2010

Modeling and Analyzing Timed Security Protocols Using Extended Timed CSP

Zhang, Xian; Liu, Yang; Auguston, Mikhail

IEEE

Zhang, Xian, Yang Liu, and Mikhail Auguston. "Modeling and analyzing timed security protocols using extended timed csp." Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference on. IEEE, 2010.
http://hdl.handle.net/10945/59394

2010 Fourth International Conference on Secure Software Integration and Reliability Improvement

Modeling and Analyzing Timed Security Protocols Using Extended Timed CSP

Xian Zhang and Yang Liu
School of Computing,
National University of Singapore
{zhangxi5, liuyang}@comp.nus.edu.sg

Mikhail Auguston
Department of Computer Science
Naval Postgraduate School, USA
maugusto@nps.edu

***Abstract*— Security protocols are hard to design, even under the assumption of perfect cryptography. This is especially true when a protocol involves different timing aspects such as timestamps, timeout, delays and a set of timing constraints. In this paper, we propose a methodology for modeling and analyzing security protocols that are aware of timing aspects. We develop a formalism for modeling security protocols by extending Timed CSP with the capability of stating complicated timing behaviors for processes and events. A reasoning mechanism for the proposed formalism is developed based on Constraint Logic Programming (CLP). Using the reasoning engine built in CLP, the authentication properties of timed security protocols are able to be verified and attacks can be discovered. We demonstrate the capability of our method by modeling and verifying real-world security protocols. New approaches of using timing information to unfold and prevent potential attacks are also presented.**

I. Introduction

Security protocols are widely used for securing application-level data transport crossing distributed systems, typically by exchanging messages constructed using cryptographic operations (e.g. message encryption). In general, designing security protocols is notoriously difficult and error-prone. Many protocols proposed in the literature and many protocols exploited in practice turned out to be awed, or their well-functioning was found to be based on implicit assumptions. Since the late eighties various approaches [11], [16], [6], [9], [10], [7] have been put forward for the formal verification of security protocols to overcome the problems of faulty implementations and hidden requirements.

The new challenges are raise when different timing aspects are required in the security protocol design, such as timestamps, delays, timeout and a set of timing constraints. In the past years, there has been an increasing interest in the formal analysis of timed cryptographic protocols. However, there are few tool supports for modeling and analyzing security protocols with the capability of capturing various timing features. A particularly successful approach to analyze untimed security protocols is using CSP [18] to model and CSP model checker FDR [15] to analyze protocols [26], [23]. Motivated by this approach, we focus on timed extensions of CSP to accomplish the modeling and analyzing of timed security protocols, particularly Timed CSP.

Timed CSP [24] has been proposed as a formalism to model concurrent systems with timing behaviors. It is elegant and intuitive as well as precise such that it has been widely accepted and applied to a wide arrange of systems, including communication protocols, embedded systems, etc [25]. Our previous work [12] built a first tool for Timed CSP based on Constraint Logic Programming (CLP) [19]. However Timed CSP has limitations for specifying hard timing constraints such as *deadline*, execution time of a process or time-related constraints among events which are common requirements for timed security protocols.

Contribution In this work, we propose a formalism to model timed security protocols by substantially extending Timed CSP with the capability of stating complicated and critical timing constraints. In order to specify more timing behaviors of a process, we attach timed-related system requirements to each process in a modular manner. Our proposed language can specify expressions which cannot be modeled in Timed CSP, such as number of event occurrences, relations of occurrences time of events from different processes. Furthermore, it provides a generic way of specifying additional critical requirements as a first order logic predicate attached to each process. In this work, we formally define both syntax and operational semantics of the extended Timed CSP.

Our approach is different from the previous approaches by taking into account the time information. The use of explicit timing information allows us to specify security protocols with timestamps, timeout and retransmissions which can be naturally modeled using the specification. In the timing analysis, we could verify timed non-injective agreement authentication property which can be easily extended to other authentication property verification [16]. We also propose a novel approach of using the capability of the extended Timed CSP to avoid such attacks without changing the original specifications of the protocols. Besides, we can model timing requirements/constraints and verify other timed sensitive properties such as execution time of a protocol which is beyond the capability of existing approaches.

Our engineering effort realizes all the techniques in a verification engine for analyzing security protocols. The underlying reasoning mechanism is based CLP, which has been successfully applied to model programs and transition systems for the purpose of verification [17], [21]. In our previous work [12], we developed a reasoning mechanism based on CLP to verify Timed CSP. In this work, we extend the reasoning mechanism to support the extension of Timed CSP. A prototype is implemented based on one of the established

978-0-7695-4086-3/10 $26.00 © 2010 IEEE
DOI 10.1109/SSIRI.2010.29

CLP solvers, CLP($\mathcal{R}$) [20]. CLP($\mathcal{R}$) is chosen for its support of real numbers and continuous time variables. A number of theories, libraries and patterns are developed for easy querying and proving.

The rest of the paper is organized as follows. Section II introduces the syntax of the modeling language for timed security protocols. Section III presents the operational semantics of the proposed formalism. The encoding of the proposed semantics in CLP is described in Section IV. Section V introduces the reasoning methods for security protocols. Section VI explains how to conduct the analysis for timed security protocols. The last section discusses related works and concludes this paper.

II. Modeling of Timed Security Protocols

In this section, we present the formalism for timed security protocols. Firstly, we will briefly introduce the Timed CSP language. Secondly, the syntax of the newly proposed extension is presented. Thirdly, we will demonstrate how to model timed security protocols using the formalism proposed.

A. Timed CSP Language

Hoare's CSP [18] is an event-based notation primarily aimed at describing the sequencing of behavior within a process and the synchronization (or *communication*) between processes. Timed CSP [24] extends CSP by introducing a capability to quantify temporal aspects of sequencing and synchronization.

Definition 1: (Timed CSP) A Timed CSP process is defined by the following syntax,

$$\begin{aligned} P ::= {} & \textsc{Stop} \mid \textsc{Skip} \mid \textsc{Run} \mid e \xrightarrow{t} P \mid e : E \to P(e) \\ \mid {} & e@t \to P(t) \mid P_1 \,\Box\, P_2 \mid P_1 \sqcap P_2 \mid P_1 \;{}_X||_Y\; P_2 \\ \mid {} & P_1 \,|[X]|\, P_2 \mid P_1 \,|||\, P_2 \mid P_1;\; P_2 \mid P_1 \,\triangledown\, P_2 \\ \mid {} & P1 \overset{d}{\triangleright} P2 \mid \textsc{Wait}\, d \mid P_1 \,\triangledown\{d\}\, P_2 \mid \mu X \bullet P(X) \end{aligned}$$

