CardS4: Modal Theorem Proving on Java Smartcards

Rajeev Prabhakar Goré[†] and Phuong Thê Nguyên[‡]

Abstract — We describe a successful implementation of a theorem prover for modal logic S4 that runs on a Java smart card with only 512 KBytes of RAM and 32KBytes of EEPROM. Since proof search in S4 can lead to infinite branches, this is "proof of principle" that non-trivial modal deduction is feasible even on current Java cards. We hope to use this prover as the basis of an on-board security manager for restricting the flow of "secrets" between multiple applets residing on the same card, although much work needs to be done to design the appropriate modal logics of "permission" and "obligations". Such security concerns are the major impediments to the commercial deployment of multi-application smart cards.

Keywords — security of mobile code, modal deduction

1. Introduction

Smart cards are credit-card sized pieces of plastic with an embedded silicon chip. Smart cards are either memory cards, which cannot be programmed, or microprocessor cards, which contain a small amount of RAM and disc (EEPROM) on the card itself. A card reader/writer is required to provide power to the card, to provide a clock signal, and to act as an interface between the card and the terminal (a PC, an ATM machine, a public telephone, or even a mobile telephone).

Java cards are smart cards that contain a (downsized) Java platform, installed by the manufacturer, thus allowing users to download Java applets and run them on the card. Java cards can therefore provide multiple applications such as electronic purse, credit card, passport, loyalty programmes, all residing on the same card.

While exchanging data securely already poses a number of problems (this prompts the need for cryptographic protocols, examined in a number of other papers in this issue), exchanging, even if only down-loading *programs* entails quite a number of new problems. It is all too easy to break every security policy by just down-loading one bad applet and letting it loose on the card. One example, not specific to cards, is the BrownOrifice applet [CER00], a Java applet that installs on any PC that down-loads and serves its entire file system to any outside attacker. Another is an attack allowing an outside intruder to register a bank transfer via the Quicken home-banking software [Bru97].

The purpose of this paper is, first, to give a short survey of proposed techniques to enforce security in settings where applications can be loaded or down-loaded, and more specifically of Java card related techniques. Orthogonally, *logics*, and more specifically modal logics, have been used to specify security policies. We review the use of logics in this context. One challenge here is to be able to prove automatically formulae in sophisticated modal logics, efficiently, and—in the Java card context at least—under sparse memory resources. We shall demonstrate how this can be done for the logic **S4**—not yet the kind of logics we would like to deal with, but already one which is known to pose non-trivial problems. (Technically, this is because the transitivity of **S4** frames, as opposed to, say, **K** frames requires loop checks that are usually memory-consuming.) The Java cards used in this project were the GemXpresso RAD Protyping card, containing a 32-bit microprocessor with 512 bytes of RAM, 32 KBytes of Flash EEPROM and 8 KBytes of ROM. As of 2001, this is state of the art in Java card technology, and gives an indication of how little memory is available on Java cards.

The paper is set out as follows. We spend some time in Section 2. surveying method for ensuring security of mobile code, and Java card applets in particular. This includes discussions of several models or techniques, including the notion of non-interference, verification by typing, by static analysis of programs and by formal proofs in specific logics. We argue that being able to prove formulae of modal logics on-card is a promising security enforcement technique. The rest of the paper shows that, in principle, even complex modal logics with transitive frames can be handled on a card, by studying the prototypical transitive modal logic, **S4**. Section 3. describes the logic **S4** and the basics of modal theorem proving using tableaux. Section 4. explains the design of our prover `CardS4`, while Section 5. refines this by explaining the precise data structures that allow our prover to run in tiny memory spaces. Section 6. describes our implementation, and Section 7. presents test results. We conclude in Section 8.

ACKNOWLEDGEMENTS: We are grateful to Didier Tollé of Gemplus Australia for donating the hardware necessary to carry out this project.

2. Java cards and security of mobile code

2.1. Java cards

Current Java cards are preprogrammed to contain applets by the manufacturer for the card vendor, typically a bank (for credit and debit cards), or an airline (for frequent flyer cards). But if Java cards are to succeed then a card carrier

† Supported by a Queen Elizabeth II Fellowship from the Australian Research Council.

‡ Supported by an RSISE Summer Research Scholarship.

must be able to down-load new applets onto an existing card "just in time", or even merge existing cards into one card. This would mean that multiple applets from different vendors would reside on the same card.
The single biggest problem with this scenario is that of security. How can we guarantee that a simple query to the drivers licence section of the card for identification purposes (say) will not steal money from the card's electronic purse? If new applets are to be down-loaded then how can the vendor of applet A ensure that a competing vendor's applet will not be down-loaded at a later stage and steal information from applet A? Alternatively, applets A and B may trust each other to some extent, and therefore share some information. But if applets B and C enjoy a similar trust relationship, how can A be sure that B will not tell C information which it has obtained from A [Gir99, PB00] ?
Many methodologies for guaranteeing such security have been investigated, but almost all of them involve a trusted "third" party. For example, the bank applet may be signed using a digital signature obtained from the government that certifies that the applet really did originate from the bank in question. The digital certificate is decoded by the card's on-board digital signature chip and the applet is allowed to access the card's electronic purse. But the need for a certification agency and a certification procedure makes this avenue cumbersome.

2.2. Enforcing security policies for mobile code

An alternative methodology that involves no third parties is for card owners to implement a personal security policy using some international standard "language for security". The electronic purse applet installed on the card may come with such a built in security policy which the user is prompted to tailor to his or her needs. Another applet which wishes to access the electronic purse must now pass a challenge determined by the level of security chosen by the card user.
As new applets are added to the card, they are slotted into this set up either explicitly by the card user, or by some implicit default method. The simplest method is to use some form of access control list as is done by the Smart Card for Windows system (`http://www.microsoft.com/smartcard`), which uses simple propositional logic in its access control lists. A more sophisticated approach is to use a hierarchy with the "public" applets at the bottom, the "private" applets at the top, and the others in between these two extremes in some partial order [Gir99, PB00]. This is similar to the Bell-La Padula model of military security [BL75], which is the basis of the access-right policy of operating systems like Unix. Each applet A is given an *accreditation* $acc(A)$ from some partially ordered set, while each object O on the card has an *access right* $right(O)$; the security policy is that A can only access O if $acc(A) \geq right(O)$; also A can only modify O, storing the contents of object O' into O, if $acc(A) \geq right(O)$ and $right(O') \leq right(O)$. This is Bell and La Padula's star condition; without it, A might unwittingly declassify O' by storing its contents into O, thereby allowing non-accreditated applets to access the contents of O' by subsequently reading that of O. These conditions can be enforced at run-time. The relationship between access control lists and object-level access rights is essentially a matter of whether access rights are stored in a centralized way or on a per-object basis. The precise relation between these and other so-called *trust management models*, such as capabilities, is analyzed in detail in [CDM01].
The latter paper in particular addresses the difficult matters of handling *delegation*, whereby a subject (an applet, in our context) is allowed to act in the name of another for some designated objects, and *revocation*, whereby subjects are deprived of their accreditation. In the latter case, think of a vendor applet that will only provide a service to the card owner while she holds an annual subscription to that service. Apart from revocation, trust management policies are still limited in the way they deal with dynamic change. In particular, how should the ordering $\geq$ be modified when a new applet is down-loaded? The Bell-La Padula model and its variants all assume a fixed ordering. But if down-loading an applet is allowed to modify the access rights ordering, possibly adding new accreditations or object access rights, a new policy is needed to prevent abuses; e.g., we should not allow the down-loading process to accept applets claiming to introduce a new accreditation greater than all pre-existing ones. Also, there should be some mechanism to enforce that the modified ordering still is an ordering.
Even then, solving these problems would not solve problems related to *transitive* information flows. Take the shared secrecy example above, where applet A trusts B, and B trusts C, but A does not trust C [Gir99, PB00]. For example, A might be a loyalty applet, and B might be a banking applet originating from a bank that has business deals with A's originator, so that if the card owner has accumulated enough loyalty points through A (think frequent flyer miles), then B will offer the card owner some added payment facilities. Now C might be the on-card part of an account management programme, which will need to access information from B, and will be trusted by B to do so. Then C can learn about the degree of loyalty of the card owner vis-a-vis A's originator by examining payment facilities offered by B: although A and C's originators never signed a deal allowing C to access loyality information from A, C can still get it through interaction with B.

