Automated Reasoning in the Simulation of Evolvable Systems

Djihed Afifi[1], David E. Rydeheard[1] and Howard Barringer[1]

University of Manchester, Manchester, U.K.
{djihed,david,howard}@cs.man.ac.uk

Abstract

We present a novel application of automated theorem proving for the logical simulation of *evolvable* systems. Modelled using a logical framework, these systems are built hierarchically from components where each component is specified as a first order theory and may have an associated supervisory component. The supervisory component monitors and possibly changes its associated component. The simulation of this framework makes intensive use of automated theory proving – when running a simulation, almost all computational steps are those of a theorem prover. We present this novel combination of a logical setting involving meta-level logics and large sets of formulae for system description, together with theorem proving requirements which involve often slowly changing specifications with the need for rapid assessment of deducibility and consistency. We illustrate how theorem provers are used using an evolvable extension of the blocks world and present a caching structure to reduce simulation times. We then evaluate the suitability of several theorem provers for this application.

1 Introduction

We present an application of automated theorem proving for the simulation of computational systems. The computational systems we consider are *evolvable*, i.e. may reconfigure their structure and programs at run-time. Examples include business process modelling [9], adaptive query processing over changing databases [7] and data structure repair [4]. In [1], a logical framework for describing such systems is introduced. The underlying logic of this framework allows us to build a simulation engine for executing system specifications. This engine uses automated theorem proving technology to determine the satisfiability of logical formula sets as well as the deducibility of a logical formula.

The architecture of evolvable systems that we employ allows the 'localisation' of monitoring and evolution. Components in a system may have 'supervisors' which are themselves components and which monitor their 'supervisees' and may evolve them if required. Such supervised components may be assembled hierarchically, with supervisors at each level of the hierarchy if necessary (cf. Archware [14]).

In the logical framework [1], the state of a component is a set of ground atomic formulae that describe the current properties of the system. An action operates on the state in a style similar to STRIPS [8], changing the formula set by adding and deleting formulae. Models consist of sets of interpretations of the theories of each component. This revision-based method of description, which is common in planning and other AI applications, should be contrasted with the use pre and post-conditions [10]. This revision-based approach leads to a simple mechanical execution process, which we employ to build a simulator. This mechanisation makes intensive use of automated theorem proving (ATP) technology. Some issues relating to the appropriateness of the technology are:

- Changing verification requirements: ordinary actions affect the state only, while evolutionary actions may involve changing a component theory as well as its state. Thus as a system runs, theorem proving takes place in a highly dynamical setting.

- Determining the applicability of actions: an executed action may have a set of logical formulae as preconditions. A valid application of an action requires that (a) the preconditions are derivable from the system state and the component theory, and (b) the resultant state is consistent.
- Handling meta-logics: the supervisor's state is at a *meta-level* to that of the supervisee allowing the supervisor to hold facts about the supervisee. the meta-level logic of the supervisor's state may hold facts about the supervisee. These facts need to be consistent in the supervisor-supervisee pairing at all times during execution.
- Verifiability for minimum models: The revision-based logic used in system specifications is based on a notion of 'minimum model' [1] and we require that deducibility is relative to this minimality requirement. This requirement is related to the notions of the closed world assumption and circumscription in AI [13].

The logic-based simulation imposes a range of requirements on any theorem prover including the ability to:

- handle large sets of formulae for realistic systems,
- determine satisfiability of a set of formulae and deducibility of formulae from a given set of axioms.
- run unassisted, as opposed to guided, to make it possible to run large simulations that generate a high number of proof calls possible.
- construct models, not just establish satisfiability of a given formula set. Models are used to instantiate free variables in formulae.
- extract 'support sets' of formulae for proofs: States are often large sets of formulae, but those required to establish a property may be a small subset (often various different minimal 'support sets' may exist). Identifying support sets can aid proof caching (see below).

We have experimented with several theorem proving systems for this application:

- Paradox - from Chalmers University, a model finder for first-order logic [2].
- iProver - from The University of Manchester, an instantiation-based prover [11].
- Vampire - from The University of Manchester, a very fast resolution-based first-order theorem prover [15].