STOP denotes a process that deadlocks and does nothing. A process that terminates is written as SKIP. $\textsc{Run}_\Sigma$ is a process always willing to engage any event in Σ where Σ is the universal set of events. A process which may participate in event e then act according to process description P is written as $e@t \to P(t)$. The (optional) timing parameter t records the time, relative to the start of the process, at which the event e occurs and allows the subsequent behavior P to depend on its value. The process $e \xrightarrow{t} P$ delays process P by t time units after engaging event e. The external choice operator, written as $P \,\Box\, Q$, allows a process of choice of behavior according to what events are requested by its environment. Internal choice $\sqcap$ represents variation in behavior determined by the internal state of the process. The parallel composition of processes P_1 and P_2, synchronized on common events of their alphabets X, Y (or a common set of events A) is written as $P_1 \;{}_X||_Y\; P_2$ (or $P_1 \,|[A]|\, P_2$). $P_1 \,|||\, P_2$ is the interleaving composition. The sequential composition of P_1 and P_2, written as $P_1;\; P_2$, acts as P_1 until P_1 terminates by communicating a distinguished event $\checkmark$ and then proceeds to act as P_2. The interrupt process $P_1 \,\triangledown\, P_2$ behaves as P_1 until the first occurrence of event in P_2, then the control passes to P_2. The timed interrupt process $P_1 \,\triangledown\{d\}\, P_2$ behaves similarly except P_1 is interrupted as soon as d time units have elapsed. A process which allows no communications for period d time units then terminates is written as $\textsc{Wait}\, d$. The timeout construct written as $P_1 \,\triangleright\{d\}\, P_2$ passes control to an exception handler P_2 if no event has occurred in the primary process P_1 by some deadline d. Recursion is used to give finite representation of non-terminating processes. The process expression $\mu X \bullet P(X)$ describes processes which repeatedly act as $P(X)$. The detailed illustration of each process type can be found in [27]. The semantics of a Timed CSP process is precisely defined either by identifying how the process may evolve through time or by engaging in events (i.e., the operational semantics defined in [28]) or by stating the set of observations, e.g., traces, failures and timed failures (i.e., the denotational semantics as defined in [8]).

B. Extension of Timed CSP

In the extended Timed CSP specification, a process is extended with an optional WHERE clause, which consists of a (first order) predicate over a predefined set of time variables. More precisely, we introduce time point variables: START, END, ENGAGE and TES as explained as follows.

Given a process P, the variable $P.\textsc{Start}$ ($P.\textsc{End}$) denotes the exact starting (ending) time of process P. More specifically, $P.\textsc{Start}$ captures the starting time of a process P when its first event is enabled or when the "WAIT d" process is enabled if P starts with a WAIT process. $P.\textsc{End}$ is the ending time of the process P. If the process is terminating, $P.\textsc{End}$ is the engage time of event $\checkmark$. If the process is non-terminating, then $P.\textsc{End} = \infty$. Naturally, condition $P.\textsc{End} \geq P.\textsc{Start}$ holds all the time. Using the two variables, a *deadline* property (a task must be accomplished within a certain time) is expresses as P WHERE $P.\textsc{End} - P.\textsc{Start} \leqslant d$ where d is a constant ($d \in \mathbb{R}^+$), if and only if this process is terminating.

In many scenarios, there are requirements on an event to occur at some expected time, for example attending a meeting at 10am or printing the job within 15 minutes after receiving this job. A time variable ENGAGE is attached to an event e to denote the exact time when e is engaged, in the form of $e.\textsc{Engage}$. In an evaluation, event e is likely to be engaged more than once, variable $e.\textsc{Engage}_i$ is introduced to denote the i_{th} occurrence of e in an evaluation. For instance, $P \mathrel{\widehat{=}} a \xrightarrow{2} P$, one possible evaluation is $(a.\textsc{Engage}_1 = 0, a.\textsc{Engage}_2 = 3, a.\textsc{Engage}_3 = 5, \ldots)$. For event e engaged more than once in a trace, $e.\textsc{Engage}$ is the set of all $e.\textsc{Engage}_i$, i.e., $e.\textsc{Engage} = \{e.\textsc{Engage}_i\}$. Constraint $e1.\textsc{Engage}_i - e2.\textsc{Engage}_i \leqslant t$ is exactly the same as $\forall i \bullet e1.\textsc{Engage}_i - e2.\textsc{Engage}_i \leqslant t$, which means that each time after $e2$ is engaged, $e1$ must be engaged within t time units. Constraint $e1.\textsc{Engage} - e2.\textsc{Engage} \leqslant t$ means $\forall i,j \bullet e1.\textsc{Engage}_i - e2.\textsc{Engage}_j \leqslant t$, which specifies the requirement that $e1$ is not allowed to be engaged after t time units with any occurrence of $e2$.

For simplicity, to specify the constraint that event e must be engaged before time t, expression $e.\text{ENGAGE} \leqslant t$ is used instead of $\forall i \bullet e.\text{ENGAGE}_i \leqslant t$, if this process is terminating. For any process P, engaged time of all events must be in the range of $P.\text{START}$ and $P.\text{END}$, i.e., $\forall e \bullet P.\text{START} \leqslant e.\text{ENGAGE} \leqslant P.\text{END}$.

In addition, the variable $P.\text{TES}$ is introduced to capture the evaluation of a process up to the current time, where TES stands for *T*imed *E*vent *S*et. TES is a set of *timed event*s. A *timed event* is a pair drawn from $e \times \mathbb{R}^+$ where $e \in \Sigma$[1], consisting of a time and an event engage time variable. TES is used to record the engage time of all events engaged so far, which can be viewed as a history of the execution. Traces of the specific evaluation are able to be retrieved from TES. We can also retrieve other information we are interested in from the set TES. The following example illustrates a constraint concerning the number of occurrences of events in P.

Example 1: The following cryptographic device provides services of encryption and decryption to protocol principals. Each encryption will be finished within T_e time units and each decryption process will be finished within T_d time unites.

$$
\begin{aligned}
&\mathit{Cryptograph} \mathrel{\widehat{=}} \\
&\quad (\mathit{start\_encrpt} \rightarrow \mathit{end\_encrypt} \rightarrow \mathit{Cryptograph}) \\
&\quad \Box\ (\mathit{start\_decrypt} \rightarrow \mathit{end\_decrypt} \rightarrow \mathit{Cryptograph}) \\
&\quad \text{WHERE } \mathit{end\_encrpyt}.\text{ENGAGE} - \\
&\qquad \mathit{start\_encrpyt}.\text{ENGAGE} \leqslant T_e \\
&\qquad \wedge\ \mathit{end\_decrypt}.\text{ENGAGE} - \\
&\qquad \mathit{start\_decrypt}.\text{ENGAGE} \leqslant T_d
\end{aligned}
$$

□

Example 2: In checking security protocol authentication property, we add two signals *run* and *commit* into two protocol principals. If we want to guarantee the non-injective agreement for a protocol, we need to make sure for each protocol run, there are less *commit* events than *run* events. We can append this property as constraint into protocol process. We will illustrate the details of non-injective agreement in Section VI.

$$
\begin{aligned}
&\mathit{Non\_inj\_Free} \mathrel{\widehat{=}} \mathit{Protocol} \\
&\qquad\qquad \text{WHERE TES} \downarrow \mathit{run} \geqslant \text{TES} \downarrow \mathit{commit}
\end{aligned}
$$

where $\text{TES} \downarrow \mathit{run}$ is number of occurrences of event *run* in TES. The WHERE predicate guarantees for each protocol run, there will be less *commit* than *run*. □

The syntax of the extended Timed CSP process is summarized in the following.

$$
\begin{aligned}
P(x_1, ..., x_n) &::= \mathit{ProcExp}\ [\text{WHERE}\ \ \mathit{WherePred}] \\
\mathit{WherePred}\ &::= \mathit{WherePred} \wedge \mathit{WherePred} \\
&\mid \mathit{WherePred} \vee \mathit{WherePred} \\
&\mid \mathit{WherePred} \Leftrightarrow \mathit{WherePred} \\
&\mid \mathit{WherePred} \Rightarrow \mathit{WherePred} \\
&\mid \neg\, \mathit{WherePre} \mid \mathit{true} \mid \mathit{false} \\
&\mid \mathit{WhereExpr} \sim \mathit{WhereExpr}, \\
&\qquad\qquad \sim\ \in \{<, \leqslant, =, >, \geq\}
\end{aligned}
$$

[1] Σ is the universal set of events.

$$
\begin{aligned}
\mathit{WhereExpr} ::= &\ [\mathit{Name}.]\text{START} \mid [\mathit{Name}.]\text{END} \\
&\mid [\mathit{Name}.]\text{TES} \\
&\mid \mathit{Name}.\text{ENGAGE} \mid \mathit{Name}.\text{ENGAGE}_i \\
&\mid \mathit{WhereExpr} \odot \mathit{WhereExpr}, \odot \in \{+, -\} \\
&\mid \mathit{WhereExpr} \odot \mathbb{R}, \odot \in \{*, /\} \\
&\mid (\mathit{WhereExp}) \mid \mathbb{R}
\end{aligned}
$$

where P is the process name, $x_1, ..., x_n$ is an optional list of process parameters and *ProcExp* is a Timed CSP process expression. Each process can be attached with an optional *WherePred* with keyword WHERE . Process P without *WherePred* is exactly the same as a Timed CSP process.