2.3. Non-interference

Checking such properties can be done, at least partially, by checking *non-interference* properties [GM82]. At a basic level, non-interference for some computer system S (an applet, or a collection of applets) means that for every collection of objects O in the system, the value of objects with low access rights should never depend on the value of objects with high access rights. In other words, no observer should be able to tell anything about the values of objects with high access rights by just looking at values of objects with lower access rights. In particular, if an applet's accreditation level is a, its output should be independent of the value of any object with access right $> a$.

Non-interference is similar to the way secrecy and authentication are originally defined in Abadi and Gordon's spi-calculus [AG97]: a message M is *secret* in some protocol $P(M)$ if and only if no outside attacker can tell the difference between running $P(M)$ or running $P(M')$ for some other value M' of the secret. Formally, this is defined by saying that for every process I in the calculus, I parallel $P(M)$ and I parallel $P(M')$ should be may-testing equivalent. (Technically, this also requires the parallel compositions to be enclosed in suitably many (νn) constructs to represent channel, nonce and key generation.) This similarity with non-interference can be used to cast it as a non-interference problem, where the secret M would be assigned some high access right, and we require that non-interference holds assuming that the intruder has strictly lower accreditation. This is used in a typing system for secrecy [Aba99]. A similar but slightly more complicated typing system also exists for authentication [GJ01].

While non-interference seems a promising idea, *checking* non-interference is harder than it seems. Many type systems have been proposed in various restricted settings. One of the most sophisticated is [Smi01], which considers non-interference in the presence of concurrency, where computations have observable durations, and in a probabilistic model.

Type systems for non-interference have to be crafted so that well-typed applets do obey non-interference. The resulting type systems are in general severely restricted as to which applets it will accept as well-typed.

To show what the difficulties are, first examine concurrency. Assume that applet A and applet B are both secure in the sense that none, when run alone, may terminate with some low object containing a value which depends on the initial contents of a high object. Then the parallel composition of A and B might be insecure. For example, consider three objects O_{hi}, O_{lo} and O'_{lo}, with respectively high, low and low access rights. For simplicity, assume that these objects only contain boolean values. Let A store the contents of O_{hi} inside O_{lo}, do nothing for a while, then erase all fields of O_{lo}; for short we write A as the program:

$$O_{lo} := O_{hi};\ \text{sleep};\ O_{lo} := 0;$$

A is secure, since the final value O_{lo}, zero, is independent of the initial value of O_{hi}. Let B do nothing for a while, test whether O_{lo} is true, and if so set O'_{lo} to true, otherwise to false. That is, B is:

$$\text{sleep; if } O_{lo} \text{ then } O'_{lo} :=\text{true else } O'_{lo} :=\text{false};$$

Again, B is secure, since it never reads the value of any high object. Note that, if O_{lo} were replaced by O_{hi} in B, and even though B never copies its value to any other object, the resulting value of O'_{lo} would depend on that of O_{hi}, and B would not be secure; this shows that Bell and La Padula's conditions are in general not enough to ensure non-interference.

The important point in this example is that A and B in parallel are *not* secure: if A first does $O_{lo} := O_{hi}$, then B tests O_{lo}, thus in effect B is computing a value of O'_{lo} that depends on O_{hi}, violating non-interference. This may in fact happen with non-negligible probability, depending on the scheduler. The paper [Smi01] examines more sophisticated interference patterns in which B cannot actually learn from the value of some high object because of timing considerations under a probabilistic model, assuming a probabilistically uniform scheduler. For example, if B sleeps long enough first in the example above, and A and B are started at the same time, then A parallel B will in fact still be secure with high probability.

2.4. Static program analysis

Checking properties of programs, whether security properties or others, can be done through typing, or through dataflow analysis, in general through any static program analysis technique.

One of the most well-known Java related dataflow analysis technique is Java's *bytecode verifier* [LY99]. Every downloaded Java class file, in particular every Java applet, is checked for format conformance first, then names are resolved, then every method in the class file is checked—this is bytecode verification proper. This latter phase checks that all operations are well-typed, that stacks do not overflow, plus a number of other sanity conditions, through a dataflow analysis.

While these checks are absolutely necessary for security (any type confusion error can indeed be exploited to create a security breach [McGF99]), there are two issues that need to be addressed. First, the Java bytecode verifier consumes too many resources to be implemented on a Java card: in particular, the first Java cards did not include any bytecode verifier, and rested solely on cryptographic certificates and a trust relationship with applet issuers; as [Bru97] demonstrates, this is not enough. Second, the bytecode verifier only addresses low-level safety issues (bounds checking, typing), and is far from ensuring any security-related property.

There are at least two different solutions to the first problem. One, inspired from Necula's *proof-carrying code* concept [Nec97], is Rose's *lightweight bytecode verification* [RR98], used in Sun's small-footprint KVM Java virtual machine [Sun00], designed for embedded applications. The idea is to split the bytecode verifier in an off-card part and an on-card part. The off-card verifier actually runs a dataflow analyzer similar to the standard bytecode verifier, except that on success it also outputs a certificate. The off-card verifier is run by the card issuer, who then appends the certificate to the applet. When the applet is down-loaded on the card, it comes with the certificate. The on-card verifier then only checks that the certificate is valid and is a certificate for the given applet. While cryptographic certificates are certainly the simplest form of certificates, the approach of [RR98] is more drastic: the certificates there are (a compressed form of) the typing information that a bytecode verifier needs to compute. It merely remains for the on-card verifier to check that the certificate is consistent with the semantics of all bytecode instructions present in each method.

This takes less time, and more importantly less space than standard bytecode verification. This method applies to essentially any verification method that relies on proving some property of an applet in some formal deduction system (in the large; here typing is thought as a formal system, while Necula considers properties expressed in a variant of the logical framework LF [HHP93] with a few extensions).

The other current way of incorporating bytecode verification into Java cards is Leroy's simplified bytecode verifier. This does not conform to Sun's specification of bytecode verifier as such, and in particular may reject applets that would be accept by the Sun's verifier. However this is repaired by an off-card component which rewrites any applet conforming to Sun's specification into one that will be accepted by the simplified verifier. The point is that the simplified verifier is actually able to run on a standard Java card, despite the severe restrictions on memory resources on cards. The basic intuitions behind this technique, as well as a lucid account of problems and solutions for bytecode verification, can be found in Leroy's paper [Ler01].

The second problem with bytecode verification is that it only addresses low-level issues: typing, stack overflows, notably. It does not address any less trivial security issue such as the transitive flows mentioned earlier, for example. There is still little work on methods for checking more sophisticated security properties. Leroy and Rouaix [LR99] address the problem of verifying, by typing, that a down-loaded applet does not corrupt designated sensitive data on the system it is down-loaded onto. El Kadhi [ElK01, BElK01] applies abstract interpretation methods to design a static analyzer that checks an applet for cryptographic confidentiality preservation properties: the goal is to ensure that designated sensitive data on a card are not leaked to a Dolev-Yao-style intruder (see [CS01]), even though this data may have to be sent out of the card (i.e., properly encrypted). This uses techniques from cryptographic protocol verification.

2.5. Logics

For checking more sophisticated security properties, it is implicit in the above discussion that we need a language to talk about the security properties of interest that can be understood both by card issuers and by on-card verifiers. This language should have a formal semantics. In other words, it should be a logic at large. Proof-carrying code already takes the viewpoint that properties should be specified in a logic, and that proofs should be sent along with the code to avoid costly reconstructions of proofs on the card side—this is a technological choice that may or may not be relevant, depending on the logic and available on-card resources. El Kadhi's work is another example: while the paper [ElK01] does not mention any specific logic, El Kadhi's analyzer actually does deductions in a system of symbolic constraints that approximate the intruder's state of knowledge.

We can also use actual logics to express and check security properties. While this is the approach in Necula's original approach, taking a general logic such as LF might be overkill. In particular, LF provability is undecidable. However, multi-modal propositional logics provide interesting languages that are expressive enough to encode most properties of interest, while usually remaining decidable. Multi-modal propositional logics are now well-established in artificial intelligence research as bases for defeasible reasoning [Shv90], logics of agents [RG93], and logics of authentication [BAN90, Mat97]. Monniaux [Mon01] shows that BAN and GNY logics are decidable, while Massacci [Mas97] gives a tableaux calculus for the (undecidable) logic of access control of [ABLP93]. We refer the reader to [Mon01] for more information on such logics. Multi-modal logics like Propositional Dynamic Logic [Gol87] have also been used to model the changing states of a program. Finally, propositional bi-modal tense logics give a very simple and elegant model of the flow of time [HC96].

