First-Order Cyberlogic Hereditary Harrop Logic

Vincent Bernat

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
bernat@csl.sri.com

Abstract. Cyberlogic [1] is an enabling foundation for building and analyzing protocols that involve the exchange of electronic forms of evidence. Protocols are distributed logic programs that gather evidence by using predicates and digital certificates. This document presents the underlying logic, based on first-order hereditary Harrop [2] formulas, and some interesting examples beyond trust management.

1 Introduction

With the advent of widespread network connectivity, many transactions that previously required the exchange of physical evidence can now take place electronically. The most obvious example is e-commerce which has benefited of much care with cryptographic research on protocols, notably SSL [3]. Other research areas include the correct use of cryptographic primitives to hold secrecy and authentication properties in protocols [4–13] or more complex properties like fairness for contract-signing protocols [14, 15]. However, the large class of transactions involving evidences in the form of licenses, registration documents, visas, certificates, etc. has been mostly ignored. To digitize such transactions, we need an agreed upon framework that can be used to exchange this kind of electronic evidences. Cyberlogic, introduced in [1], aims at describe evidential protocols so that they can be shown to achieve their intended purpose.

Main ideas in Cyberlogic are:

- keys are usually associated to agents where, in Cyberlogic, they are identified to specific authorities: one key is allowed to attest an evidence;
- evidences are encoded as digital certificates that one should gather from specific authorities to prove one's credentials; authorities are used to sign digital certificates, which in turn can be checked by any authority;
- protocols are distributed logic programs that gather evidences, encoded as digital certificates, to prove one goal.

For example, to get a loan, you need a document attesting your identity, a proof that you have a work, a proof that you will be able to refund the money. An ID card can be seen as a digital certificate that attests your identity. You may also substitute a driver's license. Your employer should issue a digital certificate that you work for him. You ask for the human resource department to give you such a certificate and a proof that your employer has delegated the issue of this kind of certificates to this department. At least, proof of property of a house or a car that you can get from an expert will hold as digital certificate for the ability to refund the money.

Related works include PolicyMaker [16] which uses "secure language" to describe policies. However, due to the use of such arbitrary language, PolicyMaker is not an unified framework and seems rather complex. A stripped down version, Keynote [17], can be used to express simple policies. PCA [18] has no execution engine: one should give the proof along with evidence, then the proof is checked by the authority. [19] presents a logic aimed at authorization and delegation not as general as Cyberlogic could be.

In this paper, we will first introduce logic used by Cyberlogic, based on first-order hereditary Harrop formulas, then give some useful insights on how things can be expressed with this logic. Endly, we will discuss on how to extend such a logic to embrace high-order hereditary Harrop formulas [2].

2 First-Order Cyberlogic Hereditary Harrop Logic

Underlying logic of Cyberlogic is very close of first-order hereditary Harrop logic described in [2] which is an intuitionistic logic. In fact, it is as close, that they are the same modulo some predicates: a first-order Cyberlogic hereditary Harrop formula *is* a first-order hereditary Harrop formula, like described in Nadathur's work, and vice-versa.

2.1 Bricks

Before describing directly the logic, we will introduce informally what "predicates" we would need for our logic. These bricks will be informally described, since we do not have yet defined the logic.

Attestations The most important brick in a trust management system is a way to say that an authority trusts an information to be correct. For example, your physician will attest that you are in a good shape and some authority will attest he is recognized as is.

We use the $:\triangleright$ symbol for this purpose. $K :\triangleright A$ would mean that we have a digital certificate signed by K and containing A. It mean that the authority[1], given at the left of the symbol, attests the veracity of the formula given at the right.

We will see that with the delegation mechanism, K could attest for the veracity of a formula A without directly signing it but trusting another authority about it. We would then write $K \triangleright A$.

[1] Remember, an authority is just a public-key

Delegation One other important mechanism in trust management is delegation. There is usually two ways for an authority to delegate some of its power: infinite delegation and bounded delegation. More complex delegation schemes could be used but we do not want them to be built in the logic.

Bounded delegation This kind of delegation is used when an authority K delegates to another authority K' authority on a formula A. However, K would like to restrict further delegation.

$\mathcal{D}_1(K, K', A)$ means that if K' gives a digital certificate signed by itself and attesting of A ($K' :\!\rhd A$), then K will attest of A ($K \rhd A$). K' is not able to delegate the formula A to a third authority K'': K is waiting for a digital certificate from K', not from K''. This *one-level* delegation mechanism is a good example of the intended meaning of $\rhd$ and $:\!\rhd$.

As in the real world, K' could receive a digital certificate of A signed by K'' and generate the same digital certificate signed by itself. K would then attests of A, through delegation. However, K' would endorse the full responsibility of A, while K will still be able to reject the responsibility of A to K'.

$\mathcal{D}_n(K, K', A)$ is the extension to an arbitrary depth n of the precedent delegation primitive. K' is allowed to delegate the formula A using $\mathcal{D}_k$, where $k < n$, delegation primitives. K' should be able to provide a chain of attestation of length n or less, ending by a digital certificate of A.

Infinite delegation $\mathcal{D}_\infty(K, K', A)$ is an extension of $\mathcal{D}_n(K, K', A)$ with $n \to \infty$: K' is allowed to delegate as depth as it wants the formula A. As long as K' attests, directly or not, of A, K will attest of it too.

Time Most certificates expire at some moment in the time; to become more realist, a model should handle time constraints. We should be able to handle three time constraints: proving that we carry an action after a given time, proving that we get a certificate before a given time and *check* that we have not reached a given time.

Consider the following scenarios:

After. You got a savings account with some high rate but the condition is to leave the money for five years. If you want to withdraw money from this account, you should provide, among other proofs, a proof that the end date has elapsed.

In time. You need to post a letter before time t. You deposit it to the post-office which timestamps the letter. The stamp is a proof that you have posted the letter in time.

Checking. We have two scenarios to illustrate this example:

1. Suppose you are now the post office and you must stamp a letter. Which time will you use for the time stamp?
2. You are a web server and get a certificate that expires at time t. How can you check that the certificate has not expired yet?

Only the *in time* scenario seems real. If you take the *after* scenario, you will give to the cashier all proofs, except the time proof. He will then look at his calendar and check that five years have really elapsed since the opening of the account. For the *checking* scenario, you just look at your clock or at your calendar and write down the hour or date you just read. What would happen if the cashier, in the first case, or you, in the second one, says the "wrong" date?

We assume there is one time authority K_T and a predicate time. If time t has elapsed, we have $K_T :\triangleright \diamond(t)$. We could imagine that such an authority broadcasts certificates in the air.

To handle time constraints, we introduce six new symbols: $\triangleright_{\leq t}$, $\triangleright_{=t}$, $\triangleright_{\geq t}$ and $:\triangleright_{\leq t}$, $:\triangleright_{=t}$, $:\triangleright_{\geq t}$. t should be replaced by any term representing a time.

$\triangleright_{\leq t}$ means that the authority attests the formula only before time t. Other symbols have similar meanings.

For example, a camera C will attest that one entity o was here between times t and t': $C :\triangleright_{\geq t}\mathrm{here}(o) \wedge C :\triangleright_{leqt'}\mathrm{here}(o)$. We can avoid the symbol $\triangleright_{\leq t}$ since we can express it by universally quantifying on $t' \leq t$ with the $\triangleright_{=t}$ symbol. The same applies for $\triangleright_{\geq t}$.

For the *after* scenario, we need to express the fact that there *exists* a time T posterior to another time t such that an expression is true at time T. We introduce the symbols $\triangleright_{\exists \geq t}$ and $:\triangleright_{\exists \geq t}$ for this usage. We use a special symbol since there are issues in using existential terms in programs.

Another important predicate to handle time is a way to get the current time. If t is the current time, then $\mathrm{curr}(t)$ holds.

2.2 Formal definition of the logic

The grammar The logic contains two different clauses (or formulas): program clauses (or cyberlogic formulas) and query clauses (or goals formulas). We define them with the grammar given in table 1, where:

- U_C is a set of arbitrary constants,
- U_V is a set of arbitrary variables,
- U_K is a set of arbitrary authorities,
- U_F is a set of arbitrary function symbols,
- U_P is a set of arbitrary predicate symbols

Note the restricted use of the delegation symbol. We try to stick to the logic of hereditary Harrop formulas whose grammar is:

$$G ::= A \mid G \wedge G \mid G \vee G \mid \exists x G \mid D \Rightarrow G \mid \forall x G$$
$$D ::= A \mid G \Rightarrow A \mid D \wedge D \mid \forall x D$$

Essential properties $\diamond$ symbols are distributable over $\vee$, $\wedge$, $\Rightarrow$, $\exists$, $\forall$ and is idempotent, as stated in table 2 on the facing page.