One point about the dynamics of theorem proving invocation needs explanation: during simulation, most individual actions cause a small change to the system, with the occasional evolution action that changes and reconfigures a system. On the other hand, actions generate multiple proof obligations that may duplicate prover dispatches. The execution of an action may trigger up to seven proof obligations for some action types. Furthermore, the overhead of discharging proof obligations is significant. The simulation of a simple system eventually spends a substantial amount of time communicating theories and proof results to and from the theorem prover. However, by examining simulations, we note that most changes do not affect state consistency, and in many cases proof obligations are duplicated or are trivially resolved.

In order to reduce the overhead of theorem proving various forms of proof caching have been employed. In general terms, proof caching is expensive. However, our particular application

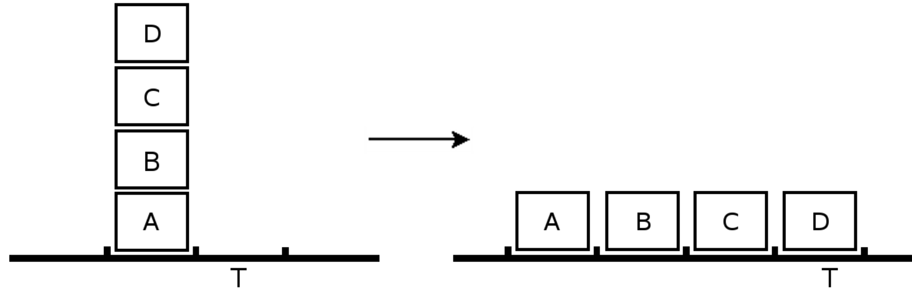


Figure 1: The scenario on the left shows the initial blocks world state, with a table having a capacity of 2 (i.e at most two blocks directly on the table) a tower of blocks. The scenario on the right shows a goal state that could be achieved by demolishing the tower and using a table of capacity 4.

enables relatively simple approaches to speed up proof matching. The result is the elimination of a significant number of proof calls and dramatic speed-ups of simulations. In most cases more than 60% of proof obligations are eliminated with a corresponding increase in performance.

This framework uses a non-standard type of inference relative to a minimum model determined by a set of 'observable formulae'. This is a generalisation of the notion that absence of ground atoms indicates falsehood. The requirement to reason in 'minimum models' [1] is handled in simple cases by a completion process in which we close observable predicates. The general case of reasoning in minimum models combines this completion process with deducibility, but is not required for the system specifications we consider here.

In the next section, using an example, we will illustrate how theorem provers are used during the simulation of an evolvable system, as well as the different between normal executions and evolutionary execution. In section 4 we describe a caching technique that eliminates a substantial number of proof dispatches. Finally, we discuss the results of experiments with several different types of theorem provers in section 4 and draw conclusions in section 5.

2 Logical Simulation of Evolvable Systems

We illustrate how theorem provers are used in our logic-based simulation through the following example of an evolvable blocks world [19]. This consists of a number of blocks which may be on each other or on a table. Actions may move blocks around the world. A theory for a blocks world is defined in Figure 2. This is a simplified theory for illustration purposes. A complete axiomatisation of the blocks world is given in [3] and is used in the simulation. We will show the simulation of a simple blocks world example and then illustrate how a supervisor may be introduced to overcome its limitations.

The blocks world theory $BlocksWorld$ defines a finite set of blocks $\{A, B, C, D\}$ and a single table T. The 'BWC' axiom defines constraints on the predicate 'on'. The constraint 'TableSize' is a parametric formula that initially defines tables with a capacity of 2, i.e at most two blocks may be directly on the table. The state of a blocks world component is recorded using ground atoms built from the binary predicate 'on'. We distinguish between 'observation' predicates and 'abstraction' predicates. A state is described using only observation predicates. Only positive atoms of an observation predicate may be present in the state, those omitted are assumed to be false. The schema defines the abstraction predicate 'free' that will be used to deduce the

BlocksWorld

Types	Observation Predicates
$Blocks \stackrel{dfn}{=} \{A, B, C, D\}$	$on : Blocks \times Objects$
$Tables \stackrel{dfn}{=} \{T\}$	**Abstraction Predicates**
$Objects \stackrel{dfn}{=} Blocks \cup Tables$	$free : Objects$