Checking that a down-loaded applet meets the security criteria is now reduced to proving, **on-board**, that an appropriate formula is a theorem of the logic used to code the criteria, since this is the only computer that the customer should trust. Let us stress that multi-modal logics are particularly well-suited to this task as most of them are decidable. Consequently, the ability to perform automated multi-modal deduction on Java smart cards may be of use in electronic commerce.

But surely multi-modal deduction is simply too difficult to perform on a smart card with extremely limited resources ? After all, even classical propositional logic is NP-complete, and most multi-modal logics are actually PSPACE-complete!

In [GNg00] automated deduction in bi-modal tense logics was shown to be feasible on a Java smart card. It is reasonably straightforward to extend this work to other multi-modal logics, and hence to logics of knowledge and belief, or to logics of authentication and security. But many of these logics (e.g. PDL) contain operators which are inherently transitive, and transitivity can lead to infinite loops. (This will be illustrated in later sections on **S4**.) In the sequel we show that transitivity is not insurmountable by implementing a prover for the transitive modal logic **S4**. This also shows that, although proof-carrying code-style techniques could be used here as well, they are probably unnecessary.

This work is naturally still far from an on-card prover for a logic of authentication or security, in the style of [BAN90, Mas97]. This work should therefore be thought of as "proof of principle" that a logic-based security policy could be implemented on current Java cards. As the resources and speed of Java cards skyrocket, the task will only become simpler.

3. Syntax, Semantics and Tableaux for Modal Logic **S4**

3.1. Syntax and Semantics for S4

Given a denumerably infinite set of atomic formulae $\mathtt{PRP} = \{p_0, p_1, p_2, \cdots\}$, a formulae φ of modal logic is defined using the following BNF grammar:

$$\begin{array}{lll} p & ::= & p_0 \mid p_1 \mid p_2 \mid \cdots \\ \varphi & ::= & \top \mid \bot \mid p \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \\ & & \mid \Diamond\varphi \mid \Box\varphi \end{array}$$

Propositional symbols in `PRP` denote elementary properties, e.g., "file `accounts` has access right `privileged`", or "user `Joe` has accreditation `standard`", or "`standard` $\geq$ `privileged`". They might be true or false; for example, we may imagine that the first two properties above are true, while the last one is false.

The other connectives are $\top$ (true), $\bot$ (false), $\neg$ (negation), $\wedge$ (and, conjunction), $\vee$ (or, disjunction), $\rightarrow$ (implication), and the modal connectives $\Diamond$ and $\Box$. The latter require some explanation. We may for example understand these connectives in the context of modeling agent knowledge (with one agent a) by letting $\Box A$ mean "a knows (is sure that) A", and $\Diamond A$ mean "a believes A". What the latter means is that a is not sure that A is false, so a accepts A as likely, although a cannot be sure of A.

The formulae of the logic **S4** can also be given another, temporal meaning, in which the truth-values of formulae evolve through time, and $\Box A$ means "from now on, A is always true", while $\Diamond A$ means "A will eventually become true at least once".

These meanings of **S4** formulae are special cases of its *Kripke semantics*. A Kripke frame is a pair $\langle W, R\rangle$ where W is a non-empty set (of worlds) and R is a binary relation over W. A Kripke model $\langle W, R, V\rangle$ is a Kripke frame $\langle W, R\rangle$ augmented with a valuation $V : \mathtt{PRP} \mapsto 2^W$ mapping each atomic formula to the subset of W where they take the value "true". If $w \in V(p)$ we write $w \models p$ and extend this satisfaction relation to arbitrary formulae in the usual way [HC96] as shown in Figure 1 where for any $w \in W$, $R(w) := \{v \in W \mid wRv\}$.

An **S4**-model is a Kripke model where R is both reflexive $(\forall w \in W)[wRw]$ and transitive $(\forall w_1, w_2, w_3 \in W)[w_1Rw_2 \;\&\; w_2Rw_3 \Rightarrow w_1Rw_3]$.

A formula φ is **S4**-*satisfiable* if and only if there exists some **S4**-model with some $w \in W$ such that $w \models \varphi$. A formula φ is **S4**-*valid* if $w \models \varphi$ for every $w \in W$ in every **S4**-model $\langle W, R, V\rangle$.

We illustrate this notion of model on a few **S4** formulae. First, $\Box\varphi \rightarrow \varphi$ is a valid formula. Temporally, this means that if from now on, φ is always true, then φ is true now. For agents, if a knows that φ holds, then φ indeed holds; that is, a does not make mistakes. Another interesting formula is $\Box\varphi \rightarrow \Box\Box\varphi$. Temporally, this means that if φ holds in every future from now, then in every future from now, in every future of this future, φ will again hold. In the world of agents, this is *positive introspection*: if a knows that φ holds, then a also knows that it knows that φ holds. A third important formula is $\Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$, which states that agents can perform deductions: if a knows that φ implies ψ, and a knows that φ holds, then a necessarily knows that ψ holds, too. Finally, we mention the subtle rule of necessitation: if φ is valid, then so is $\Box\varphi$. That is, if φ is always true, typically because there is a proof of φ, then a is sure that it holds. We let the logically-minded reader that the three formulae above and the necessitation rule all hold in any **S4**-model.

3.2. Proof Search in S4 *Using Tableau Calculi*

The problem of deciding whether or not a formula is **S4**-satisfiable is known to be `PSPACE`-complete [Lad77a, Lad77b]. The best known decision procedures use only $O(n^2.logn)$-space [Hud].

The most popular method for implementing theorem provers for **S4** is to use the tableau method [Fit83, Gor99]. This uses the rules of Figure 2, plus their duals for $\neg\Box$ and $\neg\Diamond$ obtained via the equivalences $\neg\Box\varphi = \Diamond\neg\varphi$ and $\neg\Diamond\varphi = \Box\neg\varphi$, and for negations of other connectives obtained via the equivalences $\neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2$, $\neg(\varphi_1 \vee \varphi_2) = \neg\varphi_1 \wedge \neg\varphi_2$, $\neg\top = \bot$, $\neg\bot = \top$. The rules for implication are derived from $\varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$.

$$\frac{X, \varphi_1 \wedge \varphi_2}{X, \varphi_1, \varphi_2}\,(\wedge) \qquad \frac{X, \varphi_1 \vee \varphi_2}{X, \varphi_1 \quad X, \varphi_2}\,(\vee)$$

$$\frac{X, \Box\varphi}{X, \Box\varphi, \varphi}\,(\Box\mathbf{S4}) \qquad \frac{X, \Box Y, \Diamond\varphi}{\Box Y, \varphi}\,(\Diamond\mathbf{S4})$$

Fig. 2. Tableau rules for **S4**

An **S4**-tableau for a finite set of formulae Z is a binary tree of nodes where: the root node contains Z and the children are obtained by an application of some tableau rules for **S4** to the parent node. The rules are applied systematically so that the $(\Diamond\mathbf{S4})$ rule is applied only to a *saturated* node: a node to which all other rules have already been applied. A branch of such an **S4**-tableau is closed if its leaf node contains both φ and $\neg\varphi$ for some formula φ, or if it contains $\bot$; otherwise the branch is open. The whole **S4**-tableau is closed if every branch in it is closed, otherwise it is open. A formula φ is *proved* when there is a closed tableau for the finite set $Z = \{\neg\varphi\}$.

As a first example, we may prove the formula

$$(p \wedge q) \rightarrow (q \wedge (q \rightarrow p))$$

as follows. First, negate this formula to get:

$$p \wedge q \wedge (\neg q \vee (q \wedge \neg p))$$

Formally, we have simplified the negation by pushing negations inside formulae, using transformation rules $\neg\Box A \rightarrow \Diamond\neg A$, $\neg\Diamond A \rightarrow \Box\neg A$, $\neg(A \rightarrow B) \rightarrow A \wedge \neg B$, $\neg(A \wedge B) \rightarrow \neg A \vee \neg B$, $\neg(A \vee B) \rightarrow \neg A \wedge \neg B$, $\neg\top \rightarrow \bot$, $\neg\bot \rightarrow \top$, and removing double negations $\neg\neg A \rightarrow A$. This process ends in a formula where negation is only applied to atomic formulae, a so-called *negation normal form* (NNF).

