and Z are very different in their nature, and our toolchain also supports the translation of SAL models and properties back to Isabelle as well as translation both ways between Isabelle and DOT. In total, our implementation required over 6000 lines of new Java code to be written.

5.1 Translating Isabelle to SAL

The translation from Isabelle to SAL is done by first parsing the whole Isabelle EFSM source file and then generating the new SAL translation. It cannot be done line by line on the fly because SAL needs various types to be declared before the model can be defined. In particular, the translator must know all the state and transition labels, the maximum input and output sequence length, and the value of any strings before the model can be written to file. Thus, the whole Isabelle file must be parsed before translation can begin. The internal representation created while parsing the Isabelle to find the strings, states and labels consists of the transitions, a transition matrix and the LTL expressions.

For example, our simple vending machine from Fig. 2 has three transition labels, *select*, *coin*, and *vend*. These are just strings in Isabelle, but become a free type in SAL which must be declared before the EFSM can be defined. We thus end up with the following.

```
LABEL : TYPE = DATATYPE
  coin, init, vend
END;
```

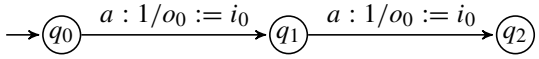
A similar datatype is created for the states of the model. In this case, there are no literal strings, so the `STRING` type contains only the dummy string discussed in Sect. 3.3. The largest literal integer in the model is 100 and occurs in the guards of the two `vend` transitions. A comment is output at the top of the file to indicate this to the user.

The transitions are represented by a generic structure because they occur in both languages. The internal representation starts out as an ordered list of these transitions derived from Isabelle. A list allows us to improve human readability by maintaining the order of transitions across representations as far as possible.

The transition matrix is a transient structure; it is parsed from the Isabelle representation but has no explicit equivalent in SAL. Instead the individual transitions must test the value of the `cfstate` variable as an extra guard condition, and update it to its new value after firing. For example, the *coin* transition from Fig. 2 is given the precondition `cfstate' = State__1` and an extra postcondition `cfstate' = State__1` which leaves `cfstate` unchanged. By contrast, the $q_1 \xrightarrow{vend} q_2$ transition gets the same precondition, but is given the postcondition `cfstate' = State__2` to move the model on to the next state.

Because transitions in SAL explicitly guard and update the control flow state, where the same transition is used more than once in an EFSM between different pairs of states, the transition must be cloned in SAL where, in Isabelle, the named definition of the transition could simply be reused. As a result there is not always a one to one mapping between individual transitions in Isabelle and SAL. Having said that we have gone to some trouble to ensure that the correspondences are as human readable as possible; the original names are retained with just a suffix (__A, __B etc.) to distinguish them, and they appear in the same order as far as possible.

Example 8 Consider the following EFSM, e.



To represent this in Isabelle, we could simply define a transition a and have $e = \{((0, 1), a), ((1, 2), a)\}$. In SAL, however, we would need two separate versions of transition a. The first would be called `a__A` and have a precondition to check that `cfstate = State__0` and a postcondition to update `cfstate' = State__1` The other would be called `a__B` and have a precondition `cfstate = State__1` and postcondition `cfstate' = State__2`.

When translating the LTL properties, we are only interested in translating top level LTL properties of our model.

These can be identified in the Isabelle source because they take the form `LTLproperty (watch EFSMname t)`, as discussed in Sect. 4, where `LTLProperty` is an LTL property, `EFSMname` is an EFSM, and `t` is a free variable. The Isabelle source files may contain other additional lemmas which cannot be meaningfully translated to SAL. Lemmas which are not of the above form are simply ignored. The LTLs we retain are semantically equivalent in both languages, and the internal structure can be exported with either SAL or Isabelle syntax. As with EFSM transitions, the order of LTL properties is maintained across both languages.

Maintaining human readability is useful for sanity checking, but also facilitates hand editing the code generated after SAL has thrown up a useful counterexample. This means that models and properties can be developed using the rapid feedback provided by SAL and can be converted back to Isabelle once there are no more counterexamples. This is explored in Sect. 6.2.

5.2 Translating SAL to Isabelle

This project initially only aimed to translate from Isabelle to SAL for counterexample generation. Having achieved this, the potential advantages of being able to translate back automatically emerged very quickly. This proved to be rather more straightforward than from Isabelle to SAL because of the less restrictive data types used by Isabelle. That is, we are able to work with the full infinite integer and string datatypes rather than just a finite subset of each. The only micro-complication was the generation of a transition matrix from state tests on each transition and reunifying cloned transitions.

One limitation of the Isabelle → SAL → Isabelle round trip translation is that the lemmas that were ignored when translating to SAL are lost forever. The user must manually copy over any non-translatable lemmas between the original Isabelle file and the return translation from SAL. It is therefore advisable to place the lemmas in a separate file which imports either the original Isabelle file defining the EFSM or the return translation from SAL. Unfortunately, there is no way round this as it is impossible to translate arbitrary Isabelle theorems to SAL as Isabelle has much greater expressivity. They could be carried over to SAL as comments, but there seems little value to this.

5.3 Translation to and from DOT

The DOT representation of the model was initially generated from the SAL version after translation as a human readable sanity check, but the human-readable nature of DOT, especially the fact that it can be compiled down to a graphical representation as per Fig. 2, can be helpful if more major structural changes to an EFSM are needed in order to make it satisfy a given property. This is illustrated in Sect. 6.1. Because our translation tool supports the full DOT → Isabelle → SAL loop, if SAL throws up a counterexample, it is possible to edit the EFSM in either the Isabelle, the SAL, or the DOT representation and generate the other two.

Since DOT is a tool for the layout and display of graphical structures like EFSMs, it has no meaningful support for LTL expressions. Thus, these are lost in translation to DOT. This is somewhat of a limitation, but the LTL properties can easily be copied in again from the original source once the return trip has been made from the edited DOT back to either Isabelle or SAL or, better yet, the LTL properties can be written in a separate file which simply imports the EFSM.

5.4 Limitations

It is important to note that the work presented in this paper does not claim to supersede Nitpick or QuickCheck as a general purpose counterexample generator for Isabelle. Instead, we focus specifically on LTL properties defined in terms of streams over EFSM models as per [27]. Further, we here limit ourselves to those properties which are also supported by and expressible in SAL. That is, properties over traces generated by applying the `watch` function as discussed in Sect. 4 over a free execution. Hyperproperties, properties involving multiple models, and arbitrary Isabelle lemmas are not supported by our toolset. This has the knock-on effect that, in translating a theory file from Isabelle to SAL and then back to Isabelle, unsupported lemmas are lost as there is no meaningful way to carry them over to SAL.

Another limitation is quantification. While Isabelle and SAL both support universal and existential quantification of variables arbitrarily within properties, we do not yet support this and leave it to future work. We do, however, support the translation of *free variables* such as in Example 1 from Isabelle to SAL. SAL does not support such variables, but the "forall introduction" rule allows us to use universal quantification as a workaround.