Constraints

$$BWC \stackrel{dfn}{=}$$
$$\forall b, b_1, b_2 : Blocks, o_1, o_2 : Objects \cdot$$
$$\neg on(b, b) \quad \wedge$$
$$on(b, o_1) \wedge on(b, o_2) \Rightarrow (o_1 = o_2) \quad \wedge$$
$$on(b_1, b_2) \Rightarrow (\exists o : Objects \cdot on(b_2, o))$$
$$\ldots$$

$$TableSize(T, 2) \stackrel{dfn}{=}$$
$$(\exists b_1, b_2 : Blocks \cdot on(b_1, T) \wedge on(b_2, T) \wedge (b_1 \neq b_2)) \Leftrightarrow \neg free(T) \quad \wedge$$
$$\forall b_1, b_2, b_3 : Blocks \cdot on(b_1, T) \wedge on(b_2, T) \wedge on(b_3, T) \Rightarrow$$
$$((b_1 = b_2) \vee (b_2 = b_3) \vee (b_1 = b_3))$$

$$BlockSize(1) \stackrel{dfn}{=}$$
$$\forall b : Blocks \cdot (\exists b_1 : Blocks, o : Objects \cdot on(b_1, b) \wedge on(b, o)) \Leftrightarrow \neg free(b) \quad \wedge$$
$$\forall b_1, b_2 : Blocks \cdot on(b_1, b) \wedge on(b_2, b) \Rightarrow (b_1 = b_2)$$

Actions

$move(x : Blocks, y, z : Objects)$	
pre	$\{on(x, z), free(x), free(y)\}$
add	$\{on(x, y)\}$
del	$\{on(x, z)\}$

Initial State

$\{on(D, C), on(C, B), on(B, A), on(A, T)\}$

Program

$move(D, C, T); move(C, B, T); move(B, A, T)$

Figure 2: The Blocks World schema. Some axioms to assert the non-circularity of 'on' have been omitted from this paper for brevity. A full axiomatisation of the blocks world is given in [3].

availability of free space on the table. Abstraction predicates are defined using the constraints of the theory and may be deduced during simulation.

The component has the single action definition, 'move', that moves a block x from an object y to an object z. The action is conditional on the set 'pre'. The action is performed on the state in a revision-based manner by deleting the atoms in the set 'del' and adding the atoms in the set 'add'.

The component has an initial state where the blocks form a tower on top of the table as shown in Figure 1. The component is equipped with a program that demolishes the tower by moving all blocks on the table in turn. The program is a sequence of 'move' actions.

When this specification is executed, the first requirement is that it is *consistent*, meaning that its constraints and state together are consistent.

After checking the component's consistency, the component's program is run. The first action to be executed in the program is $move(D, C, T)$, which moves the the topmost block D in the initial state to the table T. The action move has the precondition set *pre* which checks that both the block and the receiving object are $free$. *Precondition checking* requires a theorem prover to establish deducibility of the preconditions from the state formulae and the component theory. If this is successful, the action revises the state by adding and deleting formulae as defined in the schema. The resultant state in turn needs to be checked for consistency with the component's theory to ensure that no action renders the component inconsistent.

In order to continue demolishing the tower an attempt is made at the second action in the program, $move(C, B, T)$. This will fail because the precondition cannot be met. The precondition $free(T)$ cannot be deduced as the current blocks world theory is restricted to just towers of blocks on the table as defined in $TableSize$. The only way to change this is to alter the theory. To achieve this, we introduce another system that monitors this system and has the ability to change the blocks world specification. We refer to this new system as a supervisor.

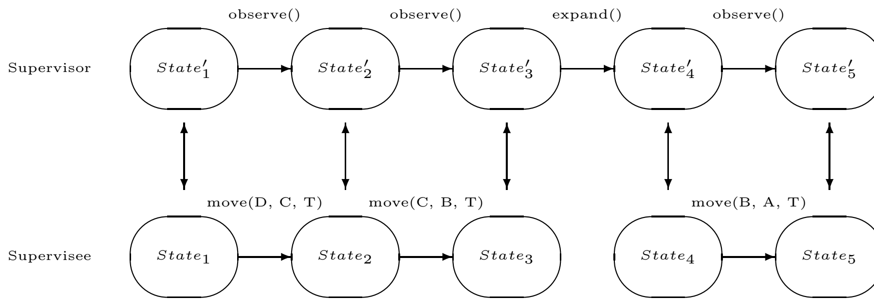


Figure 3: A paired execution trace

$BlocksWorldSupervisor$ in Figure 4 presents such a supervisory system. It records properties of its $BlocksWorld$ supervisee using meta-level predicates. The *holds* predicate is used to express properties of the supervisee. If $holds(\phi)$ occurs in the supervisor's state, then ϕ should be provable in the supervisee's state. The supervisor can use this predicate to query and monitor the supervisee. Similarly, the *constraint* predicate reflects the supervisee's constraints. The supervisor monitors the $BlocksWorld$ using the action *observe* that tests formulae at the supervisee level. It also has the action *expand* which makes use of the evolution predicate *evolve*. The predicate *evolve* induces change at the supervisee level by revising the set of constraints that it has.