The reduction rules with $\neg$, $\exists$ and $\vee$ may arise attention. Suppose A is attesting that the blue pills brings back to reality while the red one does not

$$
\begin{aligned}
\textit{(constants)}\ C &::= \mathbb{N} \mid U_C \\
\textit{(variables)}\ V &::= U_V \\
\textit{(authorities)}\ K &::= K_T \mid U_K \\
\textit{(function symbols)}\ F &::= + \mid - \mid \times \mid U_F \\
\textit{(terms)}\ T &::= C \mid V \mid K \mid F(T) \mid T, T \\
\textit{(predicates)}\ P &::= \mathrm{curr} \mid \mathrm{time} \mid \leq \mid \geq \mid = \mid U_P \\
\textit{(atomic formulas)}\ A &::= P(T) \\
\textit{(attestation symbols)}\ \diamond &::= \ :\!\rhd \mid \rhd \mid :\!\rhd_{\exists \geq T} \mid :\!\rhd_{=T} \mid \rhd_{\leq T} \mid \rhd_{\exists \geq T} \mid \rhd_{=T} \\
\textit{(delegation symbols)}\ \mathcal{D} &::= \mathcal{D}_1 \mid \mathcal{D}_2 \mid \cdots \mid \mathcal{D}_\infty \\
\textit{(Cyberlogic goals)}\ G &::= G' \mid T \diamond G \\
G' &::= A \mid G \wedge G \mid G \vee G \mid \exists V.G \mid D \Rightarrow G \mid \forall V.G \\
\textit{(Cyberlogic formulas)}\ D &::= D' \mid T \diamond D \\
D' &::= A \mid G \Rightarrow A \mid G \Rightarrow (T \diamond A) \mid G \Rightarrow \mathcal{D}(T, T, A) \mid \\
&\quad\ D \wedge D \mid \forall V.D
\end{aligned}
$$

Table 1. Cyberlogic

$$
\begin{aligned}
k \diamond (A \wedge B) &\equiv (k \diamond A) \wedge (k \diamond B) \\
k \diamond (A \vee B) &\equiv (k \diamond A) \vee (k \diamond B) \\
k \diamond (A \Rightarrow B) &\equiv A \Rightarrow (k \diamond B) \\
k \diamond (\forall x A) &\equiv \forall x (k \diamond A) \\
k \diamond (\exists x A) &\equiv \exists x (k \diamond A) \\
k \diamond (k \diamond A) &\equiv k \diamond A
\end{aligned}
$$

Table 2. Reduction rules with respect to $\rhd$ and $:\!\rhd$

and B is the assertion that the red pills brings back to reality while the blue one does not. K is an authority that does not remember well the scenario of Matrix: K can only attests that $A \vee B$ (so $K :\rhd (A \vee B)$) but cannot attest of either A or B. However, we stick to our definition. If one want to include such a weak "or", one should introduce a new predicate (which will be used just on atomic formulas). Such a view is consistent with intuitionistic logic and constructive paradigms.

Direction attestations are linked to their indirect counterparts by the equations given in table 3 where A is any formula, k any authority and t is a term.

$$
\begin{aligned}
k :\rhd A &\Rightarrow k \rhd A \\
k :\rhd_{\exists \geq t} A &\Rightarrow k \rhd_{\exists \geq t} A \\
k :\rhd_{=t} A &\Rightarrow k \rhd_{=t} A
\end{aligned}
$$

Table 3. Direct attestation is attestation

Delegation are just convenient macros, as well as the predicate curr and the symbol $:\rhd_{\exists \geq t}$ and $\rhd_{\exists \geq t}$. These macros are stated in table 4.

$$
\begin{aligned}
\mathcal{D}_\infty(k, k', A) &\equiv k' \rhd A \Rightarrow k \rhd A \\
\mathcal{D}_1(k, k', A) &\equiv k' :\rhd A \Rightarrow k \rhd A \\
\mathcal{D}_2(k, k', A) &\equiv \forall k''(k'' :\rhd A) \wedge k' :\rhd ((k'' :\rhd A) \Rightarrow A) \Rightarrow k \rhd A \\
\mathrm{curr}(t) &\equiv K_T \rhd \mathrm{time}(t) \wedge \forall t'.(K_T \rhd \mathrm{time}(t')) \Rightarrow t' \leq t \\
k \rhd_{\exists \geq t} A &\equiv K_T \rhd \mathrm{time}(t) \wedge k \rhd A \\
k :\rhd_{\exists \geq t} A &\equiv K_T \rhd \mathrm{time}(t) \wedge k :\rhd A
\end{aligned}
$$

Table 4. Macros expansion

Proof system We should now introduce a proof system on top of Cyberlogic. Part of this proof system is already presented in tables 2, 3 and 4. We should complete it by some traditional intuitionistic rules from Brouwer-Heyting-Kolmogorov (BHK) interpretation. All inferences rules are grouped in table 5 on page 8. We should add trivial rules like elimination and duplication.

We should notice that a proof is paired with an actor: for example, for an actor K to get a proof of $A \wedge B$ from a proof of A and a proof of B, the actor K must *own* the proofs of A and B. We then have a specific inference rule to allow one actor to give a proof to another actor : this is the migration rule. The context Γ is conserved but the proofs that are not owned by the target actor are useless since no rule is able to manipulate them. We keep them to shorten the proof if a proof is migrated back. If the context were wiped out, the implication introduction rule should be used to pass proofs in Γ in the conclusion of the sequent.

We note a proof p_t^k when the type of the proof is t (this is purely informational and may be omitted) and the proof is owned by k.

Note the inference rule about equivalences of tables 2 and 4: such a simple inference rule allow us to say that if two formulas are equivalent, we can easily build a proof for one from the proof for the other (just use the equivalence rule at the end of the proof). Note also decomposition rules for $\vee$ and $\wedge$: the only proof for $A \vee B$ or $A \wedge B$ is a pair like the one obtained from the corresponding composition rule.

Normalization Normalization will allow to handle simpler formulas by expanding macros and reduce scope of attestation symbols. We will then be closer to first-order hereditary Harrop formulas described in [2] and for which a non-deterministic proof procedure exists.

The size of a Cyberlogic goal or of a Cyberlogic formula, where V is a variable, G and G' are goals, D and D' are formulas, T is a term, is defined in table 6 on page 9.

The first task is to get ride of macros stated in 4 on the preceding page by expanding them. The following definition and lemma will help us:

Definition 1. *A Cyberlogic formula or Cyberlogic goal are said* expanded *if they do not contain the following symbols: $\mathcal{D}_\infty$, $\mathcal{D}_1$, $\mathcal{D}_2$, curr, $\rhd_{\exists\geq t}$ and $:\!\rhd_{\exists\geq t}$.*

Lemma 1. *To each Cyberlogic formula, we can associate an equivalent* expanded *Cyberlogic formula with respect to equivalences rules stated in table 2 on page 5, as well as classic intuitionistic rules. The same holds for Cyberlogic goal.*

We prove this lemma by recursion on the size of the goal or of the formula. If the size is 1, then it could only be an atomic formula. Only the symbol curr can then occurs. Suppose the atomic formula is $\mathrm{curr}(t)$ with t some term. We rewrite it as $K_T \rhd \mathrm{time}(t) \wedge \forall t'.(K_T \rhd \mathrm{time}(t')) \Rightarrow t' \leq t$ which is an equivalent form. It is also a goal and a formula, so the lemma holds. For all other atomic formulas, the lemma holds too.

Suppose that the lemma holds for each goal and for each formula of size $n \geq 1$. Let G be a Cyberlogic goal of size $n+1$. G cannot be an atomic formula since its size is at least 2. If G is either a conjunction or a disjunction, we expand both members, whose size is at most n, and we get an equivalent goal meeting the conditions of the lemma. The same applies for implication and existential and universal quantifications. Now suppose that G is of the form $T \diamond G'$. If $\diamond$ is neither $\rhd_{\exists\geq t}$ or $:\!\rhd_{\exists\geq t}$, then we just build the expanded form of G' and we get our equivalent goal meeting the conditions of the lemma. If we have $G = T \rhd_{\exists\geq t} G'$, then we take G'' as the expanded form of G' and we rewrite the

$$\frac{}{\Gamma, p^k : A \vdash p^k : A}\ \textsc{Close} \qquad \frac{\Gamma \vdash p_1^k : A \qquad \Gamma \vdash p_2^k : B}{\Gamma \vdash \left(p_1^k, p_2^k\right)_{\mathrm{and}_i}^k : A \wedge B}\ \textsc{And}$$

$$\frac{\Gamma \vdash p^k : A}{\Gamma \vdash \left(p^k, 1\right)_{\mathrm{or}_{i_1}}^k : A \vee B}\ \textsc{Or} \qquad \frac{\Gamma \vdash p^k : B}{\Gamma \vdash \left(p^k, 2\right)_{\mathrm{or}_{i_2}}^k : A \vee B}\ \textsc{Or}$$