To capture different aspects of the execution, we define a set of build-in libraries which can be directly used in where predicates.

- $\text{TES} \downarrow e$: the number of occurrences of event e in the timed event set TES.
- *first*(TES): the first event appearing in TES.
- *last*(TES): the last event appearing in TES.
- *fresh*(e, T, TES): check whether event e is firstly engaged at time T in TES.

Common requirements for a system can be easily specified. For instance, the *deadline* of a process, order of events, and separation time between events are specified as follows:

1) Process P must be finished within time d:
$P.\text{END} - P.\text{START} \leqslant d$
2) Process P must be finished before time d:
$P.\text{END} \leqslant d$
3) Max separation time between two events e_1, e_2 is d:
$e_2.\text{ENGAGE} - e_1.\text{ENGAGE} \leqslant d$
4) e_2 must be engages before e_1:
$e_1.\text{ENGAGE} - e_2.\text{ENGAGE} \leqslant 0$

C. Formal Specification of Timed Security Protocols

In this section, we show how to model timed security protocols using the language proposed in a structured way. All the protocols we consider have a similar objective: in each protocol, an *initiator A* seeks to establish a session with a *responder B*, possibly with the help of a *sever S*, where A, B, and S are *principal*s.

Principals (including initiator, responder and server) and intruder are modeled as processes. The whole network would be the parallel execution of all processes. Event $\mathit{send}.S.R.M$ is introduced to denote the behavior sending the message M from sender S to receiver R. Event $\mathit{receive}.S.R.M$ denotes receiver R receives a message M from sender S.

We use the Wide Mouth Frog protocol (WMF) [5] as a running example to illustrate the ideas. The Wide Mouth Frog protocol is a computer network authentication protocol designed for insecure networks. The goal is to allow two principals A and B to exchange a secret key *Kab* via a trusted server S. The model is described as follows, where *Kas* and *Kbs* are shared keys of A and B with sever S respectively. *Ta* and *Ts* are timestamps generated and sent by A and S respectively.

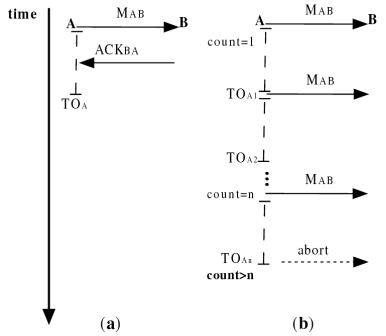

Fig. 1. Timeout patterns: (a) typical (b) count-bounded (c) time-bounded

A, B, S : *principal*
Kas, Kbs, Kab : *Key*
Ta, Ts : *timestamp*
1. $A \rightarrow S$: $A, \{Ta, B, Kab\}Kas$
2. $S \rightarrow B$: $\{Ts, A, Kab\}Kbs$

1) Timeout and Retransmissions: Look at the following simple protocol, where A sends a message M_{AB} to B, and later B will send an acknowledgement ACK_{BA} to A after receiving M_{AB}.

$1.A \rightarrow B : M_{AB}$
$2.B \rightarrow A : ACK_{BA}$

After principal A sends a message in a session of a protocol, A starts a timer that will *timeout* if A does not get ACK_{BA} from the receiver B. When a timeout is reached, the principal A can execute two actions: retransmit or reset a session. We are able to model both actions, where an implementation of the protocol should specify which action to perform. In some study with the introduction of timeout [7], principal A will resend the message if it detects a timeout. But they do not discuss the case if the resent message also gets timeout. In our specification, we introduce a bounded timeout, including count-bounded timeout and time-bounded timeout discussed as follows.

- *Count-bounded timeout* After principal A sends a message in a session of a protocol, it will start a timer and a counter. Once detecting a timeout, it will resend this message. If the resent message also gets timeout, it would resend again, by increasing number of resending by 1. If the number of times exceeds the max value, A will send request to abort this protocol (see Figure 1 (b)).
- *Time-bounded timeout* After principal A sends a message in a session of a protocol, it starts two timers. If it detects a timeout from the first timer, it will resend this message; if the resent message also gets timeout, A will resend again. Once it detects a timeout of the whole sending message process, A will send a request to abort this protocol (see Figure 1(c)).

Extended Timed CSP specification of both count bounded timeout and time bounded timeout are shown in process $CBTimeout(d, max)$ and $TBTimeout(d_1, d_2)$, respectively, where d and d_1 are the timeout for sending a message, max is the maximum retransmission times and d_2 is the timeout for the whole process. $c = \text{TES} \downarrow send.A.B.\{m_{AB}.A\}$ is the number of times of retransmission.

$$
\begin{aligned}
CBTimeout(d, max) \mathrel{\widehat{=}} & (\mu X \bullet send.A.B.\{m_{AB}.B\} \\
& \rightarrow (receive.B.A.\{ack_{BA}.B\} \rightarrow \text{SKIP} \overset{d}{\triangleright} X)) \\
& \triangledown ([c > max]\text{SKIP}) \\
& \text{WHERE } c = \text{TES} \downarrow send.A.B.\{m_{AB}.A\} \\
TBTimeout(d_1, d_2) \mathrel{\widehat{=}} & (\mu X \bullet send.A.B.\{m_{AB}.B\} \rightarrow \\
& (receive.B.A.\{ack_{BA}.B\} \rightarrow \text{SKIP} \triangledown \{d_1\}\ X)) \\
& \overset{d_2}{\triangleright} \text{SKIP}
\end{aligned}
$$

2) Timestamps and Lifetime: Timestamp is a typical way to prevent replay attacks, by simply attaching the current time value to a message. It is later used by the receiver of this message to make sure that it was recently generated, not a replay. A timestamp of a message can be easily recorded by $m.Engage$ which is the engage time of the event m, where event m denotes sending the message. In our implementation, we keep TES, a set of all timed events, which is a record of all engage time of all event engaged so far. It is easy to check whether $m.Engage$ is the most recent one or not by using the predefined predicate $fresh(m, m.Engage, TES)$ in Section II-B.