Then, we may apply rule $(\wedge)$ twice to produce the tableau node $p, q, \neg q \vee (q \wedge \neg p)$. The only rule that applies now is $(\vee)$, yielding two nodes $p, q, \neg q$ and $p, q, q \wedge \neg p$. The first

$$\begin{array}{llllll}
w \models \top & & \text{for every } w \in W & w \models \bot & & \text{for no } w \in W \\
w \models p & \text{iff} & w \in V(p) & w \models \neg\varphi & \text{iff} & w \not\models \varphi \\
w \models \varphi \wedge \psi & \text{iff} & w \models \varphi \text{ and } w \models \psi & w \models \varphi \vee \psi & \text{iff} & w \models \varphi \text{ or } w \models \psi \\
w \models \varphi \to \psi & \text{iff} & w \not\models \varphi \text{ or } w \models \psi & & & \\
w \models \Diamond\varphi & \text{iff} & (\exists v \in R(w))(v \models \varphi) & w \models \Box\varphi & \text{iff} & (\forall v \in R(w))(v \models \varphi)
\end{array}$$

Fig. 1. Kripke semantics for **S4**

one is closed. The only rule that applies to the second is $(\wedge)$, yielding $p,q,q,\neg p$, which is closed. Each branch is closed (contradictory): this terminates the proof. To sum up, this proof is written:

$$\dfrac{\dfrac{p \wedge q \wedge (\neg q \vee (q \wedge \neg p))}{p,q,\neg q \vee (q \wedge \neg p)}\,(\wedge)}{p,q,\neg q \qquad \dfrac{p,q,q \wedge \neg p}{p,q,q,\neg p}\,(\wedge)}\,(\vee)$$

One example we shall use in the sequel is:

$$\Box\Diamond p \wedge \Diamond\Box q \to \Diamond(q \wedge p) \tag{1}$$

Its temporal meaning is "if p is always such that it will become true later on, and if q eventually becomes true then remains true forever, then p and q will eventually become true simultaneously". Its meaning based on agent knowledge is "if I know that I believe p, and if I believe that I know q, then I believe p and q". This formula is **S4**-valid, as can be shown by looking at its Kripke semantics.

Alternatively, we prove it as follows. A tableau proof starts with the NNF of its negation:

$$\Box\Diamond p \wedge \Diamond\Box q \wedge \Box(\neg q \vee \neg p) \tag{2}$$

A closed tableau is then:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Box\Diamond p \wedge \Diamond\Box q \wedge \Box(\neg q \vee \neg p)}{\Box\Diamond p, \Diamond\Box q, \Box(\neg q \vee \neg p)}\,(\wedge)}{\Box\Diamond p, \Diamond p^{(a)}, \Diamond\Box q^{(b)}, \Box(\neg q \vee \neg p), \neg q \vee \neg p}\,(\Box\mathbf{S4})}{\Box\Diamond p, \Box q, \Box(\neg q \vee \neg p)}\,(\Diamond\mathbf{S4})}{\Box\Diamond p, \Diamond p^{(a)}, \Box q, q, \Box(\neg q \vee \neg p), \neg q \vee \neg p}\,(\Box\mathbf{S4})}{\Box\Diamond p, p, \Box q, \Box(\neg q \vee \neg p)}\,(\Diamond\mathbf{S4})}{\Box\Diamond p, \Diamond p^{(a)}, p, \Box q, q, \Box(\neg q \vee \neg p), \neg q \vee \neg p}\,(\Box\mathbf{S4})}{\begin{array}{c}\Box\Diamond p, \Diamond p^{(a)}, p,\\ \Box q, q, \Box(\neg q \vee \neg p),\\ \neg q\end{array} \qquad \begin{array}{c}\Box\Diamond p, \Diamond p^{(a)}, p,\\ \Box q, q, \Box(\neg q \vee \neg p),\\ \neg p\end{array}}\,(\vee) \tag{3}$$

The first (topmost) use of rule $(\Box\mathbf{S4})$ generates a node where there are two $\Diamond$-formulae, written here with superscripts (a) and (b). Then we may use $(\Diamond\mathbf{S4})$ in two ways, using (a) or (b). The proof above uses (b); in fact there is no proof where we would use (a) instead at this point. We retrieve (a) below in the same proof: this is where it is used with $(\Diamond\mathbf{S4})$. Although (a) is again regenerated below, we do not use $(\Diamond\mathbf{S4})$ again. The final rule is $(\vee)$, which closes the whole tableau. This proves (1).

Theorem 1 (Soundness and Completeness) The finite set $\{\neg\varphi\}$ has a closed **S4**-tableau iff the formula φ is **S4**-valid [Fit83, Gor99].

As a special case, φ is **S4**-valid if and only if the finite set $\{\psi\}$, where ψ is the negation normal form of $\neg\varphi$, has a closed **S4**-tableau using only the rules of Figure 2. Indeed a closed tableau for a negation normal form can only use the rules of Figure 2. We shall restrict to negation normal forms in the rest of the paper.

The completeness proof gives a systematic method for proof-search, which consists of repeatedly applying all the invertible rules $(\wedge)$, $(\vee)$ and $(\Box\mathbf{S4})$ until no more applications of these rules are possible. In the case of $(\Box\mathbf{S4})$, this has to be made more precise: given $\Box\varphi$ in the current node, we add φ to it, and mark $\Box\varphi$ so as to prevent any reapplication of $(\Box\mathbf{S4})$ to the same formula $\Box\varphi$. When none of these rules is applicable, we have reached a node that corresponds to a so-called saturated world in the underlying Kripke model under construction; see [Gor99] for details. Some $\Diamond$-formula $\Diamond\varphi$ is then singled out for attention from this saturated node and a successor is created for it using the $(\Diamond\mathbf{S4})$-rule. Precisely, $\Diamond\varphi$ is replaced by φ, all $\Box$-formulae are unmarked (so as to reenable the application of $(\Box\mathbf{S4})$), and all other formulae are removed from the current node. The application of $(\Diamond\mathbf{S4})$ usually requires backtracking: if no proof is found by singling out $\Diamond\varphi$ from the current node, some other $\Diamond$-formula has to be chosen instead, until one finds one that leads to a proof, or until all $\Diamond$-formulae have been tried.

Naive proof search for a closed **S4**-tableau for some finite set of formulae Z using this systematic method can lead to infinite loops viz. $Z = \{\Box\Diamond p, p\}$:

$$\dfrac{\dfrac{\Box\Diamond p, p}{\Box\Diamond p, \Diamond p, p}\,(\Box\mathbf{S4})}{\begin{array}{c}\Box\Diamond p, p\\ \vdots\end{array}}\,(\Diamond\mathbf{S4})$$

Hence some form of *loop-checking* is necessary: if some node is obtained that has already been generated above in the same tableau, then proof search fails. In this example, this means that there is no closed tableau for Z.

Backtracking involves additional complications, in that loops do not always mean that there is no proof, rather that we have to make different choices to find a proof (if any). This can be seen from (3), where we can simulate the above loop by repeatedly applying rule $(\Diamond\mathbf{S4})$ on formula (a) (in particular, by using this rule rather than $(\vee)$ in the bottom deduction). This would loop; nonetheless (1) has a proof, namely (3).

Technically, it is also important that nodes are compared as sets, not just as lists; that is, duplicate formulae in nodes have to be removed. Otherwise, as the reader might want to check, the formula $\Box\Diamond\Box p \to \Diamond\Box\Diamond p$ leads to infinitely many nodes of the form $\Box\Diamond\Box p, \Box\Diamond\Box(\neg p), \Box p, \ldots, \Box p$, with an unbounded number of occurrences of $\Box p$.

It turns out that sets of nodes obtained higher up by the $(\Diamond\mathbf{S4})$ rule only need to be kept in the checklist. As we shall demonstrate, this will allow us to keep the space requirements for proof search to within polynomial bounds.

4. Algorithms

We now describe our algorithm and data structures in more detail.

