# Visual Security Protocol Modeling

J. McDermott
Center for High-Assurance Computer Systems
Naval Research Laboratory
NSPW 2005 Submission

April 18, 2005

## 1 Introduction

The *model-driven-approach* [32] to software construction promises to improve software quality and reduce costs through automatic construction of software from (visual) models. Visual modeling is slowly becoming a common practice for software developers, so the hope is that practitioners will be comfortable with using visual models to design their software. (In this paper, we use the unqualified term *modeling* to mean visual modeling.)

The force of common practice is defining the model-driven-approach in terms of the Object Management Group's Model Driven Architecture or MDA. The core of the MDA is UML 2.0 [26]. Neither UML 2.0 (henceforth UML) or MDA treats security as much more than a service; there are no models for security per se.

This raises the question of what security-specific aspects of software development, if any, need coverage in this paradigm. This paper argues that there are security-specific issues that cannot be modeled well with existing features of MDA. These issues need adequate coverage in model-driven approaches.

One of the most significant security-specific aspects of software development not covered by the MDA is the *security protocol*. Security protocols are sequences of allowable interactions between *principals*. A principal is an entity that participates in a security system. Security protocols are not necessarily about cryptography; one of our examples will model a security protocol that involves no cryptography at all.

The UML candidates for visual modeling of security protocols all have shortcomings. Existing alternatives outside of UML also have similar problems, for various reasons. Some of these difficulties are visual modeling issues and others are semantic issues. One of the most critical semantic requirements for modeling security protocols is the ability to define all traces of a protocol with a single model as opposed to being able to describe any trace with a single model. Explicit definition of all traces is necessary because each bad trace has the potential to become a security flaw. A highly desirable visual modeling feature is

event-based modeling, as opposed to state-based modeling. The distinction is that state-based modeling is best for designing *reactive* behavior while event-based modeling is best for designing *interactive* behavior. State-based modeling requires us to work with internal computational aspects, such as states or triggers, to construct the traces of a protocol. An event-based modeling paradigm lets us work directly with the external events and traces of a protocol.

# 2 Motivation

The core purpose of visual modeling, as opposed to other forms of modeling, is presentation and understanding. Formal verification, machine-generated implementation, and other automatic processing are probably done better with text-based models. So our interest is in security protocol modeling that has good visual properties for presentation and understanding, without sacrificing soundness that supports translation into text-based models. This leads to the following criteria for security protocol modeling:

- The visual formalism should be *event-based*. It should focus on interaction patterns between principals and abstract away from details of internal computations. The importance of this is underscored by the fact that existing security protocol modeling tools use event-based visual models, rather than state-based models.
- The visual formalism should support *composition* in a natural way, so that models can be constructed from components that identifiably correspond to the principals of the protocol.
- The visual formalism should be *comprehensive*. It should be capable of defining all traces of a protocol by means of a single diagram.
- The visual formalism should be *concise*. Defining a complex protocol should not require an explosion of modeling details. (An event-based visual formalism can fail to be concise.)
- The visual formalism should have a *well-defined* syntax and semantics.

With these criteria (*event-based, composable, comprehensive, concise, well-defined*) we can assess the suitability of the various MDA/UML models for security protocol design and analysis. We can also investigate the usefulness of other modeling approaches that are not part of the MDA suite.

## 2.1 UML Candidates

To model security protocols in UML, we must use one or more of the available modeling mechanisms: *actions, activities, interactions, state machines*, or *use cases*. Use case models are high-level requirements tools and use the other visual modeling techniques to describe behavior, so they are not candidates for

modeling any but the most rudimentary concepts of security protocols. *UML Actions* include constructs such as BroadcastSignal, ReadVariable, and WriteLink; they correspond to individual events, methods, messages, or calls. Thus, they are also not suited to modeling complete security protocols.

*UML Activities* organize UML Actions into structures that resemble Petri nets. UML Activities employ control- and data-flow relationships in their Petri-net-like structures, which is less desirable when the issue is protocols and we wish to avoid details about internal computations.

*UML Interactions* are similar to ITU Standard Z.120 *Message Sequence Charts*, or the older *UML 1.x Sequence Diagrams*: a collection of vertical lifelines with message flow between the lifelines shown horizontally. Both UML Interactions and ITU Message Sequence Charts have semantic problems. Damm and Harel have provided a well-defined semantics for these kinds of diagrams, in a visual modeling technique called *Live Sequence Charts* [9]. All of these "sequence-diagram" modeling paradigms have the critical strength of being event-based: they model sequences without internal computational detail. That is, they model behavior directly in terms of protocol traces. Unfortunately, they all have limited usefulness in modeling security protocols because each diagram defines only a subset of the traces of a protocol. The nature of these diagrams is that they visually enumerate traces and lack the power of set theory or process algebra to explicitly define all possible traces of a combination of principals. For example, suppose we use the BPA [1] process algebra of Bergstra and Klop [1] to define $P = a \cdot P$, the process $P$ that does event $a$ and then acts like process $P$. If the expression $\mathsf{traces}(P)$ means the set of all traces of process $P$ and the symbol $\frown$ denotes concatenation of traces then we can use set theory to explicitly define all of the traces of $P = a \cdot P$ as

$$\{\langle\rangle\} \cup \{\langle a\rangle \frown tr \mid tr \in \mathsf{traces}(P)\}$$

while the corresponding "sequence-diagram" enumeration approach is equivalent to the symbolic listing of each possible trace

$$\langle\rangle, \langle a\rangle, \langle a, a\rangle, \ldots$$

As soon as there is a modest variation in the pattern of the traces, this enumeration approach begins to break down. In contrast, process algebra or set theory provides us a complete definition in a single model but still allows us to unwind the model to see or check any trace. The visual modeling equivalent of set theory or process algebra is needed to solve this problem.