3) Initiator, Responder and Server: We model each principal (initiator, responder, server) as a process. For the Wide Mouth Frog protocol, the behavior of the initiator A is sending a message $\{T_a, b, k_{ab}\}$ to Server S using public key k_{as} and waiting for acknowledgement from S, where t_a is the timestamp. The responder B receives message from the server and then send the acknowledgment to S. The server receives message from A and then message $\{T_s, a, k_{ab}\}$ to B with new timestamp T_s. The three components are modeled as follows.

$$
\begin{aligned}
Initiator \mathrel{\widehat{=}} & start\_encrpt \rightarrow end\_encrpyt \rightarrow \\
& (\mu X \bullet send.A.S.\{T_a.b.k_{ab}\}_{k_{as}} \\
& (receive.S.A.\{ack_{SA}\} \rightarrow \text{SKIP} \overset{d_1}{\triangleright} X)) \\
& \triangledown \{d_2\}\ \text{SKIP} \quad \text{WHERE} \\
& T_a = send.A.S.\{T_a.b.k_{ab}\}_{k_{as}}.\text{ENGAGE}_1 \\
Responder \mathrel{\widehat{=}} & receive.S.B.\{T_s.a.k_{ab}\}_{k_{bs}} \rightarrow \\
& send.B.S.\{ack_{SB}\} \rightarrow start\_decrypt \rightarrow \\
& end\_decrypt \rightarrow \text{SKIP} \quad \text{WHERE} \\
& T_s \leqslant receive.S.B.\{T_s.a.k_{ab}\}_{k_{bs}}.\text{ENGAGE} \\
& \wedge fresh(receive.S.B.\{T_s.a.k_{ab}\}, T_s, \text{TES})^2
\end{aligned}
$$

[2]fresh(e,t,TES) is defined in Section II-B.

$$
\begin{aligned}
\mathit{Server} \;\widehat{=}\; & \mathit{receive.A.S.}\{T_a.b.k_{ab}\}_{k_{as}} \rightarrow \\
& \mathit{send.S.A.}\{\mathit{ack}_{SA}\} \rightarrow \mathit{start\_decrypt} \rightarrow \\
& \mathit{end\_decrypt} \rightarrow (\mu X \bullet \\
& \mathit{send.S.B.}\{T_s.a.k_{ab}\}_{k_{bs}} \rightarrow \\
& (\mathit{receive.B.S.}\{\mathit{ack}_{bs}\} \rightarrow \textsc{Skip} \overset{d_3}{\triangleright} X)) \\
& \triangledown\{d_4\}\ \textsc{Skip} \quad \textsc{Where} \\
& T_s = \mathit{send.S.B.}\{T_s.k_{ab}\}_{k_{bs}}.\textsc{Engage}_1 \\
& \wedge\, \mathit{fresh}(\mathit{receive.A.S.}\{T_a.b.k_{ab}\}, T_a, \textsc{Tes}) \\
\mathit{Network} \;\widehat{=}\; & \mathit{Initiator} \;||\; \mathit{Responder} \\
& ||\; \mathit{Server} \;||\; \mathit{Cryptograph}
\end{aligned}
$$

The network is a parallel composition of the three processes, as well as the *Cryptograph*.

4) Intruder: The intruder works basically as a Dolev-Yao intruder [11]. The difference is that our intruder takes time. The intruder can impersonate each agent executing the protocol, so it can play each of the roles in the protocol. Even thought the intruder has got its own keys, nonces etc., it can also try to use all the information it is receiving in the protocol run as its own (e.g., nonces). For the purpose of this paper, we restrict the behaviors of the intruder that it cannot read the mind of other principals to get some secret and it is unable to guess values. We restrict the behaviors of the intruder to the following actions:

- encrypt and decrypt a message
- intercept a message
- replay a message
- send a message to any principals
- delay a message with arbitrary time

The models of intruder and the new system with intruder are specified as follows:

$$
\begin{aligned}
\mathit{Intruder} \;\widehat{=}\; & \mathit{start\_encrypt} \rightarrow \mathit{end\_encrypt} \rightarrow \mathit{Intruder} \\
& \Box\ \mathit{start\_decrypt} \rightarrow \mathit{end\_decrypt} \rightarrow \mathit{Intruder} \\
& \Box\ \mathit{receive.S.I.M} \rightarrow \mathit{Intruder} \\
& \Box\ \mathit{send.I.R.M} \rightarrow \mathit{Intruder} \\
& \Box\ \mathit{receive.S.I.M} \xrightarrow{T} \mathit{send.I.R.M} \rightarrow \mathit{Intruder} \\
\mathit{System} \;\widehat{=}\; & \mathit{Network} \parallel \mathit{Intruder}
\end{aligned}
$$

where *I* is the identity of itself, *S* is the sender, *R* is the receiver and *M* is the message. The intruder would intercept into the protocol sessions through channel *send* and *receive*.

We present a natural way for specifying security protocols using extended Timed CSP, including the initiator, responder, server and intruder, with timestamps and timeout. It shows the extended Timed CSP is a good mechanism to model timed security protocols in a compositional way.

III. Operational Semantics

In the previous section, syntax of extended Timed CSP and approaches of specifying timed security protocols are properly illustrated. In this section, we formally define the semantics of the new specification. An operational semantics provides a way of interpreting a language by stepping through executions of programs written in that language. It describes an operational understanding of the language. The operational semantics of Timed CSP is precisely defined in Schneider [28] by using the combination of two relations: event transition and evolution. The semantics model consists of three components: the event and timed transitions which are inherited from Timed CSP, a WHERE predicate which must be satisfied by this model, and an *Timed stamped set*. A *Timed stamped set* (Tss) is a record of an execution, consisting of a set of process related time stamps, namely the starting and ending times of processes, and a *timed event set* (TES) which is a set of timed events. TES is a subset of *Tss*: $\textsc{Tes} \subseteq \mathit{Tss}$. A *timed event* is a pair drawn from $e \times \mathbb{R}^+$ where $e \in \Sigma$, consisting of a time and an event engage time value.

We define the state of a process as a quadruple $\langle P, t, W, \mathit{Tss}\rangle$ where P is the process, t is the current time, W is the WHERE predicate and *Tss* is the *timed stamped set* of the model. *Tss* keeps value for all variables. At each transition, an evaluation of the system requirement W is performed. If the current state satisfies the requirement, the transition can be enabled, otherwise not.

Definition 2: The operational semantics of the extended Timed CSP specification is a timed transition where the state is a quadruple $\langle P, t, W, \mathit{Tss}\rangle$, and event transitions and evolution transitions are defined by the rules:

- $\langle P, t, W, \mathit{Tss}\rangle \xrightarrow{a} \langle P', t, W, \mathit{Tss}'\rangle$ where $a \neq \checkmark \wedge \exists\, i : \mathbb{N} \bullet \mathit{Tss} \cup \{(a.\textsc{Engage}_i, t), (P.\textsc{Start}, t)\} \vDash W$
- $\langle P, t, W, \mathit{Tss}\rangle \xrightarrow{\checkmark} \langle P', t, W, \mathit{Tss}'\rangle$ where $\mathit{Tss} \cup \{(P.\textsc{End}, t)\} \vDash W$
- $\langle P, t, W, \mathit{Tss}\rangle \overset{d}{\rightsquigarrow} \langle P', t+d, W, \mathit{Tss}'\rangle$ where $d > 0 \wedge \mathit{Tss} \cup \{(P.\textsc{Start}, t)\} \vDash W$

where P' is the subsequent process of P by involving either a event transition ($\rightarrow$) or a timed transition ($\rightsquigarrow$). *Tss*$'$ is a timed stamped set with updated ENGAGE, $\textsc{Engage}_i$, START, END and TES variables. The $\xrightarrow{a}$ represents an event transition, whereas $\overset{d}{\rightsquigarrow}$ is a timed transition. $\exists\, i : \mathbb{N} \bullet \mathit{Tss} \cup \{(a.\textsc{Engage}_i, t), (P.\textsc{Start}, t)\} \vDash W$ is an evaluation of the current state, which is used to check whether the current *Tss* fulfills the system requirements W or not. □