$$\frac{\Gamma, p'^k : A \vdash q^k : B}{\Gamma \vdash (P : p'^k \mapsto q^k)_{\mathrm{imp}_i}^k : A \Rightarrow B}\ \textsc{Imp} \qquad \frac{\Gamma, p'^k : A \vdash q^k : \bot}{\Gamma \vdash (P : p'^k \mapsto q^k)_{\mathrm{neg}_i}^k : \neg A}\ \textsc{Neg}$$

$$\frac{\Gamma \vdash p^k : A[x \leftarrow y]}{\Gamma \vdash (P : y \mapsto p^k)_{\mathrm{forall}_i}^k : \forall x A}\ \textsc{Forall} \qquad \frac{\Gamma \vdash p^k : A[x \leftarrow t]}{\Gamma \vdash (p^k, t)_{\mathrm{exists}_i}^k : \exists x.A}\ \textsc{Exists}$$

$$\frac{\Gamma \vdash p^k : \bot}{\Gamma \vdash p_{\mathrm{false}_i}^k : A}\ \textsc{False} \qquad \frac{\Gamma \vdash p^k : A \wedge B}{\Gamma \vdash (p, B)_{\mathrm{and}_{e1}}^k : A}\ \textsc{And} \qquad \frac{\Gamma \vdash p^k : A \wedge B}{\Gamma \vdash (p, A)_{\mathrm{and}_{e2}}^k : B}\ \textsc{And}$$

$$\frac{\Gamma, (p, A)_{\mathrm{and}_{e2}}^k : B, (p, B)_{\mathrm{and}_{e1}}^k : A, p^k : A \wedge B \vdash \Xi}{\Gamma, p^k : A \wedge B \vdash \Xi}\ \textsc{And}$$

$$\frac{\Gamma, p_2^k : A \Rightarrow B \vdash p_1^k : A \qquad \Gamma, p_2^k : A \Rightarrow B, (p_1^k, p_2^k, A)_{\mathrm{imp}_e}^k : B \vdash \Xi}{\Gamma, p_2^k : A \Rightarrow B \vdash \Xi}\ \textsc{Imp}$$

$$\frac{\Gamma, p_2^k : \neg A \vdash p_1^k : A \qquad \Gamma, p_2^k : \neg A, (p_1^k, p_2^k, A)_{\mathrm{neg}_e}^k : \bot \vdash \Xi}{\Gamma, p_2^k : \neg A \vdash \Xi}\ \textsc{Neg}$$

$$\frac{\Gamma, p^k : \forall x.A, (p, A, t)_{\mathrm{forall}_e}^k : A[x \leftarrow t] \vdash \Xi}{\Gamma, p^k : \forall x.A \vdash \Xi}\ \textsc{Forall}$$

$$\frac{\Gamma, p^k : A \wedge B, p_1'^k : A \vdash q_1^k : C \qquad \Gamma, p^k : A \wedge B, p_2'^k : B \vdash q_2^k : C}{\Gamma, p^k : A \wedge B \vdash (q_1^k, q_2^k, A, B, p^k)_{\mathrm{or}_e}^k : C}\ \textsc{Or}$$

$$\frac{\Gamma, p_1^k : \exists x.A, p'^k : A[x \leftarrow y] \vdash q^k : B}{\Gamma, p_1^k : \exists x.A \vdash (p_1^k, q^k, A)_{\mathrm{exists}_e}^k}\ \textsc{Exists} \qquad \frac{\Gamma \vdash p^k : A}{\Gamma \vdash p^k : A_{\equiv}}\ \textsc{Equiv}$$

$$\frac{\Gamma, p^k : A \vdash \Xi}{\Gamma, p^k : A_{\equiv} \vdash \Xi}\ \textsc{Equiv} \qquad \frac{\{p\}_k = A}{\vdash p^k : k :\rhd A}\ \textsc{Attest} \qquad \frac{\{p, t\}_k = A}{\vdash p^k : k :\rhd_{=t} A}\ \textsc{Attest}$$

$$\frac{\Gamma \vdash p^k : k :\rhd A}{\{p\}_k = A}\ \textsc{Attest} \qquad \frac{\Gamma \vdash p^k : k :\rhd_{=t} A}{\{p, t\}_k = A}\ \textsc{Attest} \qquad \frac{\Gamma \vdash p^k : k :\rhd A}{\Gamma \vdash p^k : k \rhd A}\ \textsc{Attest}$$

$$\frac{\Gamma \vdash p^k : k :\rhd_{=t} A}{\Gamma \vdash p^k : k \rhd_{=t} A}\ \textsc{Attest} \qquad \frac{\Gamma, p^k : k :\rhd A \vdash \Xi}{\Gamma, p^k : k \rhd A \vdash \Xi}\ \textsc{Attest} \qquad \frac{\Gamma, p^k : k :\rhd_{=t} A \vdash \Xi}{\Gamma, p^k : k \rhd_{=t} A \vdash \Xi}\ \textsc{Attest}$$

$$\frac{\Gamma \vdash p^{k'} : A}{\Gamma \vdash p^k : A}\ \textsc{Migrate}$$

Where $A' \equiv A$, as stated in tables 2 and 4; $A[x \leftarrow t]$ is a substitution of x by t in A; t is a term, x and y are variables, k and k' are actors. p', p_1', p_2' and y do not occur elsewhere.

Table 5. Inference rules for Cyberlogic

$\lvert T \diamond G\rvert = 1 + \lvert G\rvert$	$\lvert T \diamond D\rvert = 1 + \lvert D\rvert$	$\lvert A\rvert = 1$	$\lvert G \vee G'\rvert = \lvert G\rvert + \lvert G'\rvert$
$\lvert D \vee D'\rvert = \lvert D\rvert + \lvert D'\rvert$	$\lvert G \wedge G'\rvert = \lvert G\rvert + \lvert G'\rvert$	$\lvert \exists V.G\rvert = 1 + \lvert G\rvert$	
$\lvert \forall V.G\rvert = 1 + \lvert G\rvert$	$\lvert \forall V.D\rvert = 1 + \lvert D\rvert$	$\lvert D \Rightarrow G\rvert = \lvert D\rvert + \lvert G\rvert$	$\lvert G \Rightarrow A\rvert = 1 + \lvert G\rvert$
$\lvert G \Rightarrow (T \diamond A)\rvert = 1 + \lvert G\rvert$	$\lvert G \Rightarrow \mathcal{D}(T,T,A)\rvert = \lvert \mathcal{D}(T,T,A)\rvert + \lvert G\rvert$		

Table 6. Size of Cyberlogic goal or formula

goal as $K_T \rhd \text{time}(t) \wedge T \rhd G''$ which is an equivalent Cyberlogic goal meeting the conditions of the lemma. The transformation for $G = T :\rhd_{\exists \geq t} G'$ is similar.

Let D be a Cyberlogic formula of size $n+1$. D cannot be an atomic formula. If D is a disjunction or an universal quantification, we apply the same transformations as if it was a goal. Now assume that $D = G \Rightarrow A$. If curr does not occur in A, we just expand G into G' and rewrite D as $G' \Rightarrow A$ which is an expanded and equivalent form of D. Now, assume that $A = \text{curr}(t)$, then we expand G into G' and rewrite D as an expanded and equivalent form:

$$G' \Rightarrow (K_T \rhd \text{time}(t) \wedge \forall t'.(K_T \rhd \text{time}(t')) \Rightarrow t' \leq t)$$

We then use the distributivity of $\Rightarrow$ over $\wedge$ and $\forall$ to get:

$$G' \Rightarrow (K_T \rhd \text{time}(t)) \wedge \forall t'.G' \Rightarrow ((K_T \rhd \text{time}(t')) \Rightarrow t' \leq t)$$

It is then easy to prove this equivalence in intuitionistic logic[2], where A, B and C are formulas:

$$A \Rightarrow (B \Rightarrow C) \equiv (A \wedge B) \Rightarrow C$$

We can then rewrite D as:

$$G' \Rightarrow (K_T \rhd \text{time}(t)) \wedge \forall t'.\,(G' \wedge ((K_T \rhd \text{time}(t')))) \Rightarrow t' \leq t$$

We then have a Cyberlogic formula, in expanded form, equivalent to D. Now, assume that $D = G \Rightarrow (K \diamond A)$. We rewrite it as $K \diamond (G \Rightarrow A)$. $G \Rightarrow A$ is of size n and we use its expanded form to build an expanded equivalent to D. Assume that $D = G \Rightarrow \mathcal{D}(T, T', A)$ with G a goal, T and T' terms and A an atomic formula. We note G' the expanded equivalent of G. If $\mathcal{D}$ is $\mathcal{D}_\infty$, we rewrite D as:

$$G' \Rightarrow (T :\rhd (T' \rhd A) \Rightarrow T \rhd A)$$

Using the previous intuitionistic equivalence, and the equivalence rule for $\diamond$ and $\Rightarrow$, we rewrite it as:

$$T \rhd ((G' \wedge (T :\rhd (T' \rhd A))) \Rightarrow A)$$

[2] The proof is easily done by sequent calculus using notations in [2].