Using the *observe* action, the supervisor queries the state of the supervisee and detects when the table has an insufficient number of slots using the object level formula $free(T)$ as reflected at the supervisor level. This is a paired execution of the two example programs where the execution traces of the supervisor and the supervisee programs run in synchrony and are related by a meta-view relationship as depicted in Figure 3.

The supervisor monitors the supervisee and only intervenes when all all space on the table has been used, i.e when at the supervisee level $\neg free(T)$ holds. In this case, the *expand* action alters the $BlocksWorld$ by replacing the $TableSize$ constraint with one that allows for one more space. This is done using the *evolve* predicate, which introduces changes given by the supervisor to the supervisee. In this example, the change is the replacement of a constraint. The supervisor also has the ability to alter the state, redefine predicates or actions, or reconfigure the supervisee with a new component structure.

Meta-level conditions reflected by the predicate *holds* are checked by firing proof obligations

BlocksWorldSupervisor META TO *BlocksWorld*

TYPES
$ConfigName$
FUNCTIONS
$s : ConfigName \to ConfigName$

OBSERVATION PREDICATES
$current : ConfigName$
$holds$: FORMULA $\times$ $ConfigName$
$constraint$: CONSTRAINTNAME
$evolve$:
ATOMS $\times$ ATOMS $\times$
CONSTRAINTNAMES $\times$
CONSTRAINTNAMES $\times$ $ConfigName$

CONSTRAINTS
...
ACTIONS

observe(P : FORMULAE)	
pre	$\{current(c)\}$
add	$\{holds(p, s(c)) \mid p \in P\} \cup \{current(s(c))\}$
del	$\{current(c)\}$

expand($n : Int$)	
pre	$\{current(c), constraint(TableSize(T, m)), m < n\}$
add	$\{current(s(c)), holds(free(T), s(c)),$ $evolve(\{\}, \{\}, \{TableSize(T, n)\}, \{TableSize(T, m)\}, \{\}, \{\}, s(c)),$ $constraint(TableSize(T, n))\}$
del	$\{current(c), constraint(TableSize(T, m))\}$

INITIAL STATE
$\{current(c_0)\}$

Figure 4: The Blocks World Supervisor schema

using the supervisee's theory. The act of changing the supervisee's constraint using the *expand* action is called an *evolution* step. Theory consistency checks are necessary after performing the evolution for both components to disallow evolutions that lead to inconsistent theories.

Finally, component programs may have guarded instructions in which an instruction is executed only if its guard can be proved. We conclude this section by listing the cases where a theorem prover is invoked during the simulation of a component:

1. **Consistency** in the theory and state of each component, checked before and after each action. Models constructed in consistency checking may also be required.

2. **Precondition testing** by proving action preconditions.

3. **Guarded choice checking** by proving the guards.

4. **Meta-level checking** that the supervisee's reflection in the supervisor is correct, done before and after each evolution.