In the operational semantics, we define both event transition and timed transition relations for all primary and compositional operators in Timed CSP specification. The operational semantics of the alphabetized parallel composition operator $P_1\ {}_X||_Y\ P_2$ are illustrated in the following rules.

$$
\frac{\begin{array}{l}\mathit{Tss} \cup \{(\textsc{Start}, t), (e.\textsc{Engage}_i, t)\} \vDash W \\ P_1 \xrightarrow{e} P_1'\end{array}}{\begin{array}{l}[e \in X \cup \{\tau\} \setminus Y, \mathit{Tes} \downarrow e = i-1] \\ \langle P_1\ {}_X||_Y\ P_2, t, W, \mathit{Tss}\rangle \xrightarrow{e} \\ \langle P_1'\ {}_X||_Y\ P_2, t, W, \mathit{Tss} \cup \\ \{(\textsc{Start}, t), (e.\textsc{Engage}_i, t)\}\rangle\end{array}} \quad [\,r1\,]
$$

$$\frac{Tss \cup \{(\text{START}, t), (e.\text{ENGAGE}_i, t)\} \models W \quad P_2 \xrightarrow{e} P_2'}{\begin{array}{l}[e \in Y \cup \{\tau\} \setminus X, Tes \downarrow e = i-1] \\ \langle P_1 \ {}_X||_Y \ P_2, t, W, Tss\rangle \xrightarrow{e} \\ \langle P_1 \ {}_X||_Y \ P_2', t, W, Tss \cup \\ \{(\text{START}, t), (e.\text{ENGAGE}_i, t)\}\rangle\end{array}} \quad [\, r2\,]$$

$$\frac{Tss \cup \{(\text{START}, t), (e.\text{ENGAGE}_i, t)\} \models W \quad P_1 \xrightarrow{e} P_1' \quad P_2 \xrightarrow{e} P_2'}{\begin{array}{l}[e \in X \cap Y - \{\checkmark\}, Tes \downarrow e = i-1] \\ \langle P_1 \ {}_X||_Y \ P_2, t, W, Tss\rangle \xrightarrow{e} \\ \langle P_1' \ {}_X||_Y \ P_2', t, W, Tss \cup \\ \{(\text{START}, t), (e.\text{ENGAGE}_i, t)\}\rangle\end{array}} \quad [\, r3\,]$$

$$\frac{Tss \cup \{(\text{START}, t), (\checkmark.\text{ENGAGE}, t)\} \models W \quad P_1 \xrightarrow{\checkmark} P_1' \quad P_2 \xrightarrow{\checkmark} P_2'}{\begin{array}{l}\langle P_1 \ {}_X||_Y \ P_2, t, W, Tss\rangle \xrightarrow{e} \\ \langle P_1' \ {}_X||_Y \ P_2', t, W, Tss \cup \\ \{(\text{START}, t), (\text{END}, t)\}\rangle\end{array}} \quad [\, r4\,]$$

$$\frac{Tss \cup \{(\text{START}, t)\} \models W \quad P_1 \overset{d}{\rightsquigarrow} P_1' \quad P_2 \overset{d}{\rightsquigarrow} P_2'}{\begin{array}{l}\langle P_1 \ {}_X||_Y \ P_2, t, W, Tss\rangle \overset{d}{\rightsquigarrow} \\ \langle P_1' \ {}_X||_Y \ P_2', t+d, W, Tss \cup \{(\text{START}, t)\}\rangle\end{array}} \quad [\, r5\,]$$

The first two rules state that either of the components (P_1 or P_2) may engage an event as long as the event is not shared if and only if the evaluation on whether the current Tss appended with $\{(\text{START}, t), (e.\text{ENGAGE}_i, t)\}$ still satisfies process requirement W is true. Tss will be updated to $Tss' = Tss \cup \{(\text{START}, t), (e.\text{ENGAGE}_i, t)\}$. Rule $r3$ states that a shared event can be engaged simultaneously by both components as long as the event satisfies the requirements. Rule $r4$ is a special case for the third rule, whereas the event is the $\checkmark$ which is a special event used purely to denote termination. A new pair (END, t) is added to the Tss and hence checked. Rule $r5$ says that the composition may allow time elapsing when both the components do. We define the semantics rules for each operator in Timed CSP, which are fully explained in [13].

IV. Encoding Operational Semantics in CLP

In this section, we encode the extended Timed CSP Semantics in CLP. First, we briefly introduce the CLP language then followed by the translation of syntax and semantics into CLP rules.

A. CLP Preliminaries

Constraint Logic Programming (CLP [19]) began as a natural merger of two declarative paradigms: constraint solving and logic programming. This combination helps make CLP programs both expressive and flexible, and in some cases, more efficient than other kinds of programs. The CLP scheme defines a class of languages based upon the paradigm of rule-based constraint programming, where CLP($\mathcal{R}$) is an instance of this class. We present some preliminary definitions about CLP.

Definition 3: (Atom, Rule and Goal) An *atom* is of the form $p(\tilde{t})$, where p is a user defined predicate symbol and $\tilde{t}$ is a sequence of terms '$t_1, t_2.., t_n$'. A *rule* is of the form $A : -\tilde{B}, \Psi$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and the constraint Ψ constitute the *body* of the rule. A *goal* has exactly the same format as the body of the rule of the form $? - \tilde{B}, \Psi$. If $\tilde{B}$ is an empty sequence of atoms, we call this a *fact*. All goals, rules and facts are terms. A program is a set of rules. □

CLP has been successful as a programming language, and more recently, as a model of executable specifications. There have been numerous works which use CLP to model systems or programs and which use an adaptation of the CLP proof system for proving certain properties [22], [12]. In this work, we follow this trend and use existing powerful constraint solvers CLP($\mathcal{R}$) for mechanized extended Timed CSP.

B. Encoding Extended Timed CSP in CLP

The very initial step is to encode the extended Timed CSP models in to CLP rules. This step is automatically done by syntax rewriting. A process *"Proc* WHERE *WherePred"* is encoded to a relation *tproc(N, P, W)* in CLP. *tproc(N, P, W)* consists of three parts, where N is the name of this process, P is the CLP representation of *Proc* and W represents the *WherePred*. The syntax translation consists of two parts, process operators translation and *Where* clause translation.

All operators of the extended specification which inherited from Timed CSP are encoded into CLP rules in a compositional way. A library of all operator translation is built. For example:

- $a \rightarrow$ SKIP : `eventprefix(a, skip)`
- $a \xrightarrow{t}$ SKIP : `delay(a, skip, t)`
- $P_1 \,|||\, P_2$: `interleaving(P1,P2)`
- P_1; P_2 : `sequential(P1,P2)`

In our library, *eventprefix*(A, P) is defined to denote a process $A \rightarrow P$. *delay*(A, P, T) is the CLP form of operator $A \xrightarrow{T} P$ in Timed CSP. *interleaving*$(P1, P2)$ is to represent operator $P_1 \,|||\, P_2$ where $P1$ and $P2$ are the CLP formate of process P_1 and P_2. Relations *sequential*$/2$ is to represent a sequential operator ";".

For each process, a WHERE clause *WherePred* is encoded into W in CLP. W is a list of the form $[W1, W2, ..., Wn]$. Before we encode *WherePred* into a CLP list, we need to convert *WherePred* into conjunctive normal form (CNF), which is a conjunction of Horn clauses. Logically, $W = W1 \wedge W2... \wedge Wn$ where each W_i is a Horn clause in the form of $or(Wi_1, Wi_2)$, $not(Wi_k)$ or an atom. If there are no WHERE predicates defined for this process, $W = []$.