$(G' \wedge (k :\triangleright (k' \triangleright A))) \Rightarrow A$ is a Cyberlogic formula of size n. We grab its expanded and equivalent form and we get the equivalent form of D which meets the conditions of the lemma.

If $\mathcal{D}$ is $\mathcal{D}_1$, we rewrite D as:

$$T \triangleright ((G' \wedge (T :\triangleright (T' :\triangleright A))) \Rightarrow A)$$

If $\mathcal{D}$ is $\mathcal{D}_2$, we rewrite D as:

$$T \triangleright (G' \wedge (T :\triangleright (\forall T''(T'' :\triangleright A) \wedge T' :\triangleright ((T'' :\triangleright A) \Rightarrow A))) \Rightarrow A)$$

We conclude that for any Cyberlogic goal of formula of size $n \geq 1$, we can build an equivalent expanded goal or formula meeting the conditions of the lemma. $\square$

Definition 2. *A Cyberlogic formula or Cyberlogic goal are said* reduced *if each subexpression containing a $\diamond$ symbol at the top-level, can be expressed as B in the following grammar:*

$$B ::= T \diamond B \mid T \diamond A$$

Each Cyberlogic formula or goal is equivalent to a reduced form:

Lemma 2. *With respect to the equivalence rules stated in table 2 on page 5, as well as classic intuitionistic rules, each Cyberlogic formula or Cyberlogic goal can be associated with an equivalent reduced Cyberlogic formula or goal.*

We prove this lemma by recursion on the size of the Cyberlogic formula or Cyberlogic goal. The minimal size of both a goal or a formula is 1 and only an atomic formula can hold. Since an atomic formula cannot contain a $\diamond$ symbol, there is nothing to do and the lemma holds.

We assume now that each Cyberlogic goal whose size is less or equal to $n \geq 1$ can be turned into an equivalent Cyberlogic goal in respect with equivalences states in 2 where the formula or goal in the scope of the symbol at its right is an atomic formula. We assume the same for each Cyberlogic formula whose size is less or equal to $n \geq 1$.

Let G be a Cyberlogic goal whose size is $n + 1$. We discriminate over the form of G:

- G cannot be an atomic formula since its size is at least 2.
- If G is written as a conjunction of two goals, then each of them is of size of at most n and we apply the recursion hypothesis and the lemma holds.
- The same applies for the disjunction and the implication.
- The same applies for existential and universal quantification.
- G can be written as $T \diamond G'$ where T is a term and G' a Cyberlogic goal. $|T \diamond G'| = n + 1 = 1 + |G'|$ so $|G'| = n$ and, from the recursion hypothesis, all $\diamond$ symbols in G' are in the correct form. We will now recursively rewrite $T \diamond G'$ to equivalent forms until the newly introduced $\diamond$ symbol is in the correct form:

- If G' is of the form $T' \diamond G''$, we are in the terminal case of our construction. We use the recursion hypothesis that says $T' \diamond G''$ is a "B", so is $T \diamond G$ and the lemma holds.
- If G' is of the form $G_1 \wedge G_2$ or $G_1 \vee G_2$, then we use the appropriate equivalence in 2 and we get an equivalent formula $(T \diamond G_1) \wedge (T \diamond G_2)$ or $(T \diamond G_1) \vee (T \diamond G_2)$ and we roll up again the construction with the two parts of the conjunction or of the disjunction.
- If G' is of the form $\exists V.G''$ or $\forall V.G''$, we use the appropriate equivalence to get an equivalent formula $\exists V.(T \diamond G'')$ or $\forall V.(T \diamond G'')$. We roll up again the construction with $T \diamond G''$.
- If G' is of the form $D'' \Rightarrow G''$ we rewrite G to the equivalent form $D'' \Rightarrow (T \diamond G'')$ and we apply the construction to $T \diamond G''$, all $\diamond$ symbol in D'' are, by the recursion hypothesis, already in the good shape.

Obviously, the construction terminates and produces an equivalent form satisfying the condition of the lemma.

Let D be a Cyberlogic formula whose size is $n+1$. We discriminate over the form of D:

- D cannot be an atomic formula.
- If D is an implication (either $G \Rightarrow A$, $G \Rightarrow (T \diamond A)$ or $G \Rightarrow (T, T', A)$), both members of the implication are of size at most n and we apply the recursion hypothesis.
- The same applies for if D is a conjunction or an universal quantification.
- D can be written as $T \diamond D'$ and we use the same reasoning as if D was a goal[3].

In all cases, the goal or the formula is in the correct form. □

3 Examples of use

3.1 Bank's example

Instead of sticking with very small examples, we will develop here a full-blown example which will trigger many design issues with Cyberlogic. The most important authorities are:

HQ The headquarter of the bank should delegate most of its power but can fix general policy.

Office The bank has an office in each area of the world. An office can have corporate clients or individual clients. It can fix its own policy (in respect to the policy of the headquarter). ATM are managed by offices.

Sub-office Local clients may be managed by a sub-office which can fix its own policy with respect to the office from which it depends. ATM may also be managed by such small offices.

[3] the delegation symbol has to be expanded.

ATM ATM are special authorities allowing to withdraw money.

Client Clients may be single individuals or married individuals as well as corporate customers. Clients may have one or more accounts, may or may not delegate the withdraw order and may get one or several keys to access a given account.

We now define the most important predicates:

cash A given account a has x credits at time t.
owner A given account a is owned by a client K.
client A given client K is client of office O.
withdraw A given account a is withdrawn of amount x at time t.

To these predicates, we add "typing" predicates whose goal is to type a given term. For example office(O) is true if O is an office.

We could extend our scenario with more predicates but these one will already issue some design issues:

- How should we check that for a given time t and account a, only one cashing predicate holds ?
- How should we enforce policy of headquarter ? We could for example ask each withdrawal being approved by headquarters or just write policy of offices with respect of the policy of headquarter. Both approaches may find their ways in the real world.
- Should we check consistency at each level ? For example, if one office say that after one withdrawal, the account a has x credits, does the headquarter double check this affirmation ?

Most of design questions arise from the confidence we can put in each entity: if the headquarter writes a policy without knowing the policy of each office, it should double-check everything and request for approval of anything. However, if policies are elaborated in collaboration with the headquarters and all offices and may not change, the headquarters are aware of all the checks done by an office and can avoid to issue the same checks.

In designing the bank example, we follow a mix of those two approaches: the headquarters are confident about the correctness of operations done by the offices but ask them to request an approval for each transaction. To define a complete policy, we should take each authority and wonder what has this entity to say about each predicate.

Headquarter First, headquarter needs to define what the primary offices are:

$$
\begin{aligned}
HQ &:\triangleright \text{office}(O_1)\\
HQ &:\triangleright \text{office}(O_2)\\
&\dots\\
HQ &:\triangleright \text{office}(O_n)
\end{aligned}
$$

Then, we allow these offices to indicate suboffices:

$$\forall O, \forall O', (HQ :\rhd \text{office}(O)) \Rightarrow \mathcal{D}_1(HQ, O, \text{office}(O'))$$

Then, the headquarter delegates most of the other predicates, without perform any test, as stated above:

$$\begin{aligned}
\forall O, \forall A, HQ \rhd \text{office}(O) &\Rightarrow \mathcal{D}_1(HQ, O, \text{atm}(A)) \\
\forall O, \forall C, HQ \rhd \text{office}(O) &\Rightarrow \mathcal{D}_\infty(HQ, O, \text{client}(C, O)) \\
\forall O, \forall a, \forall x, \forall t, HQ \rhd \text{office}(O) &\Rightarrow \mathcal{D}_\infty(HQ, O, \text{cash}(a, x, t)) \\
\forall O, \forall a, \forall C, HQ \rhd \text{office}(O) &\Rightarrow \mathcal{D}_\infty(HQ, O, \text{owner}(a, C)) \\
\forall O, \forall a, HQ \rhd \text{office}(O) &\Rightarrow \mathcal{D}_\infty(HQ, O, \text{saving}(a)) \\
\forall O, \forall a, HQ \rhd \text{office}(O) &\Rightarrow \mathcal{D}_\infty(HQ, O, \text{checking}(a))
\end{aligned}$$

Predicates atm, saving and checking indicate respectively an ATM machine, a saving account and a checking account. Our bank policy would like to state that the maximum overdraft amount is 5000. We will request, physically, that each withdraw should be approved by the headquarter and we state:

$$\begin{aligned}
\forall O, \forall a, \forall x, \forall t, (HQ \rhd \text{office}(O) \ &\wedge O \rhd \text{withdraw}(a, x, t) \\
&\wedge O \rhd \text{cash}(a, y, t) \\
&\wedge \ y - x \geq -5000) \\
&\quad \Rightarrow HQ :\rhd \text{withdraw}(a, x, t)
\end{aligned}$$

Office In this section we will define the program of two offices. Other offices may copy it or have their own one. When we write O, it should be quantified over O_1, O_2 set.