Another limitation is that our translation tools are not integrated into Isabelle like Nitpick and QuickCheck. Instead, users must manually specify those files they wish to translate. This has the advantage that the user can access and modify the translated files such that SAL can be easily used for rapid development, but does mean that we do not have access to Isabelle's parser, so are limited by way of the representations of EFSMs and properties we currently support. We do not, for example, support the translation of EFSMs, transitions, or states represented by the application of functions (e.g. $f(x) = EFSM$). We also cannot translate properties which contain predicates other than those detailed in Sect. 4. This is all left to future work.

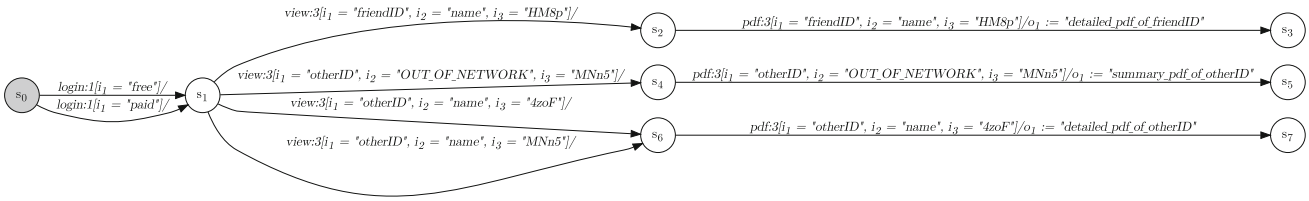


Fig. 6 An abstract EFSM model of the LinkedIn site protocol

6 Worked examples

This section seeks to illustrate the utility of our technique through two realistic examples. The source files for both examples are openly available at [26]. For our technique to be successful, we should be able to translate EFSM models and LTL properties over them between Isabelle and SAL such that SAL can be used to find counterexamples to facilitate rapid development of models and properties and Isabelle can then be used to formally prove that these properties hold when no counterexamples can be found.

6.1 LinkedIn

In 2014, the popular social networking site LinkedIn was shown to have a flaw in its behaviour [20]. When a LinkedIn user u_1 views another user's profile u_2, they are presented with a link to view the information as a PDF file. The amount of information u_1 can see both online and in the PDF varies depending on whether u_1 is a free or paid member and on whether u_2 is in their network (i.e. a "friend"). If u_1 has not paid and u_2 is not in their network, they only get to see summary information rather than the full details.

The attack relies upon the fact that the information which tells the server whether to provide detailed or summary information is contained within the `GET` parameters of the link, so can be edited freely. If u_1 knows or can obtain suitable parameter values, the server can be made to provide the full details of u_2's profile both online and as a PDF. The researchers in [20] discovered that modifying some of the parameter values in the URL caused the server to generate valid links to download detailed PDFs of user profiles which were not in their network even though they had not paid to access this information.

Although [20] does not include an EFSM model of the LinkedIn system, it does publish the sequence of steps necessary to carry out the attack along with the appropriate data values. These are effectively execution traces as discussed in Sect. 2. From these, we used the EFSM inference tool from [25] to produce our model, shown in Fig. 6. The output of this is a DOT file which we converted to Isabelle and then to SAL using our translation tools.

The page requests are represented as events in the traces, with the various URL parameters presented as inputs. There are three principal actions in the system: *login*, *view*, and *pdf*. We abstract the parameter to *login* to represent either a `free` or a `paid` user. The *view* and *pdf* actions both take three parameters. The first of these is the ID of the target profile. To improve readability, these have been replaced by `friendID` (representing the ID of the friend's profile), and `otherID` (representing the ID of the non-friend's profile). The second parameter represents whether the target profile is a friend or not. This appears as `name` if the target profile is a friend and `OUT_OF_NETWORK` if they are not. The third parameter is a pseudorandom authentication token.

In Fig. 6, it is the $s_6 \xrightarrow{pdf} s_7$ transition which represents the attack. The authors of [20] discovered that, when viewing a the profile of a user outside their network (represented by `otherID` in Fig. 6), they could replace the value of the second parameter (`OUT_OF_NETWORK`) with `name`. This caused the server to generate a link to the full detailed PDF file and a valid authentication token (`4zoF`) which could then be manually inserted as the third argument on a second run to view the full profile online.

Only traces involving the `free` user were published in [20], but the written description of the `paid` user functionality was sufficient to infer traces of what would happen if a paid user had attempted the same process. Traces involving the `paid` user were added as a means of inserting a differential between free and paid users; however, due to the vulnerability of the LinkedIn system, these traces did not affect the inferred EFSM beyond an additional *login* transition.

The fact that free users can access detailed information of users outside their network of friends is clearly not what was intended. The required property of the system could be expressed as: "*after a user has logged in as a* `free` *user, they should never be able to get the* `pdf` *action to output the detailed report for users who are not their friend*".

lemma *LTL-neverDetailed*:
(((*label-eq "login" aand input-eq* [*Str "free"*]) *impl*
(*nxt* (*alw* ((*label-eq "pdf" aand*
check-exp (*Eq* (*V* (*Ip* 0)) (*L* (*Str "otherID"*)))) *impl*
(*not* (*nxt* (*output-eq* [*Some* (*Str "detailed-pdf-of-otherID"*)])))))))))
(*watch linkedIn i*)

Fig. 7 An LTL property that defines the expected behaviour

```
Counterexample:
=======================
Path
=======================
Step 0:
--- Input Variables (assignments) ---
label = login
I(1) = STR(String_free)
I(2) = VALUE_BB
I(3) = VALUE_BB
--- System Variables (assignments) ---
ba-pc!1 = 2
cfstate = State_0
O(1) = OPTION_BB
-----------------------
Transition Information:
(module instance at [Context:
  xxxlinkedin_ext, line(186), column(30)]
(label LOGIN
transition at [Context:
  xxxlinkedin_ext, line(75), column(10)]))
-----------------------
Step 1:
--- Input Variables (assignments) ---
label = view
I(1) = STR(String_otherID)
I(2) = STR(String_name)
I(3) = STR(String_MNn5)
--- System Variables (assignments) ---
ba-pc!1 = 1
cfstate = State_1
O(1) = OPTION_BB
-----------------------
Transition Information:
(module instance at [Context:
  xxxlinkedin_ext, line(186), column(30)]
(label VIEW3
transition at [Context:
  xxxlinkedin_ext, line(127), column(10)]))
-----------------------
Step 2:
--- Input Variables (assignments) ---
label = pdf
I(1) = STR(String_otherID)
I(2) = STR(String_name)
I(3) = STR(String_4zoF)
--- System Variables (assignments) ---
ba-pc!1 = 1
cfstate = State_6
O(1) = OPTION_BB
-----------------------
Transition Information:
(module instance at [Context:
  xxxlinkedin_ext, line(186), column(30)]
(label PDF2
transition at [Context:
  xxxlinkedin_ext, line(167), column(10)]))
-----------------------
Step 3:
--- Input Variables (assignments) ---
label = pdf
I(1) = STR(String_otherID)
I(2) = VALUE_BB
I(3) = VALUE_BB
--- System Variables (assignments) ---
ba-pc!1 = 1
cfstate = State_7
O(1) = Some(STR(String_detailed_pdf_of_otherID))
-----------------------
Transition Information:
(module instance at [Context:
  xxxlinkedin_ext, line(186), column(30)]
(label SINK_HOLE
else transition at [Context:
  xxxlinkedin_ext, line(181), column(10)]))
-----------------------
Step 4:
--- Input Variables (assignments) ---
label = pdf
I(1) = NUM(-66)
I(2) = VALUE_BB
I(3) = NUM(64)
--- System Variables (assignments) ---
ba-pc!1 = 0
cfstate = NULL_STATE
O(1) = Some(STR(String_detailed_pdf_of_otherID))
```