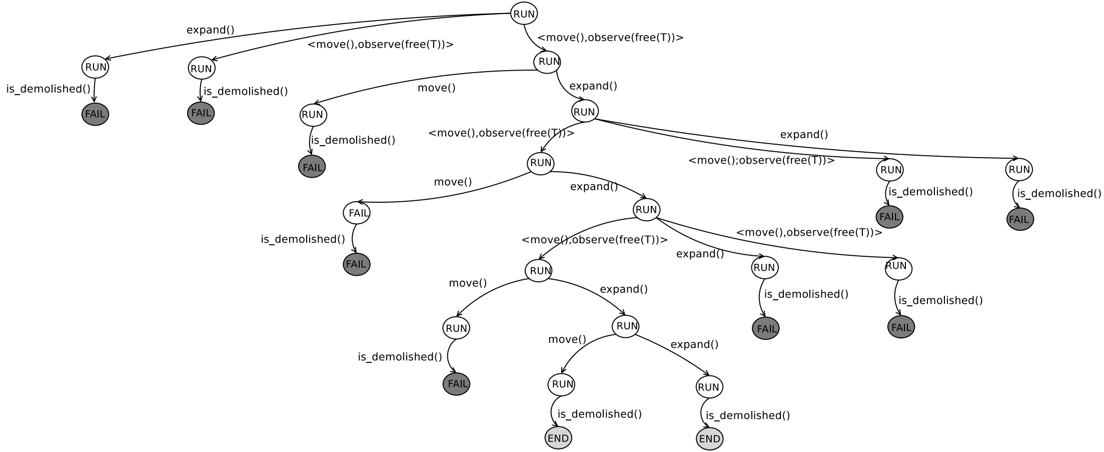


Figure 5: A complete run of the blocks world

5. **Post-evolution consistency**, checking that evolutions don't produce inconsistent specifications.

In the above list, items 1 and 5 invoke theorem provers for satisfiability checking, while items 2, 3 and 4 invoke derivability checks.

2.1 System Simulation

In the example, the pairing of $BlocksWorldSupervisor$ and $BlocksWorld$ specifications will, when executed, ensure the complete demolition of the tower. The supervisor intervenes repeatedly to evolve the supervisee and expand the table whenever it cannot accommodate any more blocks. To achieve this, the supervisor is equipped with the following program:

$([$
$\quad |observe(free(T))$
$\quad |(observe(\neg free(T)); expand())$
$])*;$
$is_demolished();$

This program is an iteration of an non-deterministic choice instruction that queries whether the table is free after every supervisee move action. The *expand* action is only performed when the table is observed to be not free. The *is_demolished* action performs a logical check to determine whether the demolition objective was successfully reached.

A complete run of this paired simulation will result in a configuration trace tree as shown in figure 5. Each node in this tree represents a state of the system configuration. This simulation is exhaustive as it looks for all possible runs of the pair of the components (this search can either be depth-first or breadth-first). The dark nodes indicate actions that have failed: this occurs when a supervisee's *move* action is not possible, or when an evolution is necessary. The grey nodes represent successful runs where the run successfully demolished the tower. Following the nodes in the trace tree, these successful runs occur when each block move is immediately preceded by an evolution to expand the table size.

$$
\begin{array}{lcl}
Cache & \stackrel{dfn}{=} & ComponentCacheKey \rightarrow ComponentCache \\
ComponentCacheKey & \stackrel{dfn}{=} & SchemaID \times ConstraintNames \\
ConstraintNames & \stackrel{dfn}{=} & ConstraintName^{*} \\
ComponentCache & \stackrel{dfn}{=} & State \rightarrow ProvenFormulae \times UnprovenFormulaeSets \\
State & \stackrel{dfn}{=} & Atom^{*} \\
ProvenFormulae & \stackrel{dfn}{=} & Formula^{*} \\
UnprovenFormulaeSet & \stackrel{dfn}{=} & FormulaeSet^{*} \\
FormulaeSet & \stackrel{dfn}{=} & Formula^{*}
\end{array}
$$

Figure 6: Cache Structure

3 Caching and Eliminating Proof Obligations

Simulating specifications can generate a substantial number of proof obligations. A basic non-supervisory action generates two proof obligations for every sub-component that it modifies, but an evolution action generates seven proof obligations for each pair of components that it affects. The simulation of the previous relatively simple blocks world generates 35 deducibility checks and 29 satisfiability checks. Firing an external theorem prover and communicating theories and results uses a substantial amount of time relative to the simulation execution time. Given that in any realistic simulation very large states will result and that the performance of a theorem prover is typically non-linear with the size of the state, it is important to reduce the number of dispatched proof obligations. Several techniques can be used to optimise execution and discharge some of the proof obligations without the need to call an external theorem prover.

By examining simulations, we note that the most commonly executed actions are of a basic type, which are occasionally interrupted by the rarer large system reconfigurations that alter component theories or replace components. Most actions do not affect the axioms of a component and therefore do not affect its internal consistency. A substantial number of consistency checks can therefore be eliminated by storing the consistency results of previous prover invocations. A component is given a 'consistent' flag that is set once and is reset when actions changes are deemed to affect this consistency.