First, O_1 will have several suboffices:

$$\begin{aligned}
O_1 &:\rhd \text{office}(O_{1,1}) \\
O_1 &:\rhd \text{office}(O_{1,2}) \\
&\dots \\
O_1 &:\rhd \text{office}(O_{1,n})
\end{aligned}$$

We assume that O_2 has no suboffice. O_1 has no direct clients and delegate this aspect to its suboffices, but manage transactions[4], while O_2 manages several clients. Moreover O_1 has no ATM while O_2 manages some. It is useless for O_1 to delegate the management of ATM since the headquarter will not approve:

$$\begin{aligned}
\forall O', \forall C, O_1 \rhd \text{office}(O') &\Rightarrow \mathcal{D}_\infty(O_1, O', \text{client}(C, O')) \\
\forall O', \forall a, \forall C, O_1 \rhd \text{office}(O') &\Rightarrow \mathcal{D}_\infty(O_1, O', \text{owner}(a, C)) \\
\forall O, \forall a, O_1 \rhd \text{office}(O') &\Rightarrow \mathcal{D}_\infty(O_1, O', \text{saving}(a)) \\
\forall O, \forall a, O_1 \rhd \text{office}(O') &\Rightarrow \mathcal{D}_\infty(O_1, O', \text{checking}(a))
\end{aligned}$$

[4] Suboffices will then attests that C is a client and a is an account.

$$
\begin{aligned}
O_2 &:\triangleright \text{client}(C_{2,1}, O_2)\\
O_2 &:\triangleright \text{client}(C_{2,2}, O_2)\\
O_2 &:\triangleright \text{client}(C_{2,3}, O_2)\\
O_2 &:\triangleright \text{client}(C_{2,4}, O_2)\\
O_2 &:\triangleright \text{atm}(A_{2,1})\\
O_2 &:\triangleright \text{atm}(A_{2,2})\\
O_2 &:\triangleright \text{atm}(A_{2,3})
\end{aligned}
$$

Both offices will manage the withdrawals in the same way: check that a client has issued an order to withdraw the credits, check that he owns the account, check that the balance is correct, check that there is no concurrent withdrawals, etc. To see how things are nested, we make a special case for client $C_{2,1}$: the withdraw should be approved by client $C_{2,2}$ if the amount is above 100.

$$
\begin{aligned}
\forall a, C, x, t, C \neq C_{2,1} \wedge\ & HQ \triangleright \text{owner}(a, C)\\
\wedge\ & C \triangleright \text{withdraw}(a, x, t)\\
\wedge\ & (\forall t', \forall y, y \neq x \wedge C \triangleright \text{withdraw}(a, y, t') \Rightarrow t \neq t')\\
\wedge\ & (\forall y, O \triangleright \text{cash}(a, y, t) \wedge \mathcal{Z}_{a,C,x,t,y})\\
& \Rightarrow O :\triangleright \text{withdraw}(a, x, t)
\end{aligned}
$$

where

$$
\begin{aligned}
\mathcal{Z}_{a,C,x,t,y} \equiv y \geq x \vee [O \triangleright \text{checking}(a) \wedge \exists a'\ & O \triangleright \text{saving}(a')\\
\wedge\ & HQ \triangleright \text{owner}(a', C)\\
\wedge\ & \forall y' O \triangleright \text{cash}(a', y', t) \Rightarrow y' + y \geq x]
\end{aligned}
$$

$\mathcal{Z}_{a,C,x,t,y}$ is here to check that the balance of the account will stay positive or that there is enough cash in the savings account of the client. The special case of client $C_{2,1}$ is then:

$$
\begin{aligned}
\forall a, C, x, t, C = C_{2,1} \wedge\ & HQ \triangleright \text{owner}(a, C)\\
\wedge\ & C \triangleright \text{withdraw}(a, x, t)\\
\wedge\ & x \geq 1000 \Rightarrow C_{2,2} \triangleright \text{withdraw}(a, x, t)\\
\wedge\ & (\forall t', \forall y, y \neq x \wedge C \triangleright \text{withdraw}(a, y, t') \Rightarrow t \neq t')\\
\wedge\ & (\forall y, O \triangleright \text{cash}(a, y, t) \wedge \mathcal{Z}_{a,C,x,t,y})\\
& \Rightarrow O :\triangleright \text{withdraw}(a, x, t)
\end{aligned}
$$

Once a withdrawal is approved (by the headquarter, which asks the office too, see above), the balance of the account should be "updated". The idea is to say that during the timeframe between t and t", if no withdraw is approved, then the amount of cash available is the initial amount minus the withdrawn amount:

$$\begin{aligned}\forall a, \forall x, y, z, \forall t, t', t'', O \rhd \mathrm{cash}(a, z, t) \wedge HQ \rhd \mathrm{withdraw}(a, x, t) \\ \wedge (HQ \rhd \mathrm{withdraw}(a, y, t') \Rightarrow t' < t \vee t' > t'') \\ \Rightarrow O :\rhd \mathrm{cash}(a, z - x, t'')\end{aligned}$$

Note that on these formulas, several assumptions are made about unicity of cash predicate for a given time. Such an assumption should be checked as early as possible since it is heavily used when using the universal quantifier to select the only existing possibility. Moreover, for the same reason, we assume that such a predicate exists for each possible t. Again, we will have to check this.

Since O_2 manages ATM, we should allow delegation of withdrawal to them:

$$\forall A, \forall a, \forall x, \forall t, O_2 \rhd \mathrm{atm}(A) \wedge x < 1000 \Rightarrow \mathcal{D}_1(O_2, A, \mathrm{withdraw}(a, x, t))$$

The balance of the corresponding account is still managed by O_2 with the formula given earlier.

Suboffices $O_{1,1}, \ldots, O_{1,n}$ should be managed in a similar way than ATM. We skip this part of the specification.

ATM We assume that all ATM use the same program. Here A will design one of O_2's ATM. The ATM will only work with clients of O_2 (where O_2's cashiers were able to serve any client who have an account). We do not allow overdraft, even with enough provision on the saving account. $C_{2,1}$ is not allowed to use ATM[5]. At least, we do not allow more than 800 credits to be withdrawn. We could add some temporal limit too (no more than 800 per day):

$$\begin{aligned}\forall a, C, x, t, C \neq C_{2,1} &\wedge O_2 \rhd \mathrm{owner}(a, C) \\ &\wedge O_2 \rhd \mathrm{client}(C, O_2) \\ &\wedge C \rhd \mathrm{withdraw}(a, x, t) \\ &\wedge (\forall t', \forall y, y \neq x \wedge C \rhd \mathrm{withdraw}(a, y, t') \Rightarrow t \neq t') \\ &\wedge (\forall y, O \rhd \mathrm{cash}(a, y, t) \wedge y \geq x) \\ &\wedge x < 800 \\ &\quad \Rightarrow O :\rhd \mathrm{withdraw}(a, x, t)\end{aligned}$$

The "leaves" We should now set the program of clients, tell who has what account in which bank, what is the balance of each account at the initial time, etc.

[5] We could introduce a more general predicate stating who is authorized to use ATM and who is not.

$$
\begin{aligned}
O_2 &:\rhd \text{owner}(a_1, C_{1,1})\\
O_2 &:\rhd \text{owner}(a_2, C_{1,1})\\
O_2 &:\rhd \text{saving}(a_1)\\
O_2 &:\rhd \text{checking}(a_2)\\
O_2 &:\rhd \forall t, t \leq 10 \Rightarrow \text{cash}(a_1, 5000, t)\\
O_2 &:\rhd \forall t, t \leq 10 \Rightarrow \text{cash}(a_2, 1000, t)\\
C_{1,1} &:\rhd \text{withdraw}(a_2, 100, 10)\\
C_{1,2} &:\rhd \forall t, t \geq 300 \text{withdraw}(a_2, 1500, t)
\end{aligned}
$$

Note the last clause say that $C_{1,2}$ gives his agreement for a withdrawal of 1500 on one account of $C_{1,1}$ after some time. The withdrawal will be approved only if $C_{1,1}$ wants to withdraw this amount of money (no more, no less).