*UML State Machines* would appear to offer some promise. They are based upon (but are not the same as) the object-oriented version [13] of Harel's elegant *statechart* [12] visual formalism. Since statecharts are a well-defined visual model, UML State Machines should be able to define completely any security protocol, with a single model. Unfortunately, UML State Machines have some problems: 1) received events are modeled by a different mechanism that sent

[1] Basic Process Algebra

events, 2) the semantics are run-to-completion which poses problems for modeling some forms of recursion (Tenzer and Stevens provide good examples of this) [34], and 3) some of the events are not atomic [22]. Some of these problems are avoided by the concept of *UML Protocol State Machines*. UML Protocol State Machines are like UML State Machines without UML Activities. That is, a UML Protocol State Machine only has triggers associated with its transitions while the more general UML State Machine also has UML Activities associated with its transitions. The effect of this is that a UML Protocol State Machine can describe one side of an interaction between two security principals: either the sequence of requests a principal can make or the sequence of responses that that a principal can provide. This is sufficient for constraining interfaces but not for modeling a complete security protocol.

From these circumstances we can conclude that UML is not well-suited to modeling security protocols. This leads us to examine other visual modeling techniques outside of UML, to see if they are better tools for modeling security protocols.

## 2.2 Existing Visual Models Outside of UML

We have already mentioned Live Sequence Charts as a well-defined event-based modeling technique. The problem of needing more than one diagram to define all of a protocol remains. Another possibility is a visual representation of *labeled transition systems*. A labeled transition system or LTS is a triple $(\Gamma, A, \rightarrow)$ where $\Gamma$ is a set of *configurations*, $A$ is a set of events, and $\rightarrow$ is a ternary relation: $\rightarrow \ \subseteq \ \Gamma \times A \times \Gamma$. Intuitively, the relation $\rightarrow$ represents the transitions from one configuration to another; $\langle \gamma, a, \gamma' \rangle \in \rightarrow$ is usually written as $\gamma \xrightarrow{a} \gamma'$. Labeled transition systems are ideal for machine representation and processing of event systems. The problem with labeled transition systems as a visual modeling paradigm is the same problem that lead to the development of statecharts: "the unmanageable, exponentially growing multitude of states, all of which have to be arranged in a 'flat' unstratified fashion" [12]. They are not concise. Current LTS work is turning to algebraic treatments to overcome this difficulty. *Petri nets* were developed by Carl Petri [27] for formal modeling of concurrency, nondeterminism, and communication. Petri nets are well-defined formally and have a large body of literature. They are useful for a wide range of problems including workflow and performance modeling. The difficulty with using them to model security protocols is the presence of computation details: initial markings, places, transitions, and data flow. They are not event-based. Another difficulty is that Petri-net-based models are not naturally composable in terms of security principals.

*Port state machines*, a formalism developed by Mencl [22], have removed the semantic difficulties associated with UML State Machines, while retaining the semantic clarity of statecharts. Furthermore, port state machines also address modeling details needed for object-oriented programming, which the original statecharts lack. However, because of this and their state-based nature, port state machines have too much computational detail for modeling security pro-

tocols. They are not event-based.

Harel's original statecharts are a good candidate for modeling security protocols, because they lack the extra details needed to model object-oriented programming issues. They are semantically sound and can define an entire protocol with a single diagram. Statecharts also have excellent visual modeling characteristics. They are not event-based and require consideration of states and transitions as well as the events they model. We would prefer a more directly event-based modeling paradigm.

Walters has designed RDT [36] as a formal visual language based on activity diagrams. RDT is designed foremost for visual clarity, just what is needed for visual modeling of security protocols. It would be a good candidate but it uses an LTS form of depicting behavior, so it is not event-based.

Another alternative we have not considered up to now is a graphical form of *process algebra*. Process algebras are event-based but avoid the explosive complexity of labeled transition systems by means of algebraic operators. Process algebras view processes as abstract trace generators and provide means for composing processes to define more complex trace generators.

Cleaveland, Du, and Smolka developed *Graphical Calculus of Communicating Systems* (GCCS) [7] as part of the Concurrency Factory tool [8]. The GCCS visual notation is based on Milner's CCS [25] process algebra but the diagrams are visual depictions of labeled transition systems. GCCS diagrams have the same visual limitations as basic labeled transition systems: they are not concise.

Cerone developed *Visual Process Algebra* or VPA [5], a modeling technique based on combinations of the CCS, CSP [17], and Circal [24] process algebras. The VPA approach models processes as boxes with ports to indicate communication and thus has the potential to be event-based. Unfortunately, VPA uses an LTS or state-machine approach within each box to model the behavior of the corresponding process. For security protocol modeling we would really prefer an approach that avoids labeled transition systems altogether.