Fig. 8 The counterexample generated by SAL

```
lemma LTL-neverDetailed:
  (((label-eq ''login'' aand input-eq [Str ''free'']) impl
   (nxt (alw ((label-eq ''pdf'' aand
   check-exp (Eq (V (Ip 0)) (L (Str ''otherID'')))) impl
   (not (nxt (output-eq [Some (Str ''detailed-pdf-of-otherID'')])))))))
   (watch linkedIn i)
 apply (simp add: ltl-step-alt)
 apply (simp add: implode login-user apply-updates-login login-def
apply-updates-def join-iro-def)
 using after-login[of [] stl i]
 by (simp add: alw-mono)
```

Fig. 9 A proof of the required property for the fixed system

Given the abstraction of the millions of potential user IDs into just `friendID` and `otherID`, this becomes the requirement that after logging in as `free`, the *pdf* action called with the input `otherID` should *not* output `detailed_pdf_of_otherID`. Fig. 7 shows this as an Isabelle lemma using the LTL definitions from Sect. 4.

Attempts can be made to prove the lemma in Fig. 7 but, since the system as it stands does *not* exhibit this property, it will be impossible to prove otherwise. As in Sect. 2, Nitpick and QuickCheck are of no use here. With considerable effort, we can reach a proof state which requires us to prove that the output of transition $s_6 \xrightarrow{pdf} s_7$ in Fig. 6 is not `["detailed_pdf_of_otherID"]`. Unfolding the definition of `pdf2` reveals the contradiction since this is exactly its output. This brings the proof state to `False`, which reveals that the lemma in Fig. 7 is untrue and that there is a flaw in the system, but is too far removed from the original property to provide much insight into how this flaw might either be fixed by the system administrator or exploited by malicious individuals.

Fortunately, our translation tool can convert both the model and the property over to SAL in a few milliseconds. When the Symbolic Model Checker is executed on this property it takes less than a second to respond with the counterexample shown in Fig. 8. Although this is quite verbose, it shows a clear sequence of steps to demonstrate the problem. First, the user logs in as user `free`. Next, they call the *view* action with the parameters `otherID`, `name`, and `MNn5`. This takes the model into state s_6 in Fig. 6, from which the *pdf* action can be called with the same parameters to obtain the detailed PDF of the other user's profile.

Armed with the counterexample in Fig. 8, we now propose an improvement to the system that prevents this flaw being exploited. This is shown in Fig. 10. In this case, the flaw was assuming that session tokens in the URL were trustworthy. A better solution is to include session information in the server itself. To represent this, we added a register (r_1 in Fig. 10) to the model that records whether the user logged in as `paid` or `free`, and checked this in the guard expressions of *view* and *pdf* transitions. The attempted attack

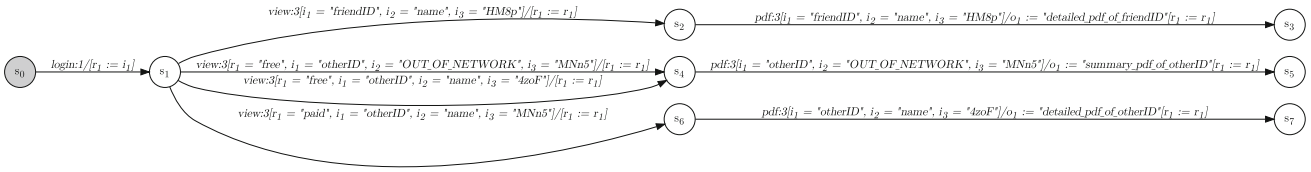


Fig. 10 The EFSM model of the fixed system

now moves the system to the state from which only the `summary_pdf_of_otherID` output can be produced by the *pdf* action. As mentioned in Sect. 5.3, the use of DOT to visualise this change was useful here.

Calling the Symbolic Model Checker on the same property for the modified system results in the output `proved`. This provides some level of assurance that our desired property holds, but is not conclusive proof. As discussed in Sect. 2, we can only use SAL to check a finite subset of transition inputs. For the strongest possible assurance, we use our tool to translate our modified SAL back to Isabelle and prove the property there, as shown in Fig. 9.

6.2 Lift controller

Having demonstrated our techniques on a small security example, we will now apply them to a more complex case study. Work published in [41] details several EFSMs which together model the full functionality of a realistic elevator system with four floors. While the model is perhaps a little simplistic, it demonstrates how EFSMs can be used to model and verify safety critical systems.

In this work, we will examine the "Central Elevator Control" model [41, Figure 3.12], shown in Fig. 11.[2] In the interest of clarity, the *up* and *down* transitions have been abbreviated in the figure and are shown in full in Table 1.

As always, state q_0 is the initial state. The full initialisation of the lift state is quite complex and is abstracted out into a separate EFSM model which sits within state q_0. This model is not shown here but can be found in [41, Figure 3.13]. For this example, only the top-level behaviour illustrated in Fig. 11 needs to be considered. The initialisation sub-model serves to ensure that the doors are closed and the lift is on the first floor. In Fig. 11, the current floor of the lift is held by the register r_4.

State q_1 represents the lift in its "idle" state. It is stationary with its doors closed. It does not contain any passengers and awaits a call to a particular floor. Again, this state contains a sub-model (which can be found in [41, Figure 3.14]) to handle the control logic of summoning the lift. The lift does not store floors to visit in a list, rather, it stores the current direction of travel, the floor it is currently at, and whether it should stop at the next floor it reaches.

States f_1 to f_4 in Fig. 11 represent the lift being in motion. The lift can travel between floors arbitrarily, but cannot ascend above floor 4, nor descend below floor 1. This is because the only incoming transitions to the respective states stop the lift. It is also impossible to select a floor which is not in the range [1..4]. States s_1 to s_4 represent the lift being stationary at a particular floor. Here, the doors may open to allow passengers to enter or alight, after which the lift awaits instruction to visit a particular floor.