C. Encoding Semantics in CLP

Having defined the corresponding CLP syntax for the extended Timed CSP specifications, we devote the rest of this

section to describe how to embed the operational semantics into CLP rules. A relation of the form *tpos*(P_1, T_1, E_1, M, P_2, T_2, E_2) is used to denote the *t*imed *p*rotocol *o*perational *s*emantics, by capturing both event transition relations and evolution relations with a set of constraints. Informally speaking, $tpos(P_1, T_1, E_1, M, P_2, T_2, E_2)$ returns true if the process P_1 evolves to P_2 through either a time evolution, i.e., let $T_2 - T_1$ time units elapse (so that $M = []$), or an event transition by engaging an event e instantly ($M = e$), as long as both transitions satisfy the WHERE requirements stored in E_1. After this transition relation, the local environment might change to E_2 by adding more predicates. E_1 (and E_2) is the environment of the system, which consists not only the WHERE predicates, but also the current values of the variables appeared in the WHERE predicates.

We define the $tpos/7$[3] relation for each and every operator of Timed CSP according to the semantics presented previously in Section III.

The only transition for process STOP is time elapsing. Process SKIP may choose to wait some time before engaging the ✓ event. We use *termination* to denote this special event in CLP. Process SKIP may not be able to terminate immediately since there might be some constraints involving P.END defined in the WHERE clause. The relation *sat* is required to be evaluated before the termination. Relation $sat(E_1, A, T, E_2)$ and $sat(E_1, T, E_2)$ are used to test whether the current state fulfills the requirements. Relation $sat/3$ handles event transition and $sat/2$ handles timed transition. The $sat/3$ and $sat/2$ rules are defines as:

```
sat(E1,termination,T)
    :- get_process(E1,N),insert(end(N,T),E1,E2),
       evaluate(E2).
sat(E1,A,T)
    :- get_process(E1,N), insert(engage(A,N,T),E1,E2),
       evaluate(E2).
sat(E1,T):- evaluate(E1).
```

The first rule says that whenever there is a *termination* event, the predicate $P.\text{END} = T$ need to be validated in conjunction with all the current requirements in $E1$. Once the resultant predicate is proved to be valid, we append this predicate to the current set of predicates. The second rule is to validate the case when an event is engaged, by adding the predicate $A.\text{ENGAGE}_i = T$ to the environment. The last rule captures the timed transition relation by evaluating the current environment with the current time. Relation $get\_process(E, N)$ is to find the current named process being executed. $evaluate(E)$ is to evaluate the requirements, namely the constraint store.

In the operational semantics, there is a set of composition operators which are more complex. For instance, the rules associated with the semantics of alphabetized parallel composition operator $P_1 \ {}_X||_Y \ P_2$ are as follows.

[3]tpos/7 indicates the relation *tpos* of arity 7, same for sat/3 and sat/4.

```
tpos(para(P1,P2,X,Y),T,E1,A,para(P3,P2,X,Y),T,E2)
    :- member(A,X), not(member(A,Y)), sat(E1,A,T),
       tpos(P1,T,E1,A,P3,T,E3), update(E3,
       engage(A, T), E2).
tpos(para(P1,P2,X,Y),T,E1,A,para(P1,P4,X,Y),T,E2)
    :- member(A,Y), not(member(A,X)),sat(E1,A,T),
       tpos(P2,T,E1,A,P4,T,E3), update(E3,
       engage(A, T), E2).
tpos(para(P1,P2,X,Y),T,E1,A,para(P3,P4,X,Y),T,E2)
    :- member(E,X),member(E,Y), not(E = termination),
       sat(E1, A,T), tpos(P1,T,E1,A,P3,T,E3),
       tpos(P2,T,E1,A,P4,T,E4),update(E3,E4,E2).
tpos(para(P1,P2,X,Y),T,E1,termination,
     para(P3,P4,X,Y),T,E2)
    :- sat(E1,termination,T),
       tpos(P1,T,E1,termination,P3,T,E3),
       tpos(P2,T,E1,termination,P4,T,E4),
       update(E3,E4, end(para(P1,P2,X,Y), T),E2).
tpos(para(P1,P2,X,Y),T1,E,[],para(P3,P4,X,Y),T2,E)
    :- tpos(P1,T1,E,[],P3,T2,E),
       tpos(P2,T1,E,[],P4,T2,E).
```

Other parallel composition operation, like $||[X]||$ and $|||$, can be defined as special cases of the alphabetized parallel composition operator straightforwardly. There is a clear one-to-one correspondence between our rules and the operators which are fully defined at [13]. Therefore, the soundness of the encoding can be proved by showing there is a bi-simulation relationship between the transition system interpretation defined in Section III and ours, and the bi-simulation relationship can be proved easily via a structural induction.

V. VERIFICATION OF EXTENDED TIMED CSP

Once we encode the semantics of processes as CLP rules, well-established constraint solvers like CLP($\mathcal{R}$) [20] can be used to reason about those systems. Operational semantics defined in Section III are all encoded systematically.

A. Feasibility Checking

After specifying the tasks using extended Timed CSP in CLP($\mathcal{R}$), the very first task is to check whether the tasks are feasible before simulation or reasoning of the system. Feasibility checking is necessary because there might be a conflict among the set of WHERE clauses of a system, which potentially invalidates any proving result. To perform this task, the conjunction of the WHERE predicates and the healthiness conditions are checked.

The output of the feasibility checking is either *yes* if the tasks are feasible or else *no*. In case the tasks are infeasible, i.e., there is no way to satisfy all the constraints, a minimum set of predicates which conflict each other can be generated so as to facilitate user correction easily. We use the CLP($\mathcal{R}$) predicate $feasibility\_checking(N, S)$ to fulfill this purpose, where N is the name of the process that is to be checked, and S is the minimum conflict set. If the process N is a feasible process $feasibility\_checking/2$ returns false, otherwise the minimum conflict set S is generated and returned.

B. Reasoning about Safety and Liveness

Feasibility checking is to check whether the tasks modeled in extended Timed CSP are feasible. Once it is proven to be feasible, we can reason about safety or liveness properties by making explicit assertions.

Relation $reachable(P, Q, E1, E2, T1, T2, Tr)$ is defined to explore the full state space if necessary. It states that "process P starts at $T1$ with environment $E1$ and is able to be executed to Q at $T2$ with environment changed to $E2$ via trace Tr".

```
reachable(P, P, _, _, T, T, []).
reachable(P, Q, E1, E2, T1, T2, N)
    :- tpos(P, T1, E1, A, P1, T3, E3),
       (A=t(_);A==tau;A=reccall(_)),
       not table(P1), assert(table(P1)),
       reachable(P1, Q, E3, E2, T3, T2, N).
reachable(P, Q, E1, E2, T1, T2, [E|N])
    :- tpos(P, T1, E1, A, P1, T3, E3),
       not (A=t(\_);A==tau;A=reccall(_)),
       not table(P1),assert(table(P1)),
       reachable(P1, Q, E3, E2, T3, T2, N).
```

$reachable/7$[4] is used to build assertions for various property checking. The first property of interest is to find one particular feasible execution for process, provided that the process is feasible. Relation $trace(P, Tr, T)$ is able to generate such feasible trace Tr of process P, whose execution time is T.

```
trace(P,Tr, T):- not feasibility_checking(N,_),
       init_Env(N,E), reachable(P,_,E,_,0,T,Tr).
```

Reachablity checking is easily carried out by executing the goal "$?-trace(P, Tr, T), property(Tr, Prop)$", which is to find a trace Tr that satisfies some property $Prop$. For example, event a is always engaged before b in Tr.