4.1. Terms

Negation Normal Form: Input formulae are assumed to be in negated normal form (NNF). This requirement is not restrictive since every formulae can be converted into a logically equivalent NNF formula in linear time [BG98]. The advantage of using NNF is that the formulae in the parse tree of the NNF formula constitute all of the formulae that can appear in any node of the search tree.

Parse tree: A formula is parsed as a tree, where each node has at most two children. The nodes are characterized as `CONJ`, `DISJ`, `ALL`, `SOME` if they are of type $\varphi \wedge \psi, \varphi \vee \psi, \Box\varphi, \Diamond\varphi$ respectively. At the leaves there are literals: atomic formulae or their negations. Each node represents a subformula of the original NNF formula, with the root representing the whole NNF formula. With the tableau rules of Figure 2, it can be shown that the subformulae that appear in the parse tree are all the formulae that can appear in the nodes of the search tree. Clearly, the number of nodes in the parse tree is less than or equal to the length of the formula (it is exactly the length of the formula less the number of negation symbols). The number of formulae in any node of the search tree is therefore less than the length of the original NNF formula.

The parse tree is indexed, i.e., each node of the parse tree receives an integer number, so that each parent node has a smaller index than its children. This simplifies the visit sequence of the parse tree, as can be seen below.

Search Tree: In the sequel we refer to the **S4** tableau as the search tree.

$n_\Diamond, n_\Box, n_\vee, n_\wedge$: The number of subformulae of the appropriate type in the original NNF formula.

4.2. Storing nodes in the search tree

The parse tree provides access to the finite list of all formulae that can appear in the nodes of the search tree. Thus each node in the search tree can be represented as a bit string, whose bits indicate whether or not the corresponding formulae is present in the node. This bit string has length equal to the length of the original formula. Thus storing one node in the search tree requires n bits, where n is the length of the original formula.

In the case of formula (2), we may index each subformula by the following numbers

$$\underbrace{\underbrace{\Box\underbrace{\Diamond\underbrace{p}_{7}}_{4}}_{1} \wedge \underbrace{\Diamond\underbrace{\Box\underbrace{q}_{8}}_{5}}_{2} \wedge \underbrace{\Box\underbrace{(\underbrace{\neg q}_{9} \vee \underbrace{\neg p}_{10})}_{6}}_{3}}_{0} \tag{4}$$

The leftmost final node of the proof (3) is then the set $\{1,4,7,5,8,3,9\}$, represented as the bit string 01110111010 (bit 0 being the rightmost), while the two conclusions of rule $(\Diamond\mathbf{S4})$ used in the proof are $\{1,5,3\}$ (00000101010) and $\{1,7,5,3\}$ (00010101010).

4.3. Loop Checking

As shown in Section 3.2., an **S4**-tableau can contain an infinite branch. This problem can be solved by noticing that only a finite number of different nodes can appear in a search tree, and by avoiding examining any node twice. Thus if a node has ever been encountered before, it can be safely ignored. In this case, we backtrack to the last application of the $(\Diamond\mathbf{S4})$ rule, and choose a different $\Diamond$ formula there. If all $\Diamond$ formulae have been tried, we backtrack higher up to the previous application of the $(\Diamond\mathbf{S4})$-rule, and so on until all avenues have been explored, or a closed **S4**-tableau is found. This, however, is not a practical method, since it would require exponential space to store all the possible nodes of the search tree.

A better method is to check for repetitions of the nodes obtained by the application of the $(\Diamond\mathbf{S4})$ rule only, since these contain the "core" of the new worlds built by this rule. Thus the latter method looks for repetitions of the initial configuration of each newly generated world. This also guarantees the solution for the infinite branch problem, yet requires polynomial space. The price is that identical nodes on different branches now have to be treated separately.

4.4. A Straightforward Nondeterministic Algorithm

A naive implementation of **S4** tableaux would be by the following recursive procedure `prove`. We specify it concretely enough that we can use it as actual code in any functional programming language. The `prove` function takes six arguments $(pos, neg, \ell, boxes, dias, checkList)$, where pos and neg are sets of propositional variables (being variables p occurring as p, resp. $\neg p$ on the current node), ℓ is a list of formulae (the part of the current node containing formulae that

$$
\begin{array}{rcl}
\texttt{prove}(pos, neg, \bot :: \ell, boxes, dias, checkList) & = & true \\
\texttt{prove}(pos, neg, \top :: \ell, boxes, dias, checkList) & = & \texttt{prove}(pos, neg, \ell, boxes, dias, checkList) \\
\texttt{prove}(pos, neg, p :: \ell, boxes, dias, checkList) & = & p \in neg \vee \texttt{prove}(pos \cup \{p\}, neg, \ell, boxes, dias, checkList) \quad (p \in \texttt{PRP}) \\
\texttt{prove}(pos, neg, \neg p :: \ell, boxes, dias, checkList) & = & p \in pos \vee \texttt{prove}(pos, neg \cup \{p\}, \ell, boxes, dias, checkList) \quad (p \in \texttt{PRP}) \\
\texttt{prove}(pos, neg, (\varphi \wedge \psi) :: \ell, boxes, dias, checkList) & = & \texttt{prove}(pos, neg, \varphi :: \psi :: \ell, boxes, dias, checkList) \\
\texttt{prove}(pos, neg, (\varphi \vee \psi) :: \ell, boxes, dias, checkList) & = & \texttt{prove}(pos, neg, \varphi :: \ell, boxes, dias, checkList) \\
 & \wedge & \texttt{prove}(pos, neg, \psi :: \ell, boxes, dias, checkList) \\
\texttt{prove}(pos, neg, \Diamond\varphi :: \ell, boxes, dias, checkList) & = & \texttt{prove}(pos, neg, \ell, boxes, dias \cup \{\varphi\}, checkList) \\
\texttt{prove}(pos, neg, \Box\varphi :: \ell, boxes, dias, checkList) & = & \texttt{prove}(pos, neg, \varphi :: \ell, boxes \cup \{\varphi\}, dias, checkList) \\
\texttt{prove}(pos, neg, [], boxes, dias, checkList) & = & \exists \varphi \in dias \cdot (boxes, \varphi) \notin checkList \\
 & & \quad \wedge \texttt{prove}(\emptyset, \emptyset, \varphi :: boxes, boxes, \emptyset, checkList \cup \{(boxes, \varphi)\})
\end{array}
$$

Fig. 3. A straightforward proof search algorithm

we have to deal with), *boxes* is a set of formulae φ such that $\Box\varphi$ occurs in the current node, *dias* is a set of formulae φ such that $\Diamond\varphi$ occurs in the current node—so that the current node is exactly the set of formulae in ℓ, plus all atoms from *pos*, all negations of atoms from *neg*, all formulae of *boxes* with a $\Box$ added in front, and all formulae of *dias* with a $\Diamond$ added in front. Finally, *checkList* is a set of pairs $(boxes, \varphi)$ representing conclusions of the ($\Diamond$**S4**) rule that we have encountered higher up in the current tableau and which should not repeat (loop-checking).

Then our first algorithm may be written as in Figure 3, which may be implemented right away in languages like ML [Pau96]. The notation $\varphi :: \ell$ denotes the list ℓ with φ added in front. This is mainly used for pattern-matching purposes, and in that case it checks that the corresponding argument is a non-empty list, gets the first element in φ and the rest of the list in ℓ. We write [] the empty list, and [*boxes*] any list whose elements are those of the set *boxes*.

All the clauses in Figure 3 except the last one dispatch formulae in the to-do list ℓ depending on their topmost symbol. Eventually, `prove` will be called with $\ell = []$, leading to the last clause, which tries to apply rule ($\Diamond$**S4**) in a way leading to a proof. The condition $(boxes, \varphi) \notin checkList$ implements loop-checking, together with the fact that we add $(boxes, \varphi)$ to *checkList* in the last call to `prove`. (It would have been more natural, by the way, to call `prove` on arguments $(\emptyset, \emptyset, \varphi :: [\Box\psi | \psi \in boxes], \emptyset, \emptyset, \ldots)$; Figure 3 instead directly precompiles the obvious applications of rule ($\Box$**S4**).)

Finally, to prove φ, call $\texttt{prove}(\emptyset, \emptyset, Z, \emptyset, \emptyset, \emptyset)$, where Z is the one-element list containing the NNF of φ.