A basic safety property of most lifts is that they must be stationary before the doors can be opened. We would like to verify that our lift controller conforms to this. To do this, we must first formalise the intuition in LTL, especially what it means to "stop the motor" and to "open the doors".

In Fig. 11, the motor is stopped by the *motorstop* transitions, and the opening of the door is done by the *opendoor* transitions. The action of successfully opening the doors can be characterised by calling the *opendoor* action and receiving the output $[n, `true']$, where n is the current floor number. We can then phrase the property "we cannot open the door until we have first stopped the lift" in LTL as the following statement, in which n has been left as a free variable to make the property independent of the current floor.

$$(\neg(label = opendoor \wedge X(output = [n, `true']\,)))W\ (label = motorstop) \quad (1)$$

This phrasing of the property omits the outputs of *motorstop* and the inputs to both actions. This is because they do not affect the validity of the property. Calling either action with invalid inputs will cause the model to enter an implicit sink state, from which our property trivially holds since we can never successfully open the doors. Similarly, we do not care about the outputs of *motorstop* either since, if it is called unsuccessfully, the model goes into the sink state, from which we cannot open the doors.

Note also that in Eq. 1, we check the output of *opendoor* in the next state rather than the current state. Recall from Sect. 4 that this is because the output of the current action can only be observed once that action has been completed, i.e. in the

[2] We make a slight deviation here from [41] in that all of our *up* and *down* transitions have the labels *up* and *down*, respectively. This is not the case in [41]. Here, these transitions are uniquely labelled according to their origin and destination floors and whether the lift is to stop at the relevant floor. This deviation has little effect on the model itself but does make Isabelle proofs shorter and simpler since there are fewer unique labels.