Gilmore and Gribaudo [11] extended the *DrawNET* tool to model the PEPA [16] stochastic process algebra. The DrawNET tool is oriented towards performance modeling; the graphical representation of process algebra retains the Petri nets of the underlying tool, so the DrawNET representation is not really well-suited to modeling security protocols.

The *gCSP* (for graphical CSP) tool, developed by Hilderink, Jovanovic, et al. [15, 18] is the most ambitious graphical form of process algebra to date. Processes are denoted as circles in gCSP. Lines connecting the processes denote composition via the various operators of CSP. A surprising omission is the graphical modeling of events and their ordering within a sequential process. That is, even though gCSP can cleanly show sequential processes $P$ and $Q$ in parallel $P \parallel Q$, it cannot show the events that make up sequential process $P$ (or $Q$). This is not a difficulty for control applications that gCSP has been applied to, but it is critical for modeling security protocols.

From a security protocol modeling perspective, the gCSP notation is interesting because it presents a contrast to the graphical modeling paradigm proposed in this paper. Process algebras are strongly compositional. It is dif-

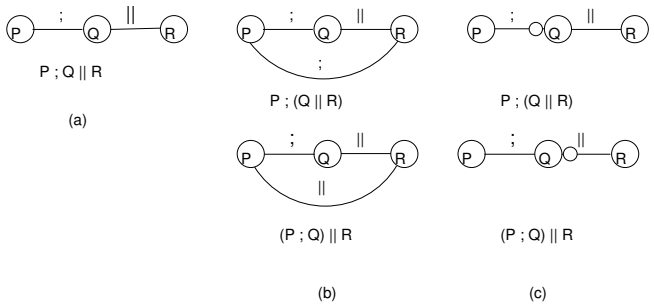


Figure 1: Resolving Compositional Ambiguity in gCSP

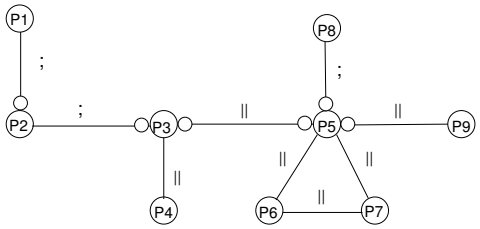


Figure 2: Complex Composition in gCSP

ficult to present complex process algebra relationships graphically. Figures 1 and 2 illustrate this difficulty in the gCSP notation. Figure 1 shows the process algebra fragment $P; Q \| R$ which is an ambiguous term specifying the sequential (via the ; operator) and parallel (via the $\|$ operator) composition of processes $P, Q$ and $R$. Figure 1 (a) shows how this ambiguity can be drawn in gCSP. Figure 1 (b) shows how this ambiguity can be resolved by drawing cycles to add arcs for all relationships. This is problematic in complex compositions since the diagram tends to become a fully connected graph. The gCSP notation has a clever solution to this, shown in Figure 1 (c), where a smaller circle is used on one side to denote the precedence. The notation is well-defined and capable of automatic simplification. However, in complex situations, the notation becomes difficult to read, as shown by Figure 2. However, it is the lack of explicit events that renders gCSP unsuitable for security protocol modeling.

## 2.3 Security Protocol Modeling Tools

Another possibility is the (visual) modeling provided by security-protocol-specific tools. Most of the these tools have visual modeling components and it is possible that we may find a satisfactory (from the visual modeling perspective) language or technique there. Considering these tools will also clarify our emphasis on presentation and understanding as opposed other purposes such as verification or analysis. Clearly the existing tools are effective for those other purposes.

The Security Protocol Engineering and Analysis Resource (SPEAR) tool [30]

provides *multidimensional protocol analysis*. Multidimensional protocol analysis combines several non-visual modeling approaches in order to get a more complete picture of the security of a cryptographic security protocol. The SPEAR tool incorporates multidimensional protocol analysis under a graphical user interface. Unfortunately, SPEAR uses message sequence charts to visually model security protocols. Its graphical language is not comprehensive.

The Common Authentication Protocol Specification Language (CAPSL) and MuCAPSL, its group multicast protocol version, is a formal language for specifying cryptographic security protocols [23]. CAPSL is well-defined, concise, comprehensive, and compositional. CAPSL models can be translated into many forms and several cryptographic protocol analysis tools have CAPSL support. Unfortunately, there is no visual form of CAPSL per se.
The Convince tool is a pioneer effort in visual modeling of cryptographic security protocols [19]. Convince uses a text-based formal language based on BGNY [3] logic. Unfortunately, the characteristics of BGNY do not carry over into the visual modeling language, which is essentially a version of UML. In particular, protocol steps are modeled visually using message sequence charts.