4.5. A Nondeterministic Algorithm

Figure 3 is quite detailed, and we shall rather use more informal notation in the sequel, so as to concentrate on the essentials. We shall also adopt a more imperative style of writing, writing *currentWorld* for the current node and assigning to it, and using the phrase "non-deterministically do" to replace existential quantifications (as in the last clause of the figure).

Since card memory is scarce, we also need to control how much space is used, and in particular it is dangerous to use a recursive style. Instead, we manage the recursion stack by hand. This saves space for many useless local variables and for return addresses: we don't need to memorize either kind of object. So our stack *stack* will only contain pairs of a checklist and a formula to backtrack to (these will always be left arguments to $\vee$). As an optimization, the sequential nature of the algorithm and the fact that check lists always grow as a tableau expands ensure that we need not store entire checklists: a single array *checkList* suffices, and the stack only memorizes which prefix of *checkList* is actually relevant: this prefix is coded as an integer *checkListSize* counting the number of initial objects in *checkList* that consistute the actual check list.

The following algorithm returns true if a given formula φ_0 of length n is **S4**-satisfiable, and false otherwise. It requires n^4 space.

stack ::= empty
checkList ::= empty
checkListSize ::= 0
currentWorld ::= $\{\varphi_0\}$
do
  while there are $\wedge$, $\vee$, $\Box$ rules that can be applied **do**
    apply these rules to *currentWorld*
    if the rule applied is an $\vee$ rule **then**
      push (*checkListSize*, left child of the rule)
          onto *stack*
      currentWorld ::= the right child of rule
    end if
  end while
  if *currentWorld* is closed **then**
    if the *stack* is empty **then**
      return false
    else
      (*checkListSize*, *currentWorld*) ::=
          pop from *stack*
      resize *checkList* to *checkListSize*
      continue
    end if

```
        end if
        if currentWorld appears in checkList[0..checkListSize-1]
            then return true
        end if
        if no (◇S4)-rules can be applied then
            return true
        end if
        non-deterministically pick a formula ◇φ
                from currentWorld
        apply (◇S4)-rule using ◇φ to currentWorld
                to get newWorld
        checkList[checkListSize] ::= currentWorld
        checkListSize::= checkListSize+1
        currentWorld ::= newWorld
while stack is not empty
```

4.5..1 Space complexity

It can be seen that *stack* grows by 1 only when the $(\vee)$ rule is applied. Thus for one world, we may need to store the maximum of $n_\vee$ possible configurations. Also the transitional rule $(\Diamond\mathbf{S4})$ which moves from one world to another preserves all the $\Box$-formulae, thus the set of all $\Box$-formulae (the core) in consecutive worlds in a branch of the search tree do not decrease. Therefore there are at most $n_\Box$ different cores in the same branch of the search tree. Also the worlds that have the same core will form a chain in the branch. We need to find the maximum number of worlds in a chain that have the same core. Since each new world is formed by taking all the $\Box$ formulae from the previous world together with one of the $\Diamond$ formulae, the maximum number of worlds in the chain that have the same core is $n_\Diamond$. Altogether, the maximum number of configurations that we need to store in the stack is $n_\vee \times n_\Box \times n_\Diamond \leq n^3$ (a better approximation is $n^3/3$). It can be seen that *checkList* contains only the node which is obtained by applying a $(\Diamond\mathbf{S4})$ rule. It is also resized so that all the nodes it contains are the initial configurations of the worlds in the current search branch. Thus *checkList* is always smaller than or equal to *stack*. Overall, the maximum number of configurations we might need to store in *stack* and *checkList* is of order n^3. Given that a configuration needs n bits, the space complexity of this algorithm is n^4 [Lad77a, Lad77b].

4.6. An improved algorithm

First, the non-deterministic choice of the final part of the algorithm must be eliminated: some form of enumeration of $\Diamond$ formulae has to be implemented. For each $\Diamond$ formula $\Diamond\varphi$ in *currentWorld*, we have to generate the initial configuration X_φ of a new world by just keeping φ and all $\Box$ formulae from *currentWorld*, and proceeding to build a closed sub-tableau for X_φ. If some such attempt succeeds, then *currentWorld* is refuted. To enumerate the formulae $\Diamond\varphi$, we store the world configuration *currentWorld* before we apply the $(\Diamond\mathbf{S4})$ rule and keep the index *lastK_index* of the last $(\Diamond\mathbf{S4})$ rule applied to that configuration.

Second, it is safe to always use the $(\vee)$ rule before any instance of $(\Diamond\mathbf{S4})$, as done in Section 4.5., but also to only apply $(\vee)$ once all rules $(\wedge)$ and $(\Box)$ have been applied. While this is not always optimal, it is generally a good heuristic: for example, the proof (3) never uses $(\vee)$ before $(\Diamond\mathbf{S4})$ except at the last step. Indeed, with respect to Figure 3, the $(\vee)$ rule involves a form of "universal" backtracking while $(\Diamond\mathbf{S4})$ involves a form of "existential" backtracking, and it is usually better to postpone backtracking rules as much as possible.

Third, while backtracking usually involves memorizing one branch of the computation on the stack while we explore the other branch, there is no need to memorize anything when backtracking is caused by instances of $(\vee)$, provided we know which subformulae in the current node are first or second arguments to an $\vee$ in the whole formula to prove, and provided we exploit certain properties of our indexing scheme for subformulae. Let us call a formula of type R if it occurs as a second argument to $\vee$, and of type L if it occurs as a first argument to $\vee$.

This is best explained on an example. Consider the node:

$$\underbrace{\Box\Diamond p}_{1},\underbrace{\Diamond p^{(a)}}_{4},\underbrace{\Box q}_{5},\underbrace{q}_{8},\underbrace{\Box(\neg q\vee\neg p)}_{3},\underbrace{\neg q\vee\neg p}_{6} \tag{5}$$

which we have already encountered in the proof (3). Note that $\neg q$ (subformula 9) is the only type L subformula, while $\neg p$ (subformula 10) is the only type R subformula, and we may go from one to the other by incrementing, resp. decrementing the index. This is because subformulae were actually indexed in a breadth-first manner.

To look for a closed tableau from this node, set *lastK_index* to 0, and generate the first of the conclusions of the $(\vee)$ rule:

$$\underbrace{\Box\Diamond p}_{1},\underbrace{\Diamond p^{(a)}}_{4},\underbrace{\Box q}_{5},\underbrace{q}_{8},\underbrace{\Box(\neg q\vee\neg p)}_{3},\underbrace{\neg q}_{9} \tag{6}$$

Note that we do *not* stack the other conclusion of the $(\vee)$ rule. Once we reach a closed node for this node, we know that we can obtain a corresponding node (not necessarily closed) of the subtableau below the other conclusion of the $(\vee)$ rule, by replacing the first type L subformula by the corresponding type R subformula (the unique other argument to the same $\vee$ operator), and all preceding type R subformulae by the corresponding type L subformula. If there is no type L subformula remaining, then all conclusions of the $(\vee)$ rule have been dealt with. Here the only type L subformula is 9, so we produce:

$$\underbrace{\Box\Diamond p}_{1},\underbrace{\Diamond p^{(a)}}_{4},\underbrace{\Box q}_{5},\underbrace{q}_{8},\underbrace{\Box(\neg q\vee\neg p)}_{3},\underbrace{\neg p}_{10} \tag{7}$$

Now no other rule than $(\Diamond\mathbf{S4})$ applies: *lastK_index* is incremented to 4, the position of formula (a). We get:

$$\underbrace{\Box\Diamond p}_{1},\underbrace{p}_{7},\underbrace{\Box q}_{5},\underbrace{\Box(\neg q\vee\neg p)}_{3} \tag{8}$$

which is the initial configuration of the new world. In particular, we memorize $\{1,7,5,3\}$ into *checkList*. We must also

reset *lastK_index* to 0, lest we lose required opportunities for applying ($\Diamond$**S4**) later on.
Apply all $\wedge$ and $\Box$**S4** rules, getting:

$$\underbrace{\Box\Diamond p}_{1},\underbrace{\Diamond p^{(a)}}_{4},\underbrace{p}_{7},\underbrace{\Box q}_{5},\underbrace{q}_{8},\underbrace{\Box(\neg q\vee\neg p)}_{3},\underbrace{\neg q\vee\neg p}_{6} \qquad (9)$$