4 Implementation

We describe here a complete implementation of Cyberlogic in Teyjus [20] as well as an interactive theorem prover. Teyjus is a Lambda Prolog interpreter. The following implementation is inspired by the work of Amy Felty [21].

4.1 Implementation of Cyberlogic

This section outlines the most important points of the implementation. See the complete archive to get the exact files.

Logic description Let `t` be the kind of a term and `form` the kind of formulas :

```
kind    t       type. % Term
kind    form    type. % Not term
```

Let us define the classical logic :

```
type    and     form -> form -> form.
type    or      form -> form -> form.
type    imp     form -> form -> form.
type    neg     form -> form.
type    forall  (t -> form) -> form.
type    exists  (t -> form) -> form.
type    false   form.
```

We describe the part specific to Cyberlogic :

```
type    curr    t -> form.
type    time    t -> form.
type    !/>     t -> form -> form.
type    />      t -> form -> form.
type    !/>E    t -> t -> form -> form.
type    />E     t -> t -> form -> form.
type    !/>T    t -> t -> form -> form.
type    />T     t -> t -> form -> form.
type    d1      t -> t -> form -> form.
type    d2      t -> t -> form -> form.
type    d@      t -> t -> form -> form.
```

The definition of functions and predicates are up to the user. For the sake of simplicity, we say that a term of kind `form` is a Cyberlogic formula. It is then possible to describe objects that are not really in Cyberlogic.

The definitions above should be compared to the one in table 1 on page 5. We define the reduction rules in tables 2 on page 5, 3 on page 6 and 4 on page 6 as follows:

```
% Signature for reduction
type   >>  form -> form -> o.
infixl >>  139.
% Signature for equivalence
type  ~~  form -> form -> o.
infixl ~~ 139.
% End Signature

% Formula >> canonical form
(K !/> (A and B)) >> X :- !, (((K !/> A) and (K !/> B)) >> X).
(K !/> (A or B)) >> X :- !, (((K !/> A) or (K !/> B)) >> X).
(K !/> (A imp B)) >> X :- !, ((A imp (K !/> B)) >> X).
(K !/> (forall X\A)) >> X :- !, ((forall X\(K !/> A)) >> X).
(K !/> (exists X\A)) >> X :- !, ((exists X\(K !/> A)) >> X).
(K !/> (K !/> A)) >> X :- !, ((K !/> A) >> X).
% If A is already normalized, and none of the above form apply then
% K !/> A is already normalized too.
(K !/> A) >> (K !/> A) :- (A >> A), !.
% Else, we normalize A then K !/> AA
(K !/> A) >> X :- !, (A >> AA), ((K !/> AA) >> X).
```

`>>` is a reduction operator, while `~~` is the equivalence as stated in table 2 on page 5. The rules printed above should be extended to other Cyberlogic operators. We complete by :

```
% Otherwise
A and B >> AA and BB :- !,  (A >> AA), (B >> BB).
A or B >> AA or BB :- !,  (A >> AA), (B >> BB).
```

```
A imp B >> AA imp BB :- !,  (A >> AA), (B >> BB).
(forall X\A) >> (forall X\AA) :- !,  (A >> AA).
(exists X\A) >> (exists X\AA) :- !,  (A >> AA).
A >> A.

% Equivalence
A ~~ B :- (A >> C), (B >> C).
```

Some tools Because the owner of a formula should be obvious, we would like not having to attach each axiom[6] to a given actor explicitly. We then define a special operator to describe this:

```
type owns t -> form -> o.
infix owns 137.

K owns (A imp B) :- !, K owns B.
K owns (A and B) :- !, K owns A, K owns B.
K owns (A or B) :- !, K owns A, K owns B.
K owns (forall X\A) :- !, (pi X\(K owns A)).
K owns (exists X\A) :- !, (pi X\(K owns A)). % Tricky !
K owns (neg A) :- !, K owns A.
K owns K !/> A :- !.
K owns KK !/> A :- !, fail.
K owns K /> A :- !.
K owns KK /> A :- !, fail.
K owns (/>T T K A) :- !.
K owns (/>T T KK A) :- !, fail.
K owns (!/>T T K A) :- !.
K owns (!/>T T KK A) :- !, fail.

% Otherwise, K _does_ own the formula
K owns A.
```

We define a judgment as the coupling of a proof and a formula. A sequent is then called a goal where the context (or antecedent) is a list of judgments and the succedent is a judgment.

```
kind    judg    type.

type    #       nprf -> form -> judg.
type    -->     (list judg) -> judg -> goal.
```

[6] This will make more sense latter when we introduce the notion of hypothesis to avoid excessive size of the context part of each sequent in the proof.

Inference rules 5 on page 8 is then used to implement the inference rules that are then called inference rules[7]:

```
% Each tactic is applied by an actor (first arg)
type    close_tac           t -> goal -> goal -> o.
type    and_i_tac           t -> goal -> goal -> o.
type    or_i1_tac           t -> goal -> goal -> o.
type    or_i2_tac           t -> goal -> goal -> o.
type    imp_i_tac           t -> goal -> goal -> o.
type    neg_i_tac           t -> goal -> goal -> o.
type    forall_i_tac        t -> goal -> goal -> o.
type    exists_i_tac        t -> goal -> goal -> o.
type    false_i_tac         t -> goal -> goal -> o.
type    and_e1_tac          t -> goal -> goal -> o.
type    and_e2_tac          t -> goal -> goal -> o.
type    and_e_tac           t -> int -> goal -> goal -> o.
type    imp_e_tac           t -> int -> goal -> goal -> o.
type    imp_e_tac_r         t -> int -> goal -> goal -> o.
type    or_e_tac            t -> int -> goal -> goal -> o.
type    forall_e_tac        t -> int -> goal -> goal -> o.
type    exists_e_tac        t -> int -> goal -> goal -> o.
type    neg_e_tac           t -> int -> goal -> goal -> o.
% Cyberlogic l = left, r = right
type    equiv_r_tac         t -> goal -> goal -> o.
type    equiv_l_tac         t -> int -> goal -> goal -> o.
type    att1_r_tac          t -> goal -> goal -> o.
type    att1_l_tac          t -> int -> goal -> goal -> o.
type    att2_r_tac          t -> goal -> goal -> o.
type    att2_l_tac          t -> int -> goal -> goal -> o.
% Migration
type    steal_tac           t -> goal -> goal -> o.
type    ask_tac             t -> t -> nprf -> goal -> goal -> o.
type    hyp_tac             t -> name -> goal -> goal -> o.

close_tac K (Gamma --> P # A) tt :- member (P # A) Gamma.

and_i_tac K (Gamma --> (and_i K P1 P2) # (A and B))
           ((Gamma --> P1 # A) && (Gamma --> P2 # B)).

or_i1_tac K (Gamma --> (or_i1 K P) # (A or B))  (Gamma --> P # A).

or_i2_tac K (Gamma --> (or_i2 K P) # (A or B))  (Gamma --> P # B).

% P is here a function taking two cyberlogic proofs
```