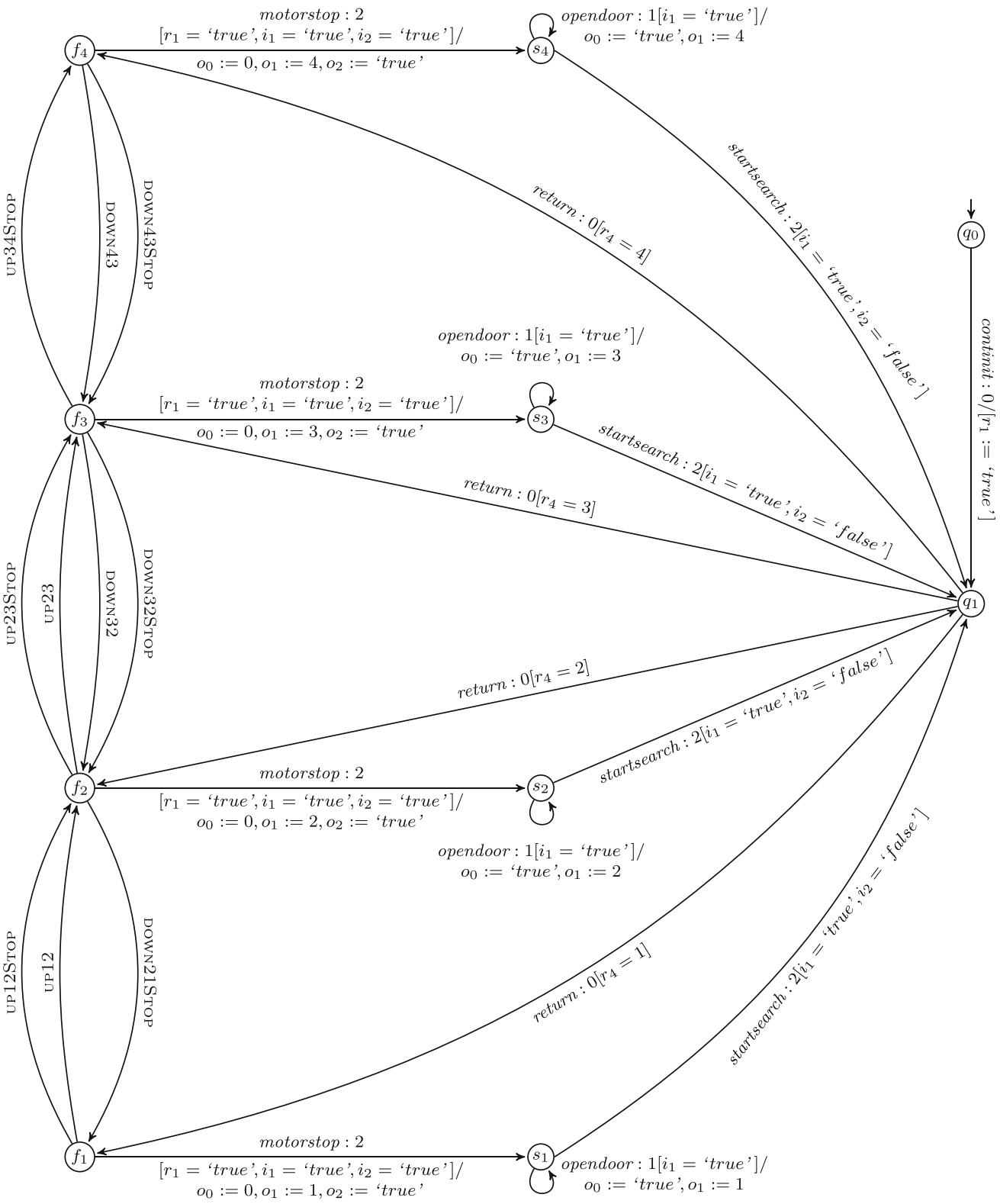


Fig. 11 The EFSM for the lift controller

Table 1 The *up* and *down* transitions from Fig. 11

DOWN43	$down : 3[r_2 = 2, r_1 = \text{`}false\text{'}, i_1 = \text{`}true\text{'}, \ i_2 = \text{`}true\text{'}, i_3 = \text{`}true\text{'}]/o_0 := 2, o_1 := \text{`}true\text{'} \ [r_4 := 3, r_1 := \text{`}true\text{'}]$
DOWN43STOP	$down : 3[r_2 = 2, r_1 = \text{`}false\text{'}, i_1 = \text{`}true\text{'}, i_2 = \text{`}true\text{'}, i_3 = \text{`}false\text{'}]/o_0 := 2, o_1 := \text{`}false\text{'}[r_4 := 3, r_1 := \text{`}false\text{'}]$
UP34STOP	$up : 2[r_2 = 1, r_1 = \text{`}false\text{'}, i_1 = \text{`}true\text{'}, \quad i_2 = \text{`}true\text{'}]/ \quad o_0 := 1, o_1 := \text{`}true\text{'}[r_4 := 4, r_1 := \text{`}true\text{'}]$
DOWN32	$down : 3[r_2 = 2, r_1 = \text{`}false\text{'}, i_1 = \text{`}true\text{'}, \ i_2 = \text{`}true\text{'}, i_3 = \text{`}true\text{'}]/ \ o_0 := 2, o_1 := \text{`}true\text{'}[r_4 := 2, r_1 := \text{`}true\text{'}]$
DOWN32STOP	$down : 3[r_2 = 2, r_1 = \text{`}false\text{'}, i_1 = \text{`}true\text{'}, i_2 = \text{`}true\text{'}, i_3 = \text{`}false\text{'}]/o_0 := 2, o_1 := \text{`}false\text{'}[r_4 := 2, r_1 := \text{`}false\text{'}]$
UP23	$up : 3[r_2 = 1, r_1 = \text{`}false\text{'}, i_1 = \text{`}true\text{'}, \ i_2 = \text{`}true\text{'}, i_3 = \text{`}true\text{'}]/ \ o_0 := 1, o_1 := \text{`}true\text{'}[r_4 := 3, r_1 := \text{`}true\text{'}]$
UP23STOP	$up : 3[r_2 = 1, r_1 = \text{`}false\text{'}, i_1 = \text{`}true\text{'}, i_2 = \text{`}true\text{'}, i_3 = \text{`}false\text{'}]/ \ o_0 := 1, o_1 := \text{`}false\text{'}[r_4 := 3, r_1 := \text{`}false\text{'}]$
DOWN21STOP	$down : 2[r_2 = 2, r_1 = \text{`}false\text{'}, i_1 = \text{`}true\text{'}, \quad i_2 = \text{`}true\text{'}]/o_0 := 2, o_1 := \text{`}true\text{'} \quad [r_3 := 1, r_1 := \text{`}true\text{'}]$
UP12	$up : 3[r_2 = 1, r_1 = \text{`}false\text{'}, i_1 = \text{`}true\text{'}, \quad i_2 = \text{`}true\text{'}, i_3 = \text{`}true\text{'}]/o_0 := 1, o_1 := \text{`}true\text{'} \quad [r_1 := \text{`}true\text{'}, r_4 := 2]$
UP12STOP	$up : 3[r_2 = 1, r_1 = \text{`}false\text{'}, i_1 = \text{`}true\text{'}, \ i_2 = \text{`}true\text{'}, i_3 = \text{`}false\text{'}]/o_0 := 1, o_1 := \text{`}false\text{'} \ [r_4 := 2, r_1 := \text{`}false\text{'}]$

lemma *alw-must-stop-to-open*:
alw ((*ev* (*nxt* ((*label-eq* ''*opendoor*'') *aand*
(*nxt* (*output-eq* [*Some* (*Str* ''*true*''), *Some n*])))))) *impl*
((*not* (*nxt* ((*label-eq* ''*opendoor*'') *aand*
(*nxt* (*output-eq* [*Some* (*Str* ''*true*''), *Some n*])))))) *until*
((((*label-eq* ''*motorstop*'') *or* (*nxt* (*output-eq* [*Some* (*Str* ''*true*''),
Some n])))))) (*watch lift-lit i*)

Fig. 12 The Isabelle representation of the property in Eq. 3

next state. This is common both to the Isabelle framework and to SAL.

Here, we need to use the weak variant of the *until* operator since we do not need to enforce that the motor is eventually stopped: if the lift doors never open, we do not need to ever stop the lift. If we were to use the strong variant of *until*, we would eventually have to stop the lift. This would make the property trivially untrue, since we could simply never call the *motorstop* action. At some point, this would result in us ending up in the sink state from which we can neither stop the motor nor open the doors, but this does not affect the invalidity of the property. We could phrase our property to explicitly exclude the sink state, but this is not as elegant or intuitive as the phrasing in Eq. 1.

Looking at Fig. 11, it seems apparent that the property in Eq. 1 holds. We can only open the doors in states s_1-s_4, which can only be reached by calling the *motorstop* action. The Isabelle proof of this is a relatively straightforward unrolling proof showing that the model enters the sink state if the *opendoor* action is called before *motorstop*.

Unfortunately, Eq. 1 does not accurately capture the intuition of what we want to verify as it does not operate over the entire lifetime of the lift controller. More specifically, once we have stopped the motor for the first time, the *until* operator is released, meaning that anything can happen after this point. It may be the case that, once the motor starts up again, we are then free to open the doors while the lift is in motion. Equation 1 does not prevent this, so we need to phrase our property in such a way that it operates *globally*. An intuitive way to do this is to simply wrap a "globally" operator around Eq. 1, as in Eq. 2.

$$G((\neg(label = opendoor \wedge X(output = [n, \text{`}true\text{'}])))W (label = motorstop)) \tag{2}$$

While Eq. 2 is *intuitive*, it is not actually true. As with the LinkedIn example, Isabelle does not make this immediately clear. While developing this example, we only reached a contradictory proof state after several days of work. However, our tool enables us to easily convert the model and property to SAL and generate a counterexample.

To work with the lift controller in SAL, we first need to account for the fact that we have simplified the initialisation of the system state. Once converted to SAL, we initialise r_4

(the register which holds the current floor) to 1, as is done by the initialisation sub-EFSM in [41].

With this workaround applied, SAL's symbolic model checker can then generate the counterexample shown in Fig. 13. The demonstrated problem is that *opendoors* is a reflexive transition, meaning that we can call it as many times as we like. Since, in s_1, the motor is already stopped, we do not have to explicitly stop it again before opening the doors. Counterexample generation is invaluable for identifying issues with the phrasing of properties (rather than actual faults in the system) as it allows us to quickly iterate our property until it no longer demonstrates spurious errors that are the result of the abstraction choices rather than the real system properties. At each stage, we get a concrete counterexample which allows us to improve the property.

We can see from the model in Fig. 11 that we can only successfully perform an *opendoor* action from a state where the motor is stopped, i.e. s_1 to s_4. We could, therefore, explicitly check this as part of the statement. This property would not be particularly robust, however, since it is not *model independent.* If we renamed the states, the property may no longer hold. It also requires us to have some knowledge of the model such that we know that the motor is stopped in the relevant states. Ideally, we would like to phrase things purely in terms of the observable behaviour of the system, i.e. the inputs and outputs.

When working with this example, it took several iterations before we eventually settled on the property shown in Eq. 3, for which SAL's symbolic model checker produces the output `proved`. The outer *globally* operator states that the property must hold true throughout the entire execution of the model. The inner predicate states that if eventually there is a next state in which we successfully open the doors, this does not occur until either the label is *motorstop* or the output is $[n, `true']$. It is this second disjunct which is the key, since this exploits the fact that, when we open the doors, the outputs are the current floor $n$ and the string $`true'$. We could make this more explicit by writing $label = opendoor \wedge X(output = [n, `true'])$ instead, but this does not affect the validity of the property since *opendoors* is the only transition which could ever produce this output.

$$
\begin{aligned}
&G(\\
&\quad F(X(label = opendoor \wedge X(output = [n, `true'])))) \implies\\
&\quad (\neg(X(label = opendoor \wedge X(output = [n, `true'])))W \qquad (3)\\
&\quad (label = motorstop \vee X(output = [n, `true'])))\\
&)
\end{aligned}
$$

The *next* operations somewhat obfuscate the meaning of Eq. 3. These are necessary to account for the fact that LTL only looks into the future. This property now states that, if we successfully open the doors, the *previous* event was either stopping the motor or opening the doors. We do not have a "previously" operator in LTL though, so we must look one step into the future to effectively treat the current action as the "previous" event.