Again apply $(\vee)$, leading to the closed node:

$$\underbrace{\Box\Diamond p}_{1},\underbrace{\Diamond p^{(a)}}_{4},\underbrace{p}_{7},\underbrace{\Box q}_{5},\underbrace{q}_{8},\underbrace{\Box(\neg q\vee\neg p)}_{3},\underbrace{\neg q}_{9} \qquad (10)$$

This is closed, so change the type L formula $\underbrace{\neg q}_{9}$ into the corresponding type R formula $\underbrace{\neg p}_{10}$. This is now closed again, and no type L formula remains. The proof is finished. We let the interested reader do the full example (2), noticing that loop-checking is now required.
Notice that our scheme for avoiding storing information for backtracking on $\vee$ subformulae requires us to index each subformula of the original formula with a distinct number, even though some subformulae may be equal. This indexing of the nodes of the parse tree forbids common sub-expressions and therefore introduces some redundancy. There may be duplications of the same subformula, but they are named with different indices, so they must be examined independently.
In the following algorithm, the stack stores only the configurations that have been fully expanded with non-($\Diamond$**S4**) rules. Also, "saturate *currentWorld* with non-($\Diamond$**S4**) rules" means applying $(\wedge)$, ($\Box$**S4**) and the left part of $(\vee)$:

$$\frac{X,\varphi_1\vee\varphi_2}{X,\varphi_1}$$

to *currentWorld* while this changes *currentWorld*. Recall that the right part of the $(\vee)$ rule will be obtained by looking at remaining type L formulae in the initial configuration of the current world, what we call "another sibling of *currentWorld*" below.

checkList ::= empty
stack ::= empty
currentWorld ::= world consisting of the original formula
do
  1. Saturate *currentWorld* with non-($\Diamond$**S4**) rules
  2. **while** *currentWorld* is closed **do**
    if there is another sibling of *currentWorld* **then**
      take *currentWorld* to be that sibling
      reset *lastK_index* to indicate no $\Diamond$**S4** rules
        have been applied yet
    else if the *stack* is empty **then**
      return false
    else
      (*lastK_index*,*currentWorld*) ::=
        pop from *stack*
      pop from *checkList*
    end if
  end while
  3. **do**
    lastK_index ::= the next $\Diamond$-formula
      from *currentWorld*
    if no more ($\Diamond$**S4**) rules can be applied **then**
      if *stack* is empty **then**
        return true
      else
        (*lastK_index*,*currentWorld*) ::=
          pop from *stack*
        pop from *checkList*
      end if
    end if
    apply the $\Diamond$**S4** rule to *currentWorld*
      to get *newWorld*
  while *newWorld* appears in *checkList*
  4. put (*lastK_index*,*currentWorld*) into *stack*
  put *newWorld* into *checkList*
  currentWorld ::= *newWorld*
while *stack* is not empty

Note that now *checkList* and *stack* are of the same size. The *checkList* is actually the core of the world configuration stored at the corresponding location in *stack*. Thus it can be obtained from *stack* by generating the core of each world in *stack*. This gives a more efficient use of space. It can be seen that for one node in the search tree, we need only one location in *stack*. Thus the space requirement is reduced by a factor of $n_\vee$: the maximum number of configurations stored in *stack* at one time is $n_\Box \times n_\Diamond \leq n^2$, and the space complexity of this algorithm is n^3 (a better approximation is $n_\Box \times n_\Diamond \leq n^2/2$, and the space requirement is $n^3/2$).

5. Data structures

5.1. The parse tree

The parse tree is stored in two byte arrays, one (*childs*) of length $2n$ and the other (*nature*) of length n. The $2i$ and $2i+1$ entries in the first array indicate the children of the i-th node in the parse tree (i.e. the indices of some other nodes in the parse tree, or the atom at that node), while the i-th entry in the second array indicates if the i-th node in the parse tree is a $\Box$, $\Diamond$, $\vee$, $\wedge$, $\neg$ or `PRP`. The $\vee$ and $\wedge$ nodes have two children. The $\Box$, $\Diamond$, $\neg$ and `PRP` nodes have one child, thus the second child for these node is redundant. This is not a severe problem, since the parse tree is fixed through the proving procedure. Note that in case of the $\neg$ and `PRP` nodes, the $2i$ entry of the *child* array contains the atom itself, not the index to another node in the parse tree.

5.2. The nodes in the search tree

As discussed above, each node in the search tree can be represented as a bit string of length n. To examine all the ($\Diamond$**S4**)-rules that can be applied to a node (i.e all the $\Diamond$ formulae in that node), we need to store the index to the last $\Diamond$ formula that has been examined, requiring one more byte.

5.3. The stack

Nodes are stored in a stack so that other branches in the search tree can be generated from the current branch, and so that a new node can be checked for duplication. This requires searching through all nodes in the stack, and also pushing and popping from the stack.

There are several possible implementations for the stack. The first two options store the stack as an array of bytes, and do not need extra memory for pointers. Since multidimensional arrays are not supported, access to elements of the stack will require some extra computations.

The upper bound of the size of the stack is known as seen above. Thus the stack can be allocated at the beginning of the procedure. We then need not worry about the growth of the stack. However, this is not a practical method, since the stack rarely grows to its theoretical upper bound size. Also, the limited amount of memory on the card will restrict the size of the input formulae. For example, with 512 bytes, the length of the formula will be less than 16, i.e. $16^3/2$ bits = 512 bytes. Allocating more than the card's RAM is possible, however it involves swapping to and from the EEPROM and will slow down the proof procedure. Consequently, the card reader usually cannot wait for the card and will throw an Exception.

Another way to implement the stack is to pre-allocate a small stack, and gradually increase its size by a large step when it becomes full. This ensures that the memory is used more effectively. However it still contains redundancy since it allocates more space than required each time it becomes full. Thus the longer formulae will result in larger redundancy. It also involves a lot of copying each time the stack grows.

The stack can also be implemented as a one way link list. Each node in the search tree is an element of the list. This requires extra memory for a pointer to the next element. However this extra memory becomes insignificant for long formulae. There is no redundancy. With this approach, the program has been tested for formulae of length up to 120.

6. Implementation

The program consists of two packages: `card` and `client`. The `client` package contains the classes for parsing the formula and converting it into NNF. Parsing is done by using Javacup (version 1.0j). We need a scanner (`scanner.java`) and a specification (`parser.cup`) for the formula, and Javacup automatically creates the parser. There is also a card proxy which manages the interactions with the card on behalf of the users. The class for testing is also in this package. Note that all of these operations are done off-board on a terminal (PC).

The `card` package contains an interface and two classes that are downloaded onto the card. These are classes `prover` and `State`, and the interface `proverInterface`. The interface `proverInterface` provides access to the services offered by the prover. These include loading the formula onto the card and proving. The interface also defines constants that are used by the prover, and are also used in the parsing and converting procedures. The class `prover` contains the codes for the proving procedure. The `prover` object that is loaded onto the card reserves enough space to hold the longest formula. When the formula is put onto the card, it is stored in the object. The user then must explicitly call the `prove` procedures. The `prove` method reads the formula from the object, performs simplifications and then starts looking for a model for the formula. (Note that there is a separation between loading the formula and proving. This is due to the fact that loading is rather complicated, and it is discussed in the next paragraph).

Despite the limitations of the card, the prover is able to work for a number of long formulae. Tests have been conducted for formulae of length up to 120. Passing the input to the card and storing input in the card then requires greater care because communication with the card is not simple. There is an upper-bound for the amount of data that can be transfered in one transmission (approaching 64K is not recommended). Long formulae therefore need to be broken into small pieces. Here, the input arrays are split into pieces of length 32 bytes and each piece is passed separately to the card, together with its length and position in the original array. Thus the maximum amount of data in one transmission is 34 bytes (one byte for the length and one for the position).

Since the inputs to the `prove` method are not ready in one pass, they need to be stored in the object `prover`, requiring the reservation of space for the longest input. Note that this also implies more time is required for copying the input from EEPROM to RAM in the `prove` procedure.

7. Results

This section shows the average time spent on the card in proving randomly generated formulae of various lengths (from 20 to 120). As can be seen, the time increases with the length of the formulae since longer formulae generally require more stack space, and require more arithmetic operations in the calculations.