[7] We follow the glossary of [21].

```
imp_i_tac K (Gamma --> (imp_i K P) # (A imp B))
           (all PA\ (((PA # A)::Gamma) --> (P PA) # B)).

neg_i_tac K (Gamma --> (neg_i K P) # (neg A))
           (all PA\ (((PA # A)::Gamma) --> (P PA) # false)).

forall_i_tac K (Gamma --> (forall_i K P) # (forall A))
              (all Y\ (Gamma --> (P Y) # (A Y))).

exists_i_tac K (Gamma --> (exists_i K T P) # (exists A)) (Gamma --> P # (A T)).

false_i_tac K (Gamma --> (false_i K P) # A)  (Gamma --> P # false).

and_e1_tac K (Gamma --> (and_e1 K B P) # A) (Gamma --> P # (A and B)).

and_e2_tac K (Gamma --> (and_e2 K A P) # B) (Gamma --> P # (A and B)).

and_e_tac K N (Gamma --> PC # C)
    ((((and_e1 K B P) # A)::((and_e2 K A P) # B)::Gamma) --> PC # C) :-
   nth_item N (P # (A and B)) Gamma.

imp_e_tac K N (Gamma --> PC # C)
              ((Gamma --> P1 # A) && ((((imp_e K A P1 P2) # B)::Gamma) --> PC # C)) :-
   nth_item N (P2 # (A imp B)) Gamma.

% Another way to eliminate implication, to use with migration
% Using lambda abstraction
imp_e_tac_r K N (Gamma --> (imp_e K A P1 P2) # B)
                (Gamma --> P2 # (A imp B)) :-
  nth_item N (P1 # A) Gamma.

neg_e_tac K N (Gamma --> PC # C)
              ((Gamma --> P1 # A) &&
               ((((neg_e K A P1 P2) # false)::Gamma) --> PC # C)) :-
   nth_item N (P2 # (neg A)) Gamma.

forall_e_tac K N (Gamma --> PC # C)
                 ((((forall_e K T A P) # (A T))::Gamma) --> PC # C) :-
   nth_item N (P # (forall A)) Gamma.

or_e_tac K N (Gamma --> (or_e K A B P Q1 Q2) # C)
             ((all P1\ (((P1 # A)::Gamma) --> (Q1 P1) # C)) &&
              (all P2\ (((P2 # B)::Gamma) --> (Q2 P2) # C))) :-
   nth_item N (P # (A or B)) Gamma.
```

```
exists_e_tac K N (Gamma --> (exists_e K A P1 Q) # B)
                 (all Y\ (all P\ (((P # (A Y))::Gamma) --> (Q Y P) # B))) :-
   nth_item N (P1 # (exists A)) Gamma.

equiv_r_tac K (Gamma --> (equiv_t K P) # A)
              (Gamma --> P # AA) :- A >> AA.

equiv_l_tac K N (Gamma --> PC # C)
                ((((equiv_t K P) # AA)::Gamma) --> PC # C) :-
   nth_item N (P # A) Gamma,
   A >> AA.

att1_r_tac K (Gamma --> (att_t1 K P) # (KK /> A))
             (Gamma --> P # (KK !/> A)).

att2_r_tac K (Gamma --> (att_t2 K P) # (/>T KK T A))
             (Gamma --> P # (!/>T KK T A)).

att1_l_tac K N (Gamma --> PC # C)
             ((((att_t1 K P) # (KK /> A))::Gamma) --> PC # C) :-
   nth_item N (P # (KK !/> A)) Gamma.

att2_l_tac K N (Gamma --> PC # C)
             ((((att_t2 K P) # (/>T KK T A))::Gamma) --> PC # C) :-
   nth_item N (P # (!/>T KK T A)) Gamma.

% Migration
% With stealing, we just stay in the same process.
steal_tac K (Gamma --> (migrate K P) # A)
            (Gamma --> P # A).

% Here, we ask for a proof P from KK (a metaprogram will give us such a proof)
% The proof stops here for this process
ask_tac K KK P (Gamma --> (migrate K P) # A)
               tt.

% Hypothesis introduction
% The hypothesis is reduced and we check that it is owned by K.
hyp_tac K Name (Gamma --> P # A)
               (((hyp_i K Name) # F)::Gamma --> P # A) :-
   formula Name FF,
   FF >> F,
   K owns F.
```

We should notice that we have some additional inference rules. The most important one is the hypothesis introduction. This rule is harmless and a convenient way of introducing an hypothesis only when we need it. The migration rule is turned into two versions : the first one is `steal_tac` which is the one described in 5 on page 8.

The second one, `ask_tac` needs an interaction with a higher-level program. The theorem prover that is described latter implements in fact one actor. The migration is then an operation to really ask another instance the proof of some formula. The higher-level program is then responsible for waiting for a proof, check it and then apply this inference rule.

Another thing to notice is that we don't enforce the fact that we need that proofs used as hypothesis of an inference rule to be owned by some specific actor: some limitations of Teyjus forced us to implement this at a higher level.

The interactive prover It is then really easy to write in Teyjus an interactive theorem prover that will map tactics and ask for inference rules to apply. The interactive prover is contained in `inter.mod`. It is possible to extend it to add the possibility to apply a batch of inference rules (or try to apply) instead of just one after one, as long as we don't use `ask_tac`. This is described in [21].

4.2 An interactive distributed theorem prover for Cyberlogic

There are several limitations of the interactive prover implemented in Teyjus:

- it is difficult to build a large proof, we need a graphical interface ;
- we do not exploit the distributivity of Cyberlogic;
- the implementation does not match the specification: for example, we do not check proofs are correctly assigned to actors to apply a sequent.

To circumvent such limitations, an interactive distributed theorem prover for Cyberlogic has been written in Python. The graphical interface is driven by the modern and widespread toolkit GTK2. Each actor has its own instance of the interpreter that will wait for query from the user or from other instances and try to solve them (with the help of the user). Several additional checks are done, including ownership. However, we still do not check that a given formula is really a Cyberlogic one.

Installation As long as Python, GTK 2 and pygtk2 are installed on the system, the installation is really straight-forward. The encryption part is assured by `ezPyCrypto` which is available at `http://www.freenet.org.nz/ezPyCrypto/`. The `INSTALL` file explains how to install this module.

Use To launch the interpreter, just type `python cyberlogic.py` in a terminal. If everything is fine, a window should popup. The interpreter reads the file `set1.mod`, which can be different for each actor if needed. This file should declare some formulas (that will be used as hypothesis or as goal). A formula is of type `name`:

```
type    needham         name.
formula needham (forall Na\ (forall Nb\ ((k2 /> (channel k1 k2 Na Nb))
                                 imp (k1 /> (channel k1 k2 Na Nb))))
       imp (k2 /> (init k1 k2 Na))).
```

The file should also define the actors as long as with their IP, public and private keys:

```
actor k "pub-k.key" "priv-k.key" "127.0.0.1:14000".
actor k1 "pub-k1.key" "priv-k1.key" "127.0.0.1:14001".
actor k2 "pub-k2.key" "priv-k2.key" "127.0.0.1:14002".
actor k3 "pub-k3.key" "priv-k3.key" "127.0.0.1:14003".
actor k4 "pub-k4.key" "None" "127.0.0.1:14004".
```

Notice that key are contained in a separate file. You can use `python genkeypair.py k5 127.0.0.1:4000` that will generate the keypair for actor `k5` which is listening at IP 127.0.0.1 and port 4000. The script will display the line that could be added in `set1.mod`.

Actors should be of type `t`:

```
type    k               t.
type    k1              t.
type    k2              t.
type    k3              t.
type    k4              t.
```

After starting the interpreter, you should get something like the figure 1 on the next page. You should then select the correct actor for this session. The server part will then start and the application will listen for incoming request. Since there is no other instance running, we should start a new session ourselves.

Notice that you should not enter anything in the zone at the right of the "Exec" button unless you know what you do: the content of the zone will be sent to the interpreter showed below.

To start a new proof, choose one formula in the drop-down box and hit the button "Solve formula". The figure 2 on page 25 shows the main interface when solving a proof. Let's start from the top:

– You can at any moment start any new proof by selecting a formula and hitting the button "Solve this formula". You can also write your own formula if the ones provided in `set1.mod` are not sufficient.
– The remaining interface is dependent of the formula you are about to solve. Notice the tab below. The main tab is useless now, it just shows the main Teyjus interpreter running, you should not worry about it. The server tab will be explained latter. All the other tabs correspond to a proof.
  - The upper part shows the current goal. A goal is a set of hypothesis (the context of our sequent) and a conclusion. The interface is here to help you solve the conclusion from the given hypothesis.

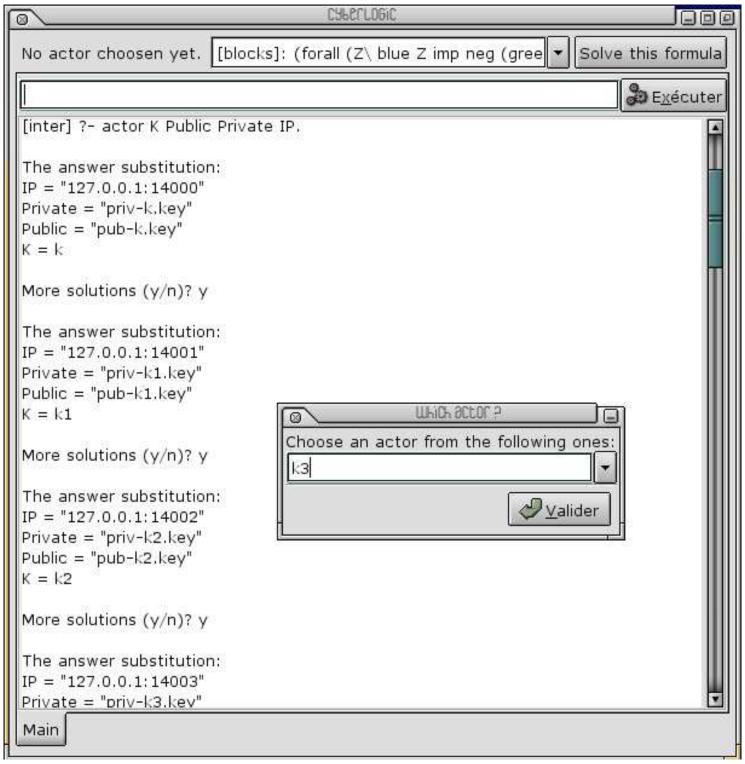

Fig. 1. Cyberlogic interpreter at start

- The lower part is the Teyjus interpreter attached to the current proof. You can interact with it by entering some clause in the zone at the left of the "Exec" button and hit the "Exec" button. However, it is safer to let the interface build the request for you.