One security protocol analysis tool that does use a distinct security-specific visual language is the NRL Protocol Analyzer (NPA) [21]. NPA has its own text-based language NPATRL (pronounced "N Patrol") that models a wide range of security protocol requirements. NPATRL is an event-based language for expressing trace properties. It uses familiar logic operators and one temporal operator to define logical properties of events or traces. The NPA tool has a corresponding tree-structured language for visual modeling of NPATRL specifications [6]. The visual language is event-based, concise, and well-defined. Our motive for looking further is that the visual NPATRL language is a trace-property-language while we are looking for a protocol-definition language. That is, the visual NPATRL language does not define the traces of a particular protocol, but the properties (i.e. requirements) of a good protocol. We are looking for a language that can define protocols as they operate, good or bad.

# 3 The GSPML Alternative

We have seen that the general purpose features of UML are inadequate for modeling security protocols. More well-defined visual paradigms including gCSP are also not suited to modeling security protocols. The visual interfaces to current security protocol modeling tools also do not use better forms of modeling.

We propose GSPML as an alternative. (At the time this paper was written, GSPML did not stand as an acronym of any particular name.) The goal of the GSPML alternative is to provide a visual modeling language suitable for the security-specific problem of protocol modeling. The emphasis is on a solid visual model with complete syntax and semantics, rather than tool application via the specific semantics. Given a well-defined visual modeling language, a variety of formal techniques could be used, including many with semantics that differ from the semantics of GSPML (e.g. NPATRL, CAPSL, strand spaces

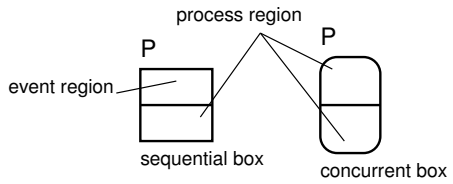


Figure 3: Basic Boxes of GSPML

[10, 33], or a general LTS).

The GSPML alternative is well-defined, event-based, compositional, comprehensive, and concise. This is demonstrated by the diagram at the end of this paper (see Figure 11) that defines a complete model of the Yahahlom cryptographic security protocol. This diagram fits on a single page but defines all possible traces of the protocol interacting with a Dolev-Yao intruder. So a single GSPML diagram can define not only all the correct behavior of a protocol but also its behavior under many attacks.

Our presentation here in Section 3 does not define a semantics for the language but provides an introduction and demonstrates the applicability of GSPML. The meaning of well-formed GSPML diagrams is compatible with several forms of process algebra but it is not necessary to understand process algebra in order to understand the GSPML presented in this paper. It is necessary to understand that GSPML models are arrangements of nested rectangles or boxes that define trace generating processes.

In GSPML, every trace-generating process is defined by either a *process box* or a *process box name*. (For the rest of this paper we will use the term process and box interchangeably.) There are two major distinctions between boxes: *sequential boxes* and *concurrent boxes*, as in Figure 3. A sequential box has rectangular corners and models sequential processes. A concurrent box has round corners and models concurrent processes. A process box name (or more simply, box name) may only appear as a label for a box, or inside a *process region* of a box. A process region may have only one box or box name in it. Sequential boxes also have *event regions* that contain the events of a GSPML model. When all of the events in the event region of a sequential box have occurred then the sequential box is replaced by or becomes the box contained or named in the process region below the event region.

We present the details of GSPML by examples of its use. We give two examples: first a cryptographic security protocol and then a non-cryptographic security protocol.

## 3.1 The Yahalom Protocol

Our example of a cryptographic security protocol is the Yahalom protocol [4]. The GSPML model is based on the CSP process algebra model presented by Ryan, Schneider, et al. [29]. Readers interested in process algebra modeling

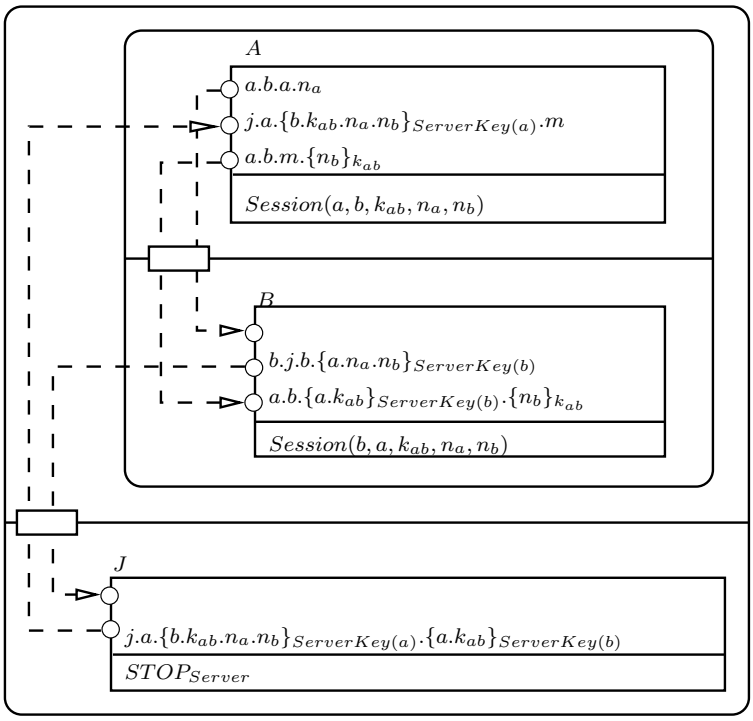


Figure 4: A GSPML Model of One Run of the Yahalom Protocol

of security protocols, as opposed to exposition of the GSPML language, should consult their work.