One property of special interest is deadlock-freeness. Relation $deadlock(P, Tr)$ is used to check the deadlock-freeness property, by trying to find a counterexample where P is deadlocked at some trace Tr.

```
deadlock(P, Tr) :- init_Env(P, E),
      reachable(P, P1, E, E2, 0, T2, Tr),
      (tpos(P1, T2, E2, [t(_)], Q1, T3, E3) ->
      not(tpos(Q1, T3, E3, A, _, _, _), not A=[t(_)]);
      not(tpos(P1, T2, E1, A, Q1, T3, _),not A=[t(_)])).
```

It states that a process P at time 0 may result in deadlock if it can evolve to the process expression Q at time $T2$ where no event transition is available neither at $T2$ nor at any later moment. The last line outputs the trace which leads to a deadlock. Alternatively, we may present it as the result of the deadlock proving. Note that the above is different from the deadlock checking for standard Timed CSP as presented in [12]. Here the WHERE clauses at each step must be fulfilled. In general, a deadlock-free Timed CSP process may become a non deadlock-free process after it is enriched with certain WHERE clauses. It is, however, also possible for a non deadlock-free process to become deadlock-free.

We can also find the execution duration of a specific event, more specifically, the range of time that the event is able to be engaged. Relation $engage\_time(P, E, R)$ is defined for the purpose, which is to find the range R of the engage time of event E in process P. The detailed definition for all relations can be found in [13].

```
engage_time(P, E, []) :- not happen_at(P, E, _).
engage_time(P, E, R)  :- happen_at(P, E, T),
          union(R, T, R1), engage_time(P, E, R1).
```

[4]reachable/7 indicates the relation *reachable* of arity 7.

where R is the range of engaged time of event E in process P which is generated after executing the relation.

VI. VERIFICATION OF AUTHENTICATION

For verifying timed security protocols, protocols are firstly modeled in extended Timed CSP models, and then translated into CLP programs. Properties which need to be verified are encoded into CLP goals using relations defined in Section V. In this section, we show how to define and verify timed security properties, including timed authentication properties. Moreover, by using timing information of each protocol run, potential attacks are also to be found.

A. Timed Authentication Property

Authentication property is very important in security protocols. Protocols need to accomplish authentication of the *Initiator* and *Responder*. [16] classified a set of authentication requirements. We will focus on timed non-injective agreement, which is an extension of the non-injective agreement defined in [16].

Definition 4 (Timed Non-Injective Agreement): A timed security protocol guarantees *timed non-injective agreement*, only if responder B thinks it has completed a run of the protocol with A using data D, then A was actually running the protocol with B using data D. To check the authentication property, signals are added to principals to indicate principal B has completed a protocol run with A and A is actually running a protocol with B, whenever necessary. □

Our method is to insert signals, which are special kind of events, into each process, and then to check the corresponding relationship of these signals. In our approach, event $commit.B.A.D$ is used to denote that principal B has completed a protocol running with A using data D, $run.A.B.D$ as principal A is running a protocol with B using data M.

$$
\begin{aligned}
Initiator \mathrel{\widehat{=}} & \ start\_encrpt \rightarrow end\_encrpyt \rightarrow \\
& run.a.b.\{t_a.b.k_{ab}\} \rightarrow \\
& (\mu X \bullet send.A.S.\{t_a.b.k_{ab}\}_{k_{as}} \\
& \rightarrow (receive.s.a.\{ack_{sa}\} \rightarrow \text{SKIP} \overset{d_1}{\triangleright} X)) \\
& \triangledown\{d_2\}\ \text{SKIP} \\
& \text{WHERE } t_a = send.a.s.\{t_a.b.k_{ab}\}_{k_{as}}.\text{ENGAGE}
\end{aligned}
$$

Properties which need to be verified are specified as assertions. There is a one-to-one relationship between the runs of A and B for protocol satisfying timed non-injective agreement property. The timed non-injective agreement means that once there is a *commit* event, there should be at least one *run* event appearing previously. The assertion is defined as:

$$
\begin{aligned}
non\_inj\_agr(P, Tr) :- & trace(P, Tr, \_), \\
& end(Tr, commit.B.A.M), not\ in(Tr, run.B.A.M).
\end{aligned}
$$

This predicate is to find a trace Tr, where there is no event *run* occurred before event *commit*. Tr is an instance of an attack. Predicate $trace(P, Tr, T)$ is defined in previous section. After checking the timed non-injective agreement property over WMF protocol, such attack has been found where responder B finds that it has more than one sessions

with initiator A but in fact there should be only one [1]. We also model the Needham-Schroeder Public-Key protocol as process *NSP*, by executing the following goal:

$$? - non\_inj\_agr(NSP, Tr).$$

We are able to find an attack where the responder B commits a session with the initiator A, but A did not establish a protocol run with B, where A is running the protocol with the intruder I [23]. The trace is:

$$\langle start\_encrypt, end\_encrypt, send.A.I.M, receive.A.I.M, send.I.B.M, receive.I.B.M, send.B.I.M2, receive.B.I.M2, send.I.A.M2, receive.I.A.M2, send.A.I.M3, receive.A.I.M3, send.I.B.M4, receive.I.B.M4, commit.B.A.D\rangle$$

Preventing Attacks: By preventing the authentication attack, we propose appending a constraint WHERE TES $\downarrow$ $run.A.B.D \geqslant$ TES $\downarrow$ $commit.B.A.D$ to the protocol process to guarantee number of event *commit* is always less than event *run* (shown in Example 2). The revised Wide Mouth Frog protocol is modeled as *WMF_R*1.

Another approach of preventing attacks is by changing the timeout values. There is a particular timed authentication attack for Wide Mouth Frog protocol where the intruder can extend the life time of a (possibly compromised) key *Kab* as wanted, whereas A and B think that it has expired and been destroyed [26].

$i.1.\quad A \rightarrow S$: – A, {Ta, B, Kab}Kas
$i.2.\ S \rightarrow I(B)$: – {Ts, A, Kab}Kbs
$ii.1.\ I(B) \rightarrow S$: – B, {Ts, A, Kab}Kbs
$ii.2.\quad S \rightarrow A$: – {T's, B, Kab}Kas
$iii.1 I(A) \rightarrow S$: – A, {T's, B, Kab}Kas
$iii.2.\ S \rightarrow B$: – {T"s, A, Kab}Kbs ....

This attack cannot be checked using timed non-injection agreement because there is exactly one session for both initiator and responder. For step $i.2$ I takes the place of B to get the message from server S where S should be expecting an *ack* from B. If S does not get the *ack* within the timeout window, it could choose to terminate the session. Let d_n be the network transaction delay for each message passing. S should be able to get the *ack* message from B after $2 * d_n$ if there is no attack. By changing the timeout window for S to less than $6 * d_n$, this attack will be avoided because B can only send *ack* to S after step $iii.2$, which takes more than $6 * d_n$ time unites. We model the Wide Mouth Frog protocol with bounded timeouts in *WMF_R*2.

We list a set of properties over Wide Mouth Frog protocols, where the results are shown in Figure 2. *CoreWMF* is the protocol without intruder; and *CoreWMF + Intruder* is the protocol with intruder I. The results are computed in minutes or even seconds in PC with 2.83GHz Intel Q9550 CPU and 2 GB memory. Comparison with other approaches are ignored since there is no other tool supporting timed analysis of security protocols.

B. Using Timing Information

We are able to check other timing properties of the protocol. For example, find the minimum and maximum execution time of a run of a protocol P by finding P.END, using predicate $engage\_time(P, termination, R)$ defined in Section V.

$$execution\_time(P, R) : -engage\_time(P, termination, R).$$

where *termination* is a special event denoting the end of the execution, R is the range of execution time of protocol P.