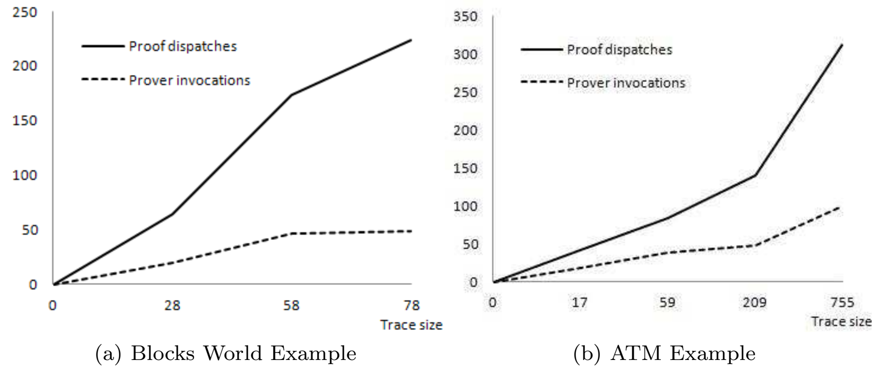
The state of a component contains only positive atoms of observation predicates. Therefore, in some cases, when preconditions are themselves ground observable atoms, derivability reduces to testing membership using symbolic equality. The minimum model interpretation means that the absence of an atom of an observation predicates indicates its falsehood. This allows us to reduce preconditions that rely heavily on observable predicates. It is often the case that the precondition set is reduced to either the empty set (i.e preconditions are met) or to $false$, eliminating the need for any external theorem prover.

3.1 Caching of Prover Results

A cache structure suitable for this application is depicted in Figure 6. Properties of our framework make this cache structure relatively efficient.

The cache stores the prover invocation results for each component separately. To perform lookup, a key for the cache consists of the name of the component together with the names of its constraints. Only the signatures of the constraints are stored, e.g. $TableSize(T, 2)$. This eliminates the need to store whole formulae and simplifies component cache lookup. An example of a cache key is

$$(BlocksWorld, \{TableSize(T, 2), BlockSize(1)\})$$



(a) Blocks World Example (b) ATM Example

Figure 7: Prover invocation elimination using the caching technique

Each component is then associated with all states that it may have had in the past. For each state that a component may have been in, the list of proved formulae is stored. Formulae lookups are done syntactically, so this is efficient for looking up previously proved ground atoms. The cache also stores the sets of formulae that were previously disproved given a state.

Looking up formulae in this cache is undertaken as follows. Given a proof obligation for the formulae set S, once a cache key has been matched together with a state, S is reduced to the set S' by eliminating formulae that were previously proved. If S' is the empty set then this is considered a cache hit as proved formulae and a prover invocation is not necessary. if S' is not empty, it is compared with every set of the previously unproven formulae. If S' contains any of these sets then this is also a cache hit as unproven formulae. If S' is not matched by any unproven formulae set, an external prover invocation is necessary to determine the deducibility of S'. If the prover returns a proved result then S' is merged with the proved formulae in the cache. if S' is not proved it is added to the set of previously unproven formulae sets.

3.2 Performance

The caching technique eliminates a substantial number of proof obligations. In the blocks world example, 20 out of 35 deduction requests (57%) and 24 out of 29 satisfiability requests (82%) are eliminated. The running time is reduced from 16 seconds to 5 seconds on an AMD Athlon 2000+ processor with 1GB of RAM. Figure 7 shows the number of total proof dispatches and actual external prover invocations for specifications that generate increasing program traces. The ATM example, given in [1], models an evolvable banking and automated teller machines system that enforces several layers of security using evolvable pairs. The trace size is an indication of the number of actions being performed. The figure shows that the longer the simulation the more beneficial caching becomes, with some simulations eliminating over 90% of all proof obligations.

4 Comparing Theorem Provers

The simulator converts theories to classical untyped first order logic in the TPTP3 [18] format so CASC (CADE ATP System Competition) [17] can be used. Three provers were used in this study:

Trace size	Paradox	iProver	Vampire
28	5	12	15
58	8	18	22
78	11	24	32

Figure 8: Simulation time (in seconds) using different provers. Note that satisfiability checking was done with Paradox when using Vampire.

- **Paradox** [2]: a finite model generator that flattens first order logic formulae into proposition clauses, then uses the MiniSat [6] SAT solver to solve the resulting problem.
- **Vampire 10** [15]: a fast resolution-based theorem prover.
- **iProver** [11]: an instantiation based prover that combines first order reasoning with a SAT solver (using MiniSat [6]).