8. Conclusions and Further Work

We have shown that even modal logic **S4** can be handled on a Java card. Thus transitive modal logics are not necessarily beyond the scope of Java cards. We now need to invent or explore appropriate logics of permissions and obligations to allow us to capture basic security notions like "trust". This is the subject of further work.

Another method for loop checking is to keep track of certain formula using a history mechanism [Heu99]. We intend to investigate whether such a history mechanism can be easily used in `CardS4`.

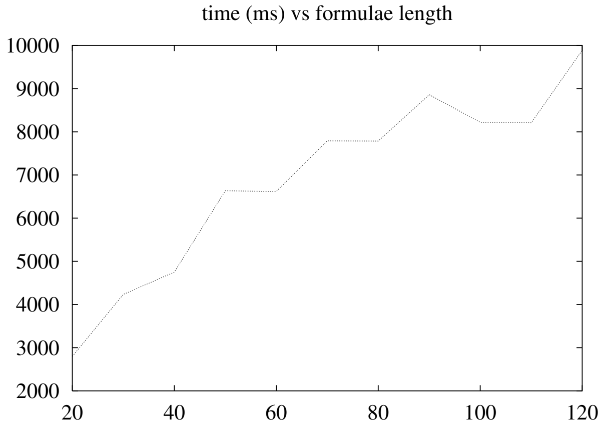

Fig. 4. Time (ms) vs formulae length

References

[Aba99] M. Abadi. Secrecy by Typing in Security Protocols. *Journal of the ACM* 46(5):749–786, 1999.

[ABLP93] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems* 15(4):706–734, 1993.

[AG97] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*. ACM Press, 1997.

[BAN90] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8:18–36, 1990.

[BElK01] P. Boury and N. El Kadhi. Static analysis of Java cryptographic applets. ECOOP'2001 Workshop on Formal Techniques for Java Programs. Technical report, FernUniversität Hagen, 2001. `http://www.informatik.fernuni-hagen.de/import/pi5/workshops/ecoop2001_papers.html`.

[BG98] N. Bonnette and R. Goré. A labelled sequent system for tense logic Kt. In *AI98: Proceedings of the Australian Joint Conference on Artificial Intelligence*, LNAI 1502:71-82. Springer, 1998.

[BL75] D. Bell and L. La Padula. Secure computer systems: unified exposition and Multics interpretation. Technical Report, MITRE Corp., Number MTR-2997, July 1975.

[Bru97] K. Brunnstein. Hostile ActiveX control demonstrated. *RISKS Forum*, 18(82), 1997.

[CDM01] A. Chander, D. Dean, and J. Mitchell. A state-transition model of trust management and access control. *14th Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, pages 27–43. IEEE Computer Society, 2001.

[CER00] CERT Coordination Center. CERT$^{(R)}$ Advisory CA-2000-15. Netscape Allows Java Applets to Read Protected Resources. `http://www.cert.org/advisories/CA-2000-15.html`.

[CS01] H. Comon and V. Shmatikov. Is it possible to decide whether a cryptographic protocol is secure or not? In this volume, 2001.

[ElK01] N. El Kadhi. Automatic Verification of Confidentiality Properties of Cryptographic Programs. *Networking and Information Systems*, 3(6), Hermès, 2001. `http://www.epita.fr:8000/~el-kad_n/Hermes.ps`.

[Fit83] M. Fitting. *Proof Methods for Modal and Intuitionistic Logics*, volume 169 of *Synthese Library*. D. Reidel, Dordrecht, Holland, 1983.

[Gir99] P. Girard. Which security policy for multiapplication smart cards. In *Proceedings USENIX Workshop on Smartcard Technology*, pages 21–28, Chicago, USA, 1999.

[Gol87] R. I. Goldblatt. *Logics of Time and Computation*. CSLI Lecture Notes Number 7, Center for the Study of Language and Information, Stanford, 1987.

[Gor99] R. Goré. Chapter 6: Tableau methods for modal and temporal logics. In M. D'Agostino, D. Gabbay, R. Hänle, J. Posegga, editor, *Handbook of Tableau Methods*, pages 297–396. Kluwer Academic Publishers, 1999.

[GJ01] A. Gordon and A. Jeffrey. Authenticity by typing for security protocols *14th Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, pages 145–159. IEEE Computer Society, 2001.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. *IEEE Symposium on Security and Privacy*, 1982.

[GNg00] R. Goré and L. D. Nguyen. CardKt: Automated Multi-modal Deduction on Javacards for Multi-application Security. In Proc. Java Card Workshop. Springer LNCS, to appear 2001.

[Heu99] A. Heuerding. *Automated Deduction in Some Propositional Modal Logics*. Institut für Angewandte Mathematik und Informatik. Universitäte Bern, Switzerland, 1999.

[HC96] G. E. Hughes and M. J. Cresswell. *A New Introduction To Modal Logic*. Routledge, 1996.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[Hud] J. Hudelmaier. Improved decision procedures for the modal logics K, T and S4.

[Lad77a] R. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal of Computing*, 6(3):467–480, September 1977.

[Lad77b] R. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal of Computing*, 6(3):467–480, 1977.

[Ler01] X. Leroy. Java bytecode verification: an overview. *Proceedings of 13th International Conference on Computer-Aided Verification*, pages 265–285. Springer-Verlag lecture notes in computer science 2102. G. Berry, H. Comon, A. Finkel eds., 2001.

[LR99] X. Leroy and F. Rouaix. Security properties of typed applets. Secure Internet Programming—security issues for mobile and distributed objects. Springer-Verlag lecture notes in computer science 1603, pages 147–182, 1999. `http://pauillac.inria.fr/~xleroy/publi/sip-typed-applets.ps.gz`.

[LY99] T. Lindholm and F. Yellin. *The Java virtual machine specification*. The Java series. Addison-Wesley, 1999. Second edition.

[Mas97] F. Massacci. Tableaux methods for access control in distributed systems. *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 246–260. Springer-Verlag lecture notes in artificial intelligence 1227, D. Galmiche, ed., 1997.

[Mat97] A. Mathuria. *Contributions to Authentication Logics and Analysis of Authentication Protocols*. PhD thesis, School of Information Technology and Computer Science, University of Woolongong, Woolongong, Australia, 1997.

[McGF99] G. McGraw and E. Felten. Securing Java—getting down to business with mobile code. John Wiley and sons, Inc. ISBN: 047131952X, 1999. `http://www.securingjava.com/`.

[Mon01] D. Monniaux. Analysis of Cryptographic Protocols using Logics of Belief: an Overview. In this volume, 2001.

[Nec97] G. Necula. Proof-carrying code. *Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[Pau96] L.C. Paulson. *ML for the Working Programmer (2nd ed)*. Cambridge University Press, 1996.

[PB00] P. Girard J.-L. Lanet V. Wiels G. Zanon P. Bieber, J. Cazin. Checking secure interactions of smart card applets. Technical report, Gemplus R&D Centre, 2000. `http://www.gemplus.com/smart/r_d/projects/pacap.htm`.

[RG93] A. Rao and M. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 318–324. Morgan-Kauffman, 1993.

[RR98] E. Rose and K. Rose. Lightweight bytecode verification. *OOPSLA Satellite Workshop on Formal Underpinnings of Java*, 1998.

[Shv90] Grigori F. Shvarts. Autoepistemic modal logics. In Rohit Parikh, editor, *Theoretical Aspects About Reasoning About Knowledge*, pages 97–109, 1990.

[Smi01] G. Smith. A new type system for secure information flow. *14th Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, pages 115–125. IEEE Computer Society, 2001.

[Sun00] Sun Microsystems. Java 2 platform micro edition technology for creating mobile devices. White paper, `http://java.sun.com/products/cldc/wp/KVMwp.pdf`.

Rajeev Prabhakar Goré
E-mail: `rpg@arp.anu.edu.au`
Web: `arp.anu.edu.au/~rpg`

Phuong Thê Nguyên
E-mail: `ntp@cse.unsw.edu.au`
Web: `www.cse.unsw.edu.au/~ntp`

Automated Reasoning Group
Computer Sciences Laboratory and Dept. of Computer Science
Institute of Advanced Studies and The Faculties
Australian National University.
Canberra, ACT 0200, Australia.