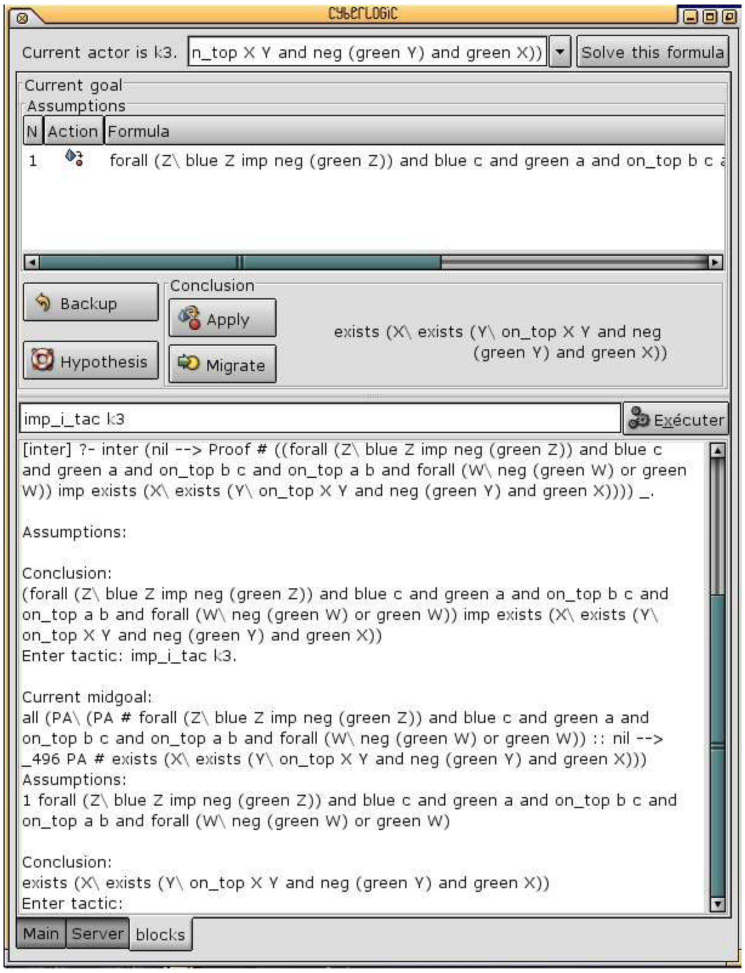

Fig. 2. Cyberlogic interpreter at work

From a given context, we can apply several kinds of inference rules:

– We can apply *introduction rules*. Hit the "Apply" button near the conclusion and a menu pops up to show the different inference rules that could be applied. Choose one. The inference rule is then applied if possible and we have a new context and a new conclusion to solve (or the proof is terminated).

– We can imply *elimination rules*. Right-click[8] on the corresponding hypothesis from the assumptions box and you should get a menu that will allow you to apply some elimination rules. As above, a new goal is proposed if the rule was successfully applied.
– We can *migrate* the rule, i.e give it to another actor. Just hit the "Migrate" button and the target actor will be asked. The application will contact this actor (it should be running at the corresponding IP) and wait for an answer from it. When the actor answers, the formula is then solved.
– We can *undo* an action by using the "Backup" button.
– We can *add an hypothesis* to the context using the "Hypothesis" button. A drop box will appear and you should enter the name of the hypothesis[9].

When a final proof is obtained, we get the possibility to add it to the server. The server tab summarizes all proofs gathered. If one actor asks for an already known proof, the server will automatically gives the proof without any user interaction. It is also a way to build a proof using lemmas: an actor can ask its own server.

When a proof is added to the server and that another actor requested it, the proof is automatically send to the corresponding actor.

How the migration work Each actor is listening to the given IP and port. It is possible to run every actor on the same computer by using the loopback interface 127.0.0.1 and a different port for each actor and it is as easy to use different computers connected by a TCP/IP network.

When a proof is requested, the requester contact the given actor and ask her for a proof. When the proof is available, it is packed with the dependences of the proof (since migration can have been used), encrypted and signed and sent back to the requester.

Strong cryptography is used but very little checking are done. It should be secured against a passive intruder or a well-manned intruder[10] but the application will just crash if the given bits of information are just random.

5 Future works

There are still several drawbacks in the current implementation:

– it is difficult to implement an automated or semi-automated version because of the explosion of states exploration that will result in explosion of the number of instances of the interpreter; the current implementation cannot handle more than several hundred instances.

[8] Or left-lick on the icon in the "Action" column.
[9] We can only introduce hypothesis that are predefined because Teyjus needs to compile them.
[10] For example, an intruder that respect the format of the protocol.

– communication between processes are done by “cut and paste” of what Teyjus outputs, however the output is not complete and lacks some information about the scope of the quantifications. Therefore, some complex proofs are incorrect.

The first point could be solved by using rewriting the application to use only one process instead of many. We would then have to do the branching ourselves. The second point is more problematic since it implies the modification of Teyjus, either to output the needed information or to implement the higher level interpreter right into Teyjus.

References

1. Ruess, H., Shankar, N.: Introducing cyberlogic. In Martin, B., ed.: HCSS'03 — High Confidence Software and Systems Conference, Baltimore, MD (2003)
2. Nadathur, G.: A proof procedure for the logic of hereditary harrop formulas. Technical Report Technical report DUKE–TR–1992–17 (1992)
3. Mitchell, J.C.: Finite-state analysis of security protocols. In: Computer Aided Verification. (1998) 71–76
4. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: 14th IEEE Computer Security Foundations Workshop (CSFW-14), Cape Breton, Nova Scotia, Canada, IEEE Computer Society (2001) 82–96
5. Song, D.X.: Athena: A new efficient automatic checker for security protocol analysis. In: PCSFW: Proceedings of The 12th Computer Security Foundations Workshop, IEEE Computer Society Press (1999)
6. Basin, D., Mödersheim, S., Viganò, L.: An on-the-fly model-checker for security protocol analysis. In: ESORICS 2003: Proceedings of the 8th European Symposium on Research in Computer Security. (2003)
7. Clark, J., Jacob, J.: A survey of authentication protocol literature: Version 1.0 (1997)
8. Schneider, S.: Verifying authentication protocols with CSP. In: PCSFW: Proceedings of The 10th Computer Security Foundations Workshop, IEEE Computer Society Press (1997)
9. Lowe, G.: A hierarchy of authentication specifications. In: PCSFW: Proceedings of The 10th Computer Security Foundations Workshop, IEEE Computer Society Press (1997)
10. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. In: Fourth ACM Conference on Computer and Communications Security, ACM Press (1997) 36–47
11. Lowe, G.: Casper: A compiler for the analysis of security protocols. In: PCSFW: Proceedings of The 10th Computer Security Foundations Workshop, IEEE Computer Society Press (1997)
12. Burrows, M., Abadi, M., Needham, R.: A logic of authentication, from proceedings of the royal society, volume 426, number 1871, 1989. In: William Stallings, Practical Cryptography for Data Internetworks, IEEE Computer Society Press, 1996. (1996)
13. Woo, T.Y.C., Lam, S.S.: A semantic model for authentication protocols. In: RSP: IEEE Computer Society Symposium on Research in Security and Privacy. (1993)
14. Shmatikov, V., Mitchell, J.C.: Analysis of abuse-free contract signing. Lecture Notes in Computer Science **1962** (2001)

15. Chadha, R., Kanovich, M., Scedrov, A.: Inductive methods and contract-signing protocols. In: Proceedings of the 8th ACM conference on Computer and Communications Security, ACM Press (2001) 176–185
16. Blaze, M., Feigenbaum, J., Strauss, M.: Compliance checking in the policymaker trust management system. In: Financial Cryptography. (1998) 254–274
17. Blaze, M., Feigenbaum, J., Keromytis, A.D.: KeyNote: Trust management for public-key infrastructures (position paper). Lecture Notes in Computer Science **1550** (1999) 59–63
18. Appel, A.W., Felten, E.W.: Proof-carrying authentication. In: ACM Conference on Computer and Communications Security. (1999) 52–62
19. Li, N., Grosof, B.N., Feigenbaum, J.: A practically implementable and tractable delegation logic. In: IEEE Symposium on Security and Privacy. (2000) 27–42
20. Nadathur, G.: Teyjus, a lambda prolog interpreter. (`http://teyjus.cs.umn.edu/`)
21. Felty, A., Miller, D.: Specifying theorem provers in a higher-order logic programming language. In: Ninth International Conference on Automated Deduction. Number 3310, Springer-Verlag LNCS (1988)