The table in Figure 8 shows the running time in seconds of the simulation using these provers. The measure of the complexity of a simulation is the trace size, which grows approximately linearly with the number of proof obligations being made.

For the blocks world example, Paradox was able to undertake both the satisfiability checks and the proofs. It is fast at finding models for axiom sets and for checking counter satisfiability [16]. Overall, it was the best performer of the three theorem provers for this application. Its ability to determine the deducibility or otherwise the counter satisfiability of formulae makes it the most suitable choice for this application.

Although Vampire's resolution is fast, it is not appropriate to establish the counter satisfiability of non-theorems, nor could it establish the satisfiability of theories in the above example. This necessitates the use of other theorem provers for these purposes when using Vampire.

iProver is unique because it can do both resolution reasoning and SAT checking, but they are done successively with manual options to turn off either features. Its resolution reasoning is fast at establishing the non-satisfiability of sets of formulae and for deriving theorems, while its SAT solving mode is fast at establishing the satisfiability of sets of axioms and the counter-satisfiability of non-theorems. iProver can spend time unnecessarily using one of its modes for an input that is best suited for the other mode.

5 Conclusion

This paper gave an overview of the practical aspects of using theorem provers for the simulation of the evolvable systems framework presented in [1]. This framework differs from rewriting tools such as Maude [12] in the fact that it allows for a supervisory model that improves usability and the separation of concerns [5]. In the simulation, theorem provers are used to deduce formulae from an axiom set and for establishing the satisfiability of sets of formulae. The simulation generates a large number of proof obligations. However, a caching technique was used to eliminate a substantial number of prover invocations and speed up simulation.

Three theorem provers were used in this study: Paradox [2], iProver [11] and Vampire [15]. Paradox's model finding was best suited for establishing satisfiability of axiom sets as well as the counter satisfiability of non-theorems. Although vampire's resolution is fast, it could not establish the counter-satisfiability of non-theorems. iProver's manually changeable two modes, resolution and SAT solving, could perform both tasks. In the future, running multiple theorem provers in parallel could be performed to exploit the strength of each prover.

References

[1] H. Barringer, D. Gabbay, and D. Rydeheard. Modelling evolvable component systems: Part I: A logical framework. *Logic Jnl IGPL*, 17(6):631–696, 2009.

[2] K. Claessen and N. Sörensson. New techniques that improve MACE-style model finding. In *Proc. of Workshop on Model Computation (MODEL)*, 2003.

[3] S.A. Cook and Y. Liu. A complete axiomatization for blocks world. *Journal of Logic and Computation*, 13(4):581, 2003.

[4] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 176–185, 2005.

[5] E.W. Dijkstra. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[6] N. Eén and N. Sörensson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919*, pages 502–518, 2003.

[7] K. Eurviriyanukul, A. Fernandes, and N. Paton. A foundation for the replacement of pipelined physical join operators in adaptive query processing. *Current Trends in Database Technology–EDBT 2006*, pages 589–600.

[8] R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[9] R.M. Greenwood, I. Robertson, B.C. Warboys, and B.S. Yeomans. An evolutionary approach to process system development. In *Proceedings of the International Process Technology Workshop, Villard de Lans (Grenoble)*. Citeseer, 1999.

[10] C. B. Jones. *Systematic Software Development using VDM.* International Series in Computer Science. Prentice-Hall, 1990.

[11] K. Korovin. iProver - an instantiation-based theorem prover for first-order logic (system description). In *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.

[12] P. Lincoln M. Clavel, S. Eker and J. Meseguer. Principles of maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[13] J. McCarthy. Circumscription–a form of non-monotonic reasoning. *Artificial intelligence*, 13(1-2):27–39, 1980.

[14] F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, and C. Occhipinti. ArchWare: Architecting Evolvable Software. In *Proceedings of the 1st European Workshop on Software Architecture (EWSA'04), European Projects in Software Architecture - Invited Paper*, LNCS, pages 257–271, St Andrews, UK, May 2004.

[15] A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002.

[16] G. Sutcliffe. The 4th IJCAR Automated Theorem Proving Competition. *AI Communications*, 22(1):59–72, 2009.

[17] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[18] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[19] T. Winograd. *Understanding Natural Language*. Academic Press, New York, 1972.