With SAL unable to find a counterexample for this property, we can now embark on an Isabelle proof. The first step is to translate the SAL property into Isabelle syntax. This can be done automatically using our SAL to Isabelle translation tool, which produced the lemma in Fig. 12.

To prove this lemma, it is helpful to strengthen the property so it applies to all control flow and data states. The reason for this is that the model contains many *cycles*, so we can easily loop back around to states which we have already visited with a different register state. If we do not generalise our property, we end up with potentially infinite proof goals, which we obviously cannot fulfil. To do this, we rephrase the lemma in Fig. 12 to that in Fig. 14.

The precondition in Fig. 14 may seem (and in most cases is) trivially true, so it may be tempting to substitute j for *make-full-observation lift-lit* (*Some s*) *r p t*, in which s, r, p, and t are free variables. This leads to difficulties further on in the proof, however, as we must then prove that any updates made to these variables leave them unchanged, which of course is untrue. We are then left with further subgoals in terms of the updated variable values. What we actually want is to prove the property for an *arbitrary* s, r, p, and t. Where Isabelle's induction package provides the `arbitrary` keyword to achieve this, the coinduction package is still very new and lacks this infrastructure. Phrasing our property as in Fig. 12 is a workaround for this.

Applying coinduction to the lemma in Fig. 14 generates two subgoals. We must show 1. that the property must hold true in the current state and 2. that it holds globally henceforth. Phrasing proof goals as in Fig. 14 makes the second step trivial, as we simply need to prove that we can take a step from any state. Because our EFSM is implicitly complete, we can easily prove this. This means that we only need to prove that the inner predicate holds in every state for every register configuration. Proving this requires that we consider each control flow state in the model as a separate case. This leaves us with ten subgoals: one for each state and an extra one for an arbitrary invalid state. While none of these subgoals are particularly intellectually challenging to prove, the sheer number of them makes the proof fairly long.

Having proved that the property in Eq. 3 holds for the lift controller, we now know that it is impossible to open the doors while the lift is in motion at any point during its operation. Because we made the exact floor a *free variable*, we only need to have proved this once rather than for each floor. Additionally, because we have proven that the property holds for all control flow states and register configurations, we need not concern ourselves with how either of the sub-models modifies the data state. If we were to work these

```
Counterexample:
=======================
Path
=======================
Step 0:
--- Input Variables (assignments) ---
label = continit
i(0) = ValueBB
i(1) = ValueBB
i(2) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 9
cfstate = State__0
r__1 = None
r__2 = None
r__3 = None
r__4 = Some(Num(1))
o(0) = OptionBB
o(1) = OptionBB
o(2) = OptionBB
------------------------
Transition Information:
((label CONTINIT transition at
  [Context: liftcontroller3,
   line(42), column(10)]))
------------------------
Step 1:
--- Input Variables (assignments) ---
label = return
i(0) = ValueBB
i(1) = ValueBB
i(2) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 9
cfstate = State__9
r__1 = Some(Str(String__true))
r__2 = None
r__3 = None
r__4 = Some(Num(1))
o(0) = OptionBB
o(1) = OptionBB
o(2) = OptionBB
------------------------
Transition Information:
((label RETURN1 transition at
  [Context: liftcontroller3,
   line(221), column(10)]))
------------------------
Step 2:
--- Input Variables (assignments) ---
label = motorstop
i(0) = Str(String__true)
i(1) = Str(String__true)
i(2) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 9
cfstate = State__1
r__1 = Some(Str(String__true))
r__2 = None
r__3 = None
r__4 = Some(Num(1))
o(0) = OptionBB
o(1) = OptionBB
o(2) = OptionBB
------------------------
Transition Information:
((label MOTORSTOP1 transition at
  [Context: liftcontroller3,
   line(95), column(10)]))
------------------------
Step 3:
--- Input Variables (assignments) ---
label = opendoor
i(0) = Str(String__true)
i(1) = ValueBB
i(2) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 9
cfstate = State__5
r__1 = Some(Str(String__true))
r__2 = None
r__3 = None
r__4 = Some(Num(1))
o(0) = Some(Num(0))
o(1) = Some(Num(1))
o(2) = Some(Str(String__true))
------------------------
Transition Information:
((label OPENDOOR1 transition at
  [Context: liftcontroller3,
   line(183), column(10)]))
------------------------
Step 4:
--- Input Variables (assignments) ---
label = opendoor
i(0) = Str(String__true)
i(1) = ValueBB
i(2) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 8
cfstate = State__5
r__1 = Some(Str(String__true))
r__2 = None
r__3 = None
r__4 = Some(Num(1))
o(0) = Some(Str(String__true))
o(1) = Some(Num(1))
o(2) = OptionBB
------------------------
Transition Information:
((label OPENDOOR1 transition at
  [Context: liftcontroller3,
   line(183), column(10)]))
------------------------
Step 5:
--- Input Variables (assignments) ---
label = up
i(0) = Num(-1)
i(1) = Str(String__true)
i(2) = Str(String__false)
--- System Variables (assignments) ---
ba-pc!1 = 20
cfstate = State__5
r__1 = Some(Str(String__true))
r__2 = None
r__3 = None
r__4 = Some(Num(1))
o(0) = Some(Str(String__true))
o(1) = Some(Num(1))
o(2) = OptionBB
```

Fig. 13 The SAL counterexample for the property in Eq. 2

```
lemma alw-must-stop-to-open-gen:
  assumes ∃s r p t. j = make-full-observation lift-lit (Some s) r p t
  shows alw ((ev (nxt ((label-eq ''opendoor'') aand
          (nxt (output-eq [Some (Str ''true''), Some n])))))) impl
         ((not (nxt ((label-eq ''opendoor'') aand
          (nxt (output-eq [Some (Str ''true''), Some n])))))) until
        (((label-eq ''motorstop'') or (nxt (output-eq [Some (Str ''true''),
Some n]))))))) j
```

Fig. 14 A generalised version of the property in Fig. 12

models into our EFSM, the property would still hold since neither of them involves opening the doors.

7 Discussion

Having presented the details of our translation framework and demonstrated its utility, this section discusses the correctness of our translations and the scalability of our tools.

7.1 Correctness of translations

Producing a complete and formal *proof* of equivalence of the translations produced would require a deep representation of both Isabelle and SAL's individual representations in a single system. This is well beyond the scope of the work presented here, and it is arguably unnecessary for the intended use of this system.

The correctness of the translations is primarily justified by a *structural equivalence* argument: both Isabelle and SAL have well defined and widely used LTL definitions based on the conventional operators. The conversion of properties between the systems is almost entirely a transliteration — i.e. a simple swapping of the syntax for an operator in one system for the syntax of the same operator in the other system.