The Yahalom protocol is used to establish a session key $k_{ab}$ between two principals $A$ and $B$, via a server $J$. Figure 4 shows a simple GSPML diagram of a single run of the protocol, where principal $A$ initiates a session with principal $B$. The protocol run is simplified in the sense that the principals are assumed to be somehow prepared to synchronize on each other's nonces and the session key, in advance of the protocol run. That is, each event contains precisely the nonces $n_a, n_b$ and session key $k_{ab}$ to make this run work. None of the three named boxes in Figure 4 is defined as being prepared to deal with any possible well-formed nonce or session key.

The goal of Figure 4 is to introduce the protocol, not model it. That is, the GSPML of Figure 4 is playing the role of the usual message sequence diagram used to introduce a cryptographic protocol. So Figure 4 shows that GSPML can be used for explanation as well as definition of security protocols.

Figure 4 provides a good non-trivial first example of GSPML. The two outermost (unnamed, round-cornered) boxes are concurrent boxes that model the concurrent interaction of the principals $A$ and $B$ and the server. Each concurrent box has two process regions. The outermost concurrent box has the next inner concurrent box in one of its process regions and a sequential box named $J$ in its other process region. The sequential box $J$ contains the events that model the protocol steps of the server in a single run of the Yahalom protocol. The second concurrent box contains the sequential boxes named $A$ and $B$ defining the corresponding protocol steps for the initiator and responder.

Each event in a sequential box is denoted by a small circle, called an *event symbol*, on the left boundary of the box's event region. The top-to-bottom order of the event symbols defines the sequential order of the events for that box. So the events of sequential box $A$ at the top of Figure 4 are

$$
\begin{array}{l}
a.b.a.n_a \\
j.a.\{b.k_{ab}.n_a.n_b\}_{ServerKey(b)}.m \\
a.b.m.\{n_b\}_k ab
\end{array}
$$

Sequential boxes communicate or share their events via *interface port symbols* on enclosing concurrent boxes. Concurrent boxes with interface port symbols are *parallel boxes* that define communication between their components. An interface port symbol is a small rectangle placed on the boundary between the process regions of a parallel box.

Shared events are connect by *synchronization lines*. The synchronization lines shown in Figure 4 are an example of *concrete* synchronization lines because they connect event symbols directly. They are also *directed* synchronization lines because they have an arrow at the head of the line. The event at the head of a directed synchronization line is the *destination event* and the event at the tail is the *source event*. Directed synchronization lines depict transmission or communication events. (GSPML also has undirected synchronization lines but they are not used in this model.) Synchronization lines may be drawn anywhere

that provides clarity while connecting the events, but must pass through the interface port symbol that defines the parallel combination.

Events in GSPML may have *compound names*. The event itself is atomic but various information about the event can be represented using a "dot" separator, as in $x.y$ between the name components $x$ and $y$.

In Figure 4's model of a run of the Yahalom protocol, the events have compound names with the first component indicating the *sender* for that event and the second component indicating the *receiver* of the event, not to be confused with the source and destination event symbols of the synchronization line. For example, the first event of box $A$ is the compound event $a.b.a.n_a$: a transmission from source $a$ to destination $b$ of the message $a.n_a$. The directed synchronization line from event $a.b.a.n_a$ in box $A$ to the unnamed destination event in box $B$ tells us that the unnamed destination event has the same name as the source event symbol. This models the reception of the named event transmitted by box $A$.

Figure 4 also contains directed synchronization lines connecting events with different names at source and destination. For example the second event of box $A$ is named $j.a\{b.k_{ab}.n_a.n_b\}_{ServerKey(a)}.m$ while the source event in box $J$ is named $j.a\{b.k_{ab}.n_a.n_b\}_{ServerKey(a)}.\{a.k_{ab}\}_{ServerKey_{(b)}}$. This aliasing indicates that the source and destination events have different interpretations of the same event. In this case, the initiator (modeled by box $A$) cannot read the last component $\{a.k_{ab}\}_{ServerKey_{(b)}}$ because it does not have $ServerKey_{(b)}$ so it interprets that component as simply as a sequence of bits $m$.

We can tell which event happens first in the diagram of Figure 4 by noticing that *second* event of the $B$ box happens at the same time as the *first* event of the $J$ box, so box $J$'s first event cannot start the protocol. As the diagram shows it, the first event in the protocol must be the shared first event of boxes $A$ and $B$: transmission of the message $a.n_a$ from $a$ to $b$. (It is not always necessary that a unique event in a GSPML diagram be the first event; the first event can be one of several possibilities.)

The Yahalom protocol works as follows: principal $A$ wishes to establish a session with principal $B$ and initiates a run of the protocol by sending its identity $a$ and a nonce $n_a$ to principal $B$. This is shown by the synchronization of the first event $a.b.n_a$ communicated from the $A$ box to the $B$ box in Figure 4. Box $B$ then sends $a, n_a$ and its own nonce $n_b$, encrypted under the key $ServerKey(b)$ to the server. This is show in Figure 4 by the synchronization of the second event of the $B$ box with the first event of the $J$ box. The third step of the protocol has the server (box $J$) send principal *B's* identity $b$, both nonces $n_a, n_b$, a session key $k_{ab}$, and a message $\{a.k_{ab}\}_{ServerKey(b)}$ to principal $A$. This is shown in Figure 4 by the synchronization of the second event of the $J$ box with the second event of the $A$ box, where the initiator sees the message $\{a.k_{ab}\}_{ServerKey(b)}$ simply as a bit string $m$.