By using the minimum and maximum execution of a protocol run, we can also check *timing authentication* properties. In our approach, if there is a run of a protocol between two principals A and B, it must be finished within a time interval $[T_{min}, T_{max}]$, without intruders. If a protocol run ends before T_{min}, this may be a result of an attack which omits at least one instructions or performs at least one instructions faster than it is expected. If a protocol run ends after T_{max}, this may be a result of an attack performing with extra actions, such as replays. We define a predicate $time\_attack\ (P, Tr, Min, Max)$ to find a trace which exceeds the time interval $[min, max]$, which is also an instance of an attack.

$$time\_attack(P, Tr, Min, Max) : - \\ trace(P, Tr, T), (T < Min;\ T > Max).$$

Assume for a protocol run, once a sender sends a message to the receiver, the message reaches the receiver within [2, 4] time units because of the network delay. for WMF protocol, without introducing an intruder, the execution time of a protocol run is [4,8]. By executing goal $?-\ time\_attack\ (WMF, Tr, 4, 8)$. We find a trace whose execution time is more than 8. If the execution time of a protocol run exceeds the expected time interval, there must be some attack in this protocol run. But if the execution time is within the time interval, attack-free is not guaranteed.

VII. RELATED WORKS AND CONCLUSION

In this work, we proposed a new method for modeling and analyzing security protocols which consists of various timing aspects. To fulfill the aim, we substantially extended Timed CSP with capabilities to stating complicated and critical timing requirements of timed security protocols, in a compositional way. Based on our previous work on building a reasoning tool for Timed CSP, a prototype mechanized proving system, based on CLP($\mathcal{R}$) to verify various properties over systems modeled in this extended specification has been built. We model principals as processes, as well as the cryptograph device, including timestamps, timeout, retransmissions and delays. The timed non-injective agreement authentication property can be verified using our underlying reasoning engine, which can be easily extended to verify other authentication properties. We propose a novel approach to find timing attacks using timing information of protocol sessions. We can also model timing requirements of the protocols in WHERE predicates and verify other timing properties of the protocols.

There are many works on analyzing security protocols. In literature, methods for formal verification of security protocols

Property	Protocol	Description	Result
P1	CoreWMF	non-injection agreement	Yes
P2	CoreWMF	timed non-injection agreement	Yes
P3	CoreWMF + Intruder	timed non-injection agreement	No
P4	CoreWMF + Intruder	replay attack	Yes
P5	CoreWMF + Intruder	timed authentication attack	Yes
P6	WMF_R1 + Intruder	timed non-injection agreement	Yes
P7	WMF_R1 + Intruder	replay attack	No
P8	WMF_R1 + Intruder	timed authentication attack	Yes
P9	WMF_R2 + Intruder	timed non-injection agreement	Yes
P10	WMF_R2 + Intruder	replay attack	No
P11	WMF_R2 + Intruder	timed authentication attack	No

Fig. 2. Analysis of Wide Mouth Frog protocols

do not take time into account, and this choice simplifies the analysis [3]. Powerful theorem provers like Isabelle and PVS have been applied to verify timed dependent security properties [2], [14]. A common practice in the area of modeling and verification of security protocols is to abstract away timestamps [4]. Our approach is similar to [26] which uses CSP to model and analyze untimed security protocols. Recently there have been also other approaches to verify such protocols [10], which does not discuss timeout and retransmissions. One more recent work, using Timed Automata for verifying timed security [7], also introduces timeout. The difference of our timeout and retransmissions is that we propose a bounded timeout, which also consider timeout over retransmissions. Moreover, they cannot model timedstamps which needs global synchronization on clocks.

As a future work, we will apply this extended Timed CSP specifications to other domains, such as timed scheduling problems, complex real-time systems and etc. For analyzing timed security protocols, we will expand the verification of security properties, such as secrecy, integrity, fairness and the timing properties such as the time range for an easy attack. For our underlying reasoning engine, there are many potential improvements to be explored to reduce the execution time, such as symmetry reduction and heuristics.

REFERENCES

[1] R. Anderson and R. Needham. Programming satan's computer. In *in Computer Science Today*, pages 426–440. Springer-Verlag, 1995.

[2] G. Bella and L. C. Paulson. Kerberos version iv: Inductive analysis of the secrecy goals. In *ESORICS 98, LNCS 1485*. Springer, 1998.

[3] M. Boreale and M. G. Buscemi. Experimenting with sta, a tool for automatic analysis of security protocols. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 281–285, New York, NY, USA, 2002. ACM.

[4] L. Bozga, Y. Lakhnech, and M. Prin. Pattern-based abstraction for verifying secrecy in protocols. *Int. J. Softw. Tools Technol. Transf.*, 8(1):57–76, 2006.

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

[6] R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*. Springer-Verlag, 2002.

[7] R. Corin, S. Etalle, P. H. Hartel, and A. Mader. Timed analysis of security protocols. *J. Comput. Secur.*, 15(6):619–645, 2007.

[8] J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.

[9] G. Delzanno and S. Etalle. Proof theory, transformations, and logic programming for debugging security protocols. In *LOPSTR '01: Selected papers from the 11th International Workshop on Logic Based Program Synthesis and Transformation*. Springer-Verlag, 2001.

[10] G. Delzanno and P. Ganty. Automatic verification of time sensitive cryptographic protocols. In *TACAS 2004*.

[11] D. Dolev and A. C. Yao. On the security of public key protocols. In *SFCS '81*, pages 350–357, Washington, DC, USA, 1981. IEEE Computer Society.

[12] J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In *ICFEM 2006*, pages 342–359, 2006.

[13] J. S. Dong, J. Sun, and X. Zhang. Reasoning of Timed CSP and Extensions. In *Technical report*. http://www.comp.nus.edu.sg/∼zhangxi5/tp.pdf.

[14] N. Evans and S. Schneider. Analysing time dependent security properties in csp using pvs. In *ESORICS '00*, pages 222–237, London, UK, 2000. Springer-Verlag.

[15] Formal Systems (Europe) Ltd. Failure Divergence Refinement: FDR2 User Manual. 1997.

[16] G.Lowe. A hierarchy of authentication specifications. In *CSFW'97*, pages 31–44, 1997.

[17] G. Gupta and E. Pontelli. A Constraint-based Approach for Specification and Verification of Real-time Systems. In *RTSS '97*, page 230, 1997.

[18] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[19] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming,1994*.

[20] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP(R) Language and System. *ACM Trans. Program. Lang. Syst.*, 14(3):339–395, 1992.

[21] J. Jaffar, A. E. Santosa, and R. Voicu. A CLP Proof Method for Timed Automata. In *Real-Time Systems Symposium*, pages 175–186, 2004.

[22] J. Jaffar, A. E. Santosa, and R. Voicu. Modeling Systems in CLP. In *ICLP'05*, 2005.

[23] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, London, UK, 1996. Springer-Verlag.

[24] G. M. Reed and A. W. Roscoe. A Timed Model for Communicating Sequential Processes. In L. Kott, editor, *ICALP*, volume 226 of *Lecture Notes in Computer Science*, pages 314–323. Springer, 1986.

[25] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[26] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. Modelling and analysis of security protocols, 2001.

[27] S. Schneider. *Concurrent and Real-time System: The CSP Approach*. JOHN WILEY & SONS, LTD, 2000.

[28] S. A. Schneider. An Operational Semantics for Timed CSP. In *Proceedings Chalmers Workshop on Concurrency, 1991*, pages 193 – 213.