The translation of the state and transition system is more subtle. SAL has a built in state and transition semantics, whereas the EFSM semantics in Isabelle was developed and finessed as part of this work. That development included specific design decisions to enable the translation to SAL, most notably the `check_exp` function, and these translatable features are an important contribution of this work.

Further to this, our tool is capable of round trip translation, i.e. Isabelle to SAL and back to Isabelle. This allows us to easily prove the original Isabelle models and properties to be equivalent to their round trip translations. These proofs go through easily in Isabelle by application of the simplifier.

In addition to the structural similarity, there is a question of objective: this work is intended to provide a helpful tool to people attempting to prove properties of EFSMs. Counterexamples produced by SAL can be checked either manually, or in Isabelle's rigorous formal system. Should the counterexample be found to be spurious due to a translation error, it would simply fail to provide helpful information. While this may delay the resolution of system issues, it is the Isabelle-assisted proof that is the guarantee of correctness, so there would be no harm done. Similar can be said for a lack of counterexample due to a mistranslation. If the property did not hold true, no proof to the contrary would be found.

While on the subject of checking counterexamples, it is worth noting Isabelle's built in counterexample generator, `nitpick`, produces counterexamples for conventional Isabelle theorems that are then automatically checked by the proof assistant. While this automatic checking of counter examples is not currently implemented for our translation system, it would be quite possible and is left for future work.

7.2 Scalability

The worked examples in Sect. 6 are insufficient to draw general conclusions about the efficiency, scalability, and usability of our tool. Instead they serve as "proof of concept" and demonstrate our tool's applicability. Notably, Sects. 6.1 and 6.2 demonstrate how the ability to quickly generate counterexamples facilitates the iterative process of model and property development. In both cases, the EFSM and LTL properties were translated between Isabelle and SAL in a few milliseconds. Without this ability, the user must stumble upon a contradictory proof state to discover their property is untrue, which can require considerable time and effort, and is very much reliant on chance.

While the work in this paper provides a fast and automated bridge between Isabelle and SAL, it does not seek to improve either tool directly. Nevertheless, it is worth commenting on their respective scalabilities insofar as Sect. 6 allows. First, the larger the model, the more states must be checked, meaning that verification takes longer. This applies to both Isabelle and SAL, with SAL taking noticeably longer to check models using a larger range of possible integers.

Isabelle does not fall victim to this since it works with the full infinite range of integers, but the construction of Isabelle proofs is labour intensive and time-consuming, with proofs usually growing with the size of the EFSM. This provides motivation for our work, since time spent attempting to construct proofs for untrue properties is time wasted, but is also a limitation to the applicability of Isabelle for the verification of models. As discussed in Sect. 6.2, the phrasing of properties is critical, and the coinduction package is currently nowhere near as developed as the induction package, lacking support for helpful keywords such as `arbitrary`.

Having said that, the coinduction package is still relatively young. Isabelle has seen considerable improvements in proof automation over the years [3,4], so improvement in this area is likely. In the meantime, the implementation of proof tactics such as proving that the property holds in all control flow

states with all register values would help to guide users in the proof process. This is left to future work.

7.3 DOT translations

In addition to translation between Isabelle and SAL, our tool also supports EFSMs represented in DOT. While this does not necessarily have theoretical value, it can add to our tool's usability. While both Isabelle and SAL notations are fairly readable to experienced users, it can be helpful to view EFSMs graphically, for example as in Fig. 11. Numerous WYSIWYG editors exist for DOT, which enables users to edit their models using a familiar GUI rather than in Isabelle or SAL directly. Moreover, several EFSM inference tools [25,43] produce a DOT representation of their inferred model as output, which can then be translated directly to Isabelle and SAL for verification.

8 Related work

There is much work on the analysis of models and many tools available to support this work, with model checkers being the most common means of verifying LTL properties in practice, for example SPIN [31], NuSMV [13] and, of course SAL [17]. A common drawback of these is that models involving infinite datatypes, such as the integers, are not supported. Instead, only a finite subset of values can be checked, which means that the absence of a counterexample does not necessarily guarantee correctness.

One model checker which appears to support some infinite datatypes (reals and integers) is NuXMV [10], although this is not particularly well-established. Another notable model checker is CAVA [22], the implementation of which is fully verified in Isabelle. Further, [39] presents an Isabelle implementation of a conversion of LTL expressions to Büchi automata, with accompanying executable code. Unfortunately, both of these tools only support Boolean-valued variables, making them an unsuitable basis for this work.

Where model checkers are (for the most part) completely automated and excel at generating counterexamples, they do not provide a mathematical certificate of correctness. Automated theorem provers are the converse of this and can prove systems to be correct through the application of sound mathematical rules. Tools such as Isabelle [35], Coq [12], and Agda [8] are all well established, trusted provers which support both infinite datatypes and LTL. Lean [16] is also popular, but is less well established and appears to lack an implementation of LTL.

While tools such as Isabelle's Sledgehammer [4] can provide significant assistance in the construction of proofs, the process is still very much interactive and relies heavily on the user to provide direction. Further, as discussed in Sect. 2, Isabelle's existing counterexample generators, Nitpick [2] and QuickCheck [9], are not applicable to the complex properties covered in this work, meaning that it is impossible to tell whether or not a property is true unless either it or its negation can be proved, or a contradictory proof state is reached.

9 Conclusion

Analysing properties of EFSM models requires an ability to prove properties with certainty and also to disprove them with counterexamples. This work has demonstrated the value of an Isabelle representation for the former, and a SAL model checker for the latter. The ability to move seamlessly and automatically between the two, as well as to work with human-readable Graphviz DOT representations is of great value to a formal analyst.

The consistency and integration with the semantics of EFSM models produced by inference tools such as [25] allows this work to answer the common, mirrored questions of "What can be done with an inferred model?" and "Where will you source your system specification for analysis?". As well as providing proof and model checking support for EFSMs, the formalism supported here is a superset of many other state machine formalisms. Classical FSMs can be modelled with empty guards and data states, allowing these techniques to be applied to the results of various FSM inference techniques [1,42]. Other FSM extensions, such as Mealy machines [34], can have literal input guards and output functions and still be analysed correctly by this infrastructure.

An obvious direction of future work would be to address the limitations set out in Sect. 5.4. Most notably, integrating the tool into the Isabelle framework in a similar manner to QuickCheck [9] would allow us to both expand the range of supported properties and provide a smoother user experience. Further, similar to QuickCheck, it may then be possible for Isabelle users to provide translations for additional datatypes to further leverage the capabilities of model checking for counterexample generation.