In Figure 4 the end of the protocol run is shown by the $J$ box becoming the (constant) process box $STOP_{Server}$, while the $A$ and $B$ boxes become parameterized *Session* boxes that denote the start of a session between principals $A$ and $B$.

The constant box *STOP* is a process box that never performs events. It maybe considered to have an interface with visible events, but it never performs them. Thus *STOP* generates only one trace: $\langle\rangle$. *STOP* does not represent normal termination but deadlock or a blocked process. In Figure 4, the constant box is called $STOP_{Server}$ to indicate that it has exactly the same interface as the *Server* box.

## 3.2 A Complete GSPML Model

Defining a complete model of the Yahalom protocol will explain more of the GSPML language and demonstrate that it is event-based, comprehensive, concise, well-defined, and composable. Our complete model follows the Dolev-Yao structure where the intruder acts as the network connecting the principals. The complete GSPML model is shown as Figure 11 at the end of the paper, but we do not use that figure to explain GSPML. Instead, the model is presented beginning from a top-level view. Then components of the complete model are explained, proceeding from simpler constructions to more complex. This will demonstrate the abstraction capabilities of GSPML. The form of our explanation is to introduce different language elements by example. The meaning of each language element is explained first and then the protocol modeling structure is explained based on the meaning.

## 3.3 High-Level Model Structure

Figure 5 shows a top-level view of the model of Figure 11, with principals $A$ and $B$ as *abstract* concurrent boxes *UserAlice* and *UserBob*, the server as the abstract concurrent box *Jeeves*, and the intruder as abstract sequential box *Yves*. Abstract boxes have no internal regions for events or processes. This use of abstract concurrent and sequential boxes shows the high-level structure of the model without the internal details.

The basic structure of Figure 5 is a parallel box synchronizing the sequential $Yves$ box with the nested *interleaving boxes* that model $UserAlice, UserBob$ and $Jeeves$. The boxes contained in the process regions of the interleaving boxes are *interleaved*, since there is no interface port symbol on the boundary between them. The events of the boxes contained in the two process regions of an interleaving box are not synchronized. For example, if the boxes $UserBob$ and $UserAlice$ each contained an event named $a$ and both boxes performed an $a$ event, then the traces of the interleaving concurrent box containing them would include two $a$ events, not one. Even though the synchronization lines connect to the interior of both $UserAlice$ and $UserBob$, we can tell that they do not communicate directly because the synchronization lines do not go through an interface port symbol on the region boundary between them.

Figure 5 shows that none of the boxes $UserAlice, UserBob$, or $Jeeves$ communicates directly but that the three interleaved boxes are connected, via the outer parallel box, with the intruder box *Yves*. The heavier synchronization lines connecting the boxes are *abstract synchronization lines* because they do

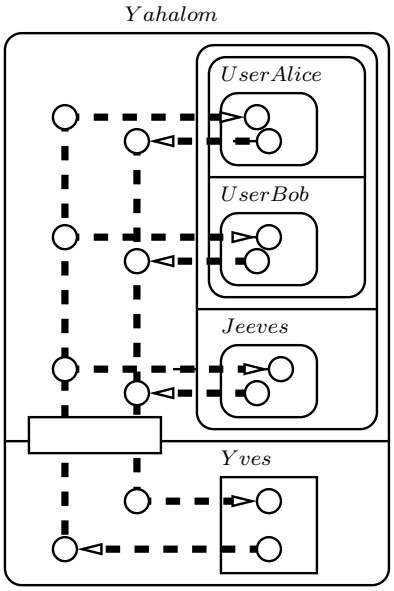


Figure 5: Yahalom: The Top-Level Structure

not connect to specific events but are capped with an event symbol. The capped termination of an abstract synch line indicates that the shared event on that end is within some greater level of detail inside the applicable box. These abstract synchronization lines are similar to the abstraction technique presented by Henderson, et al. [14] but with a different semantics. Abstract synchronization lines in a GSPML diagram, used as shown in Figure 5 have no meaning but provide a reminder of the communication pattern in the more concrete models. In a software tool these abstract synchronization lines would be place holders for the concrete lines of a more detailed view. In Figure 5 there are arrows suggesting the direction of communication, but these arrows add no meaning. In Figure 5 the abstract synchronization lines suggest that box *Yves* participates in every communication event, from any of the principals.

There are other uses of abstract synchronization lines that do have a well-defined meaning, for example as used in Figure 11 at the end of this paper. In the diagram of Figure 11, the abstract synchronization lines are used to combine concrete synchronization lines to reduce visual clutter. The distinction is easy to see because the synchronization lines in Figure 11 connect to specific events.

## 3.4 Intruder Structure

Figure 6 shows the complete structure of the intruder box *Yves*. Figure 6 is an example of an *external choice box* indicated by a square *external choice symbol* on the left end of the boundary between its process regions. An external choice box offers a choice of either of its two boxes to its environment. The first event

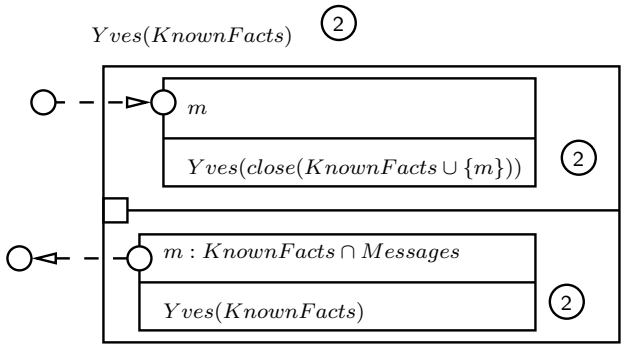


Figure 6: Yahalom: The Intruder

of the combination determines the choice of box.

In Figure 6 the events are named $m$ because the intruder may or may not be able to interpret the components of an event name. In Figure 6 this notation shows that the intruder box $Yves$ can copy and store details about any event communicated between the principals. This is shown by the *parameterized* box name $Yves(KnownFacts \cup \{m\})$. The set $KnownFacts$ models not only the events seen by $Yves$ but also any event components that $Yves$ may be able to separate and combine with components from other events. The set $KnownFacts$ also includes any event components that $Yves$ may be able to encrypt or decrypt, according to keys it already knows or learns from seeing protocol events.

The use of external choice indicates that the intruder is prepared to participate in any events that any of the three other principals offers. It can either "copy" them into $KnownFacts$ and pass them along, receive transmitted events but not relay them, or spontaneously generate bogus events that are based on the elements of $KnownFacts$.

Because the intruder's events are in distinct sequential processes, the intruder box does not have to make its traces of send events correspond to the traces of receive events it saw. The box following a receive event (incoming arrow) has the parameter $\mathsf{close}(KnownFacts \cup \{m\})$ that models the intruder accumulating facts in $KnownFacts$. The $\mathsf{close}$ function models the parsing, decrypting, encrypting and recomposing of events seen by the intruder. Definition of the $\mathsf{close}$ function is outside the scope of the GSPML language, but is represented by the parameterized box name. Use of the parameter means that there is a distinct intruder box for each possible value of the parameter.

The other part of the intruder box uses $KnownFacts \cap Messages$ to model faithful transmission as well as malicious replay and the substitution of well-formed but spurious messages by the intruder.

The meaning of the intruder's GSPML structure is that box $Yves$ must receive any event "sent" by any principal but it is not required to relay that event and may perform arbitrarily many send events before receiving an event

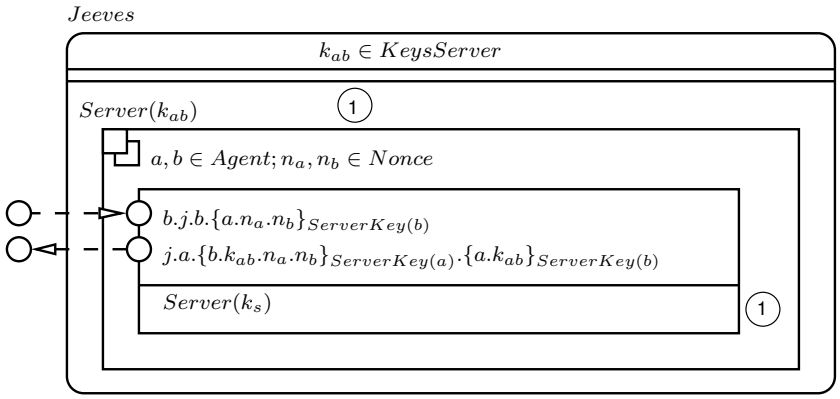


Figure 7: Yahalom: The Sever Process Jeeves

from a principal.

The sequential intruder box $Yves$ of Figure 6 is able to handle many events from many protocol runs because it is recursive. The box participates in one event of one protocol run, by its choice mechanism. After the single protocol event, it uses recursion to become another box that is prepared to make all of the same choices again. Recursion is defined by box names, rather than graphical notation. That is, the intruder box is recursive because its box name $Yves$ appears within the process regions of a box named $Yves$. (In our prototyping to date, we have found that purely visual modeling of general recursion is problematic.) GSPML uses the numbered circles to indicate boxes where recursion is intended. In Figure 6 circle number 2 indicates the recursion back to box *Yves*.

## 3.5 Server Structure

The next figure, Figure 7, shows the full definition of the server box *Jeeves*. This box demonstrates several features of GSPML that we have not seen yet, including two generalized or *indexed boxes*. The outer concurrent box is a *indexed interleaving box*. It is a concurrent interleaving box because it has no interface port on the boundary between its process regions. The double line separating the two process regions tells us that it is an indexed interleaving box. The upper process region of an indexed interleaving box has a specification for the index set and the lower process region contains a parameterized box describing the processes that are interleaved. The meaning of the box *Jeeves* is that for each possible key $k_{ab} \in KeysServer$, there is a box $Server(k_{ab})$ interleaved with all the other boxes. The index parameter $k_{ab}$ distinguishes the structural difference between each interleaved box.