References

1. Avellaneda, F., Petrenko, A.: Fsm inference from long traces. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) Formal Methods, pp. 93–109. Springer, Cham (2018)
2. Blanchette, J.C., Nipkow, T.: Nitpick: a counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving, pp. 131–146. Springer, Berlin (2010)
3. Blanchette, J.C., Popescu, A.: Mechanizing the metatheory of sledgehammer. In: Fontaine, P., Ringeissen, C., Schmidt, R.A. (eds.) Frontiers of Combining Systems, pp. 245–260. Springer, Berlin Heidelberg, Berlin, Heidelberg (2013)
4. Blanchette, J.C., Popescu, A., Wand, D., Weidenbach, C.: More spass with isabelle. In: Beringer, L., Felty, A. (eds.) Interactive Theorem Proving, pp. 345–360. Springer, Berlin Heidelberg, Berlin, Heidelberg (2012)
5. Bochvar, D.A.: On a three-valued logical calculus and its application to the analysis of the paradoxes of the classical extended functional calculus. Hist. Philos. Logic **2**(1–2), 87–112 (1981). https://doi.org/10.1080/01445348108837023
6. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) Automated Reasoning Lecture Notes in Computer Science, vol. 6173, pp. 107–121. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-14203-1_9
7. Börger, E., Stärk, R.: Abstract State Machines. Springer, Berlin (2003). https://doi.org/10.1007/978-3-642-18216-7
8. Bove, A., Dybjer, P., Norell, U.: A brief overview of agda—a functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, pp. 73–78. Springer, Berlin (2009)
9. Bulwahn, L.: The new Quickcheck for Isabelle. In: Certified Programs and Proofs, pp. 92–108. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-35308-6_10
10. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Computer Aided Verification, pp. 334–342. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
11. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: 30th ACM/IEEE Design Automation Conference, pp. 86–91. IEEE (1993). https://doi.org/10.1145/157485.164585
12. Chlipala, A.: Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. The MIT Press, Cambridge, MA (2013)
13. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource tool for symbolic model checking. In: Computer Aided Verification, pp. 359–364. Springer, Berlin (2002). https://doi.org/10.1007/3-540-45657-0_29
14. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986). https://doi.org/10.1145/5397.5399
15. de Moura, L., Bjørner, N.: Z3: an efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer, Berlin (2008)
16. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: Automated Deduction—CADE-25, pp. 378–388. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_26
17. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: Sal 2. In: Alur, R., Peled, D.A. (eds.) Computer Aided Verification, pp. 496–500. Springer, Berlin (2004). https://doi.org/10.1007/978-3-540-27813-9_45
18. Derrick, J., North, S., Simons, A.J.H.: Z2sal: a translation-based model checker for z. Formal Aspects Comput. **23**(1), 43–71 (2011). https://doi.org/10.1007/s00165-009-0126-7
19. Dutertre, B.: Yices 2.2. In: Computer Aided Verification, pp. 737–744. Springer, Berlin (2014). https://doi.org/10.1007/978-3-319-08867-9_49
20. Ellis, C.: Bypassing 3rd-degree profiles in LinkedIn by Osanda Malith (2014 (accessed 2019-09-23)). https://www.bugcrowd.com/blog/bypassed-3rd-degree-profiles-linkedin/
21. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz– open source graph drawing tools. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) Graph Drawing, pp. 483–484. Springer, Berlin (2002)
22. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: Computer Aided Verification, pp. 463–478. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-39799-8_31
23. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Computer Aided Verification, pp. 30–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_3
24. Foster, M.: Reverse engineering systems to identify flaws and understand behaviour. Ph.D. thesis, The University of Sheffield (2020)
25. Foster, M., Brucker, A.D., Taylor, R.G., North, S., Derrick, J.: Incorporating data into efsm inference. In: Ölveczky, P.C., Salaün, G. (eds.) Software Engineering and Formal Methods, pp. 257–272. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30446-1_14
26. Foster, M., North, S., Taylor, R.: https://github.com/jmafoster1/efsm-sal
27. Foster, M., Taylor, R., Brucker, A.D., Derrick, J.: A formal model of extended finite state machines. Archive of Formal Proofs (2020 (Accessed 19/09/2020)). http://isa-afp.org/entries/Extended_Finite_State_Machines.html, Formal proof development
28. Foster, M., Taylor, R., Brucker, A.D., Derrick, J.: Inference of extended finite state machines. Archive of Formal Proofs (2020 (Accessed 19/09/2020)). http://isa-afp.org/entries/Extended_Finite_State_Machine_Inference.html, Formal proof development
29. Foster, M., Taylor, R., North, S.: EFSM SAL (2021). https://github.com/jmafoster1/efsm-sal
30. Foster, M., Taylor, R.G., Brucker, A.D., Derrick, J.: Formalising extended finite state machine transition merging. In: International Conference on Formal Engineering Methods, pp. 373–387. Springer (2018)
31. Holzmann, G.: The model checker spin. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
32. Lochbihler, A.: Formalising FinFuns—Generating code for functions as data from Isabelle/HOL. In: Theorem Proving in Higher Order Logics, pp. 310–326. Springer, Berlin (2009)
33. Lorenzoli, D., Mariani, L., Pezzè, M.: Inferring state-based behavior models. In: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis—WODA'06, p. 25. ACM Press, New York (2006). https://doi.org/10.1145/1138912.1138919
34. Mealy, G.H.: A method for synthesizing sequential circuits. The Bell System Technical Journal **34**(5), 1045–1079 (1955). https://doi.org/10.1002/j.1538-7305.1955.tb03788.x

35. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. Lecture Notes in Computer Science, vol. 2283. Springer, Berlin (2002)
36. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), pp. 46–57. IEEE (1977). https://doi.org/10.1109/SFCS.1977.32
37. Popescu, A., Traytel, D.: Linear temporal logic on streams. https://isabelle.in.tum.de/dist/library/HOL/HOL-Library/Linear_Temporal_Logic_on_Streams.html, Formal proof development
38. Roşu, G.: Finite-trace linear temporal logic: coinductive completeness. In: Falcone, Y., Sánchez, C.S. (eds.) Runtime Verification, pp. 333–350. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_21
39. Schimpf, A., Merz, S., Smaus, J.G.: Construction of büchi automata for LTL model checking verified in isabelle/HOL. In: Lecture Notes in Computer Science, pp. 424–439. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-03359-9_29
40. Sickert, S.: Linear temporal logic. Archive of Formal Proofs (2016). https://isa-afp.org/entries/LTL.html, Formal proof development
41. Strobl, F., Wisspeintner, A.: Specifcation of an elevator control system. Tech. rep., TUM (1999 (Accessed 15/05/20)). https://wwwbroy.in.tum.de/publ/papers/elevator.pdf
42. Walkinshaw, N., Bogdanov, K.: Inferring finite-state models with temporal constraints. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. IEEE (2008). https://doi.org/10.1109/ase.2008.35
43. Walkinshaw, N., Taylor, R., Derrick, J.: Inferring extended finite state machine models from software executions. Empir. Softw. Eng. **21**(3), 811–853 (2016). https://doi.org/10.1007/s10664-015-9367-7