In Figure 7, the parameterized box within the interleaving is itself an indexed box, an *indexed external choice box*. Because it is an indexed external choice box, box $Server(k_{ab})$ allows its environment to choose from an indexed set of

boxes. A indexed external choice box is depicted by the double square external choice symbol in its upper left corner. The index set is the double index $a, b \in Agent; n_a, n_b \in Nonce$. This double index shows that the indexed external choice box named $Server(k_{ab})$ offers a choice of boxes over all possible pairs of agents and pairs of possible nonces. The first event of box $Server(k_{ab})$ chooses one sequential box that then performs the appropriate protocol run. This particular construction defines the server as being prepared to make any well-formed run, once.

The innermost sequential box of Figure 7 is essentially the same as the server box shown in Figure 4. It gives the order of the protocol steps followed by the server in a single run, with all values fixed. The box containing this single server run uses indexed external choice, over all possible pairs of agents and all possible pairs of nonces, to define a collection of server boxes that can conduct a single run for a fixed key $k_{ab}$, with any pair of users applying any pair of nonces. This construction models the server being prepared to engage in any run it is requested to participate in. The outer indexed interleaving box models the condition that the server Jeeves may be engaged simultaneously in many protocol runs, each with a different session key, including some bogus runs initiated by the intruder Yves.

### 3.6 User Structure

The model is completed by boxes for each of the users. In order to model a protocol of this kind, each user should be able to play either role, initiator or responder. Figure 8 shows the high-level structure of a user, either *Alice* of *Bob*. Each user is composed of two boxes, one for the user's role as a protocol run initiator and one for its role as a responder. The role modeling boxes are composed into a single user, via an interleaving box, to model the possibility of that user being engaged simultaneously in several protocol runs in either role.

Within the high-level structure of a user, the model defines the initiator and responder runs over all possible combinations of principal names, session keys, and nonces. We examine the structure of the responder role first, because it is simpler. The lower part of Figure 9 shows the box for the responder role, for user Alice. The lower part of Figure 9 does not introduce any new GSPML notation. The structure of the *Responder(a)* box is similar to the structure of the server box *Jeeves* shown in Figure 7: an outer interleaving box that allows a responder to be engaged simultaneously in several runs of the protocol, each distinguished by the responder's choice of nonce $n_a$ in the second step of the protocol. One implication of this construction is that a *Responder* may be engaged in several protocol runs, each run having identical values of $a, b, n_b,$ and $k_{ab}$. While a properly implemented protocol will not do this for a legitimate run, an intruder might try it. A good protocol model will be able to reflect this and support the investigation of its consequences. The indexed external choice box that defines $Responder(a, n_a)$ within the interleaving box gives us a choice of every possible responder process, over session keys $k_{ab}$,

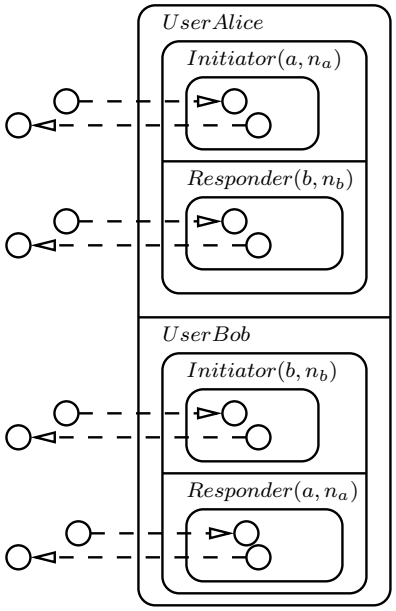


Figure 8: Yahalom: High-Level Structure of Both Users

initiator nonces $n_b$[2], and initiator names $b$. There is no recursion here, once the names, nonces, and session keys are fixed; the responder runs until a session is established, as shown by the process name $Session(b, a, k_{ab}, n_a, n_b)$ in the process region of the innermost box.

The most complex component of our complete model is the initiator role. It introduces one new GSPML construct, the *menu choice box*. Menu choice boxes offer a choice of first events, from a single box, rather than a choice of boxes. The menu choice box of Figure 9 is named $Initiator(a, n_a)$ and it is contained inside indexed interleaving box $Initiator(a)$. Menu choice is denoted by the double diamond *event choice symbol*. Above the event choice symbol there is a single event name $env?b : Agent$ that denotes a choice of event $b$ of type $Agent$, received from the environment $env$ of the box . Other than this one new box, the rest of Figure 9 uses notation already explained. Notice this initial event is not connected via a synchronization line.

The added complexity in the initiator arises because of the need to model an initiator's ability to start legitimate protocol runs entirely as a consequence of its own decision. That is, the intruder *Yves* should not be able to force any user to start a legitimate protocol run. Otherwise, the intruder either has mind control powers over the human user or has obtained control of the user's host. An intruder in either of these situations has no reason to try to break this session key establishment protocol. So the initiator has to use menu choice to

[2]When Alice is responder, the subscripts are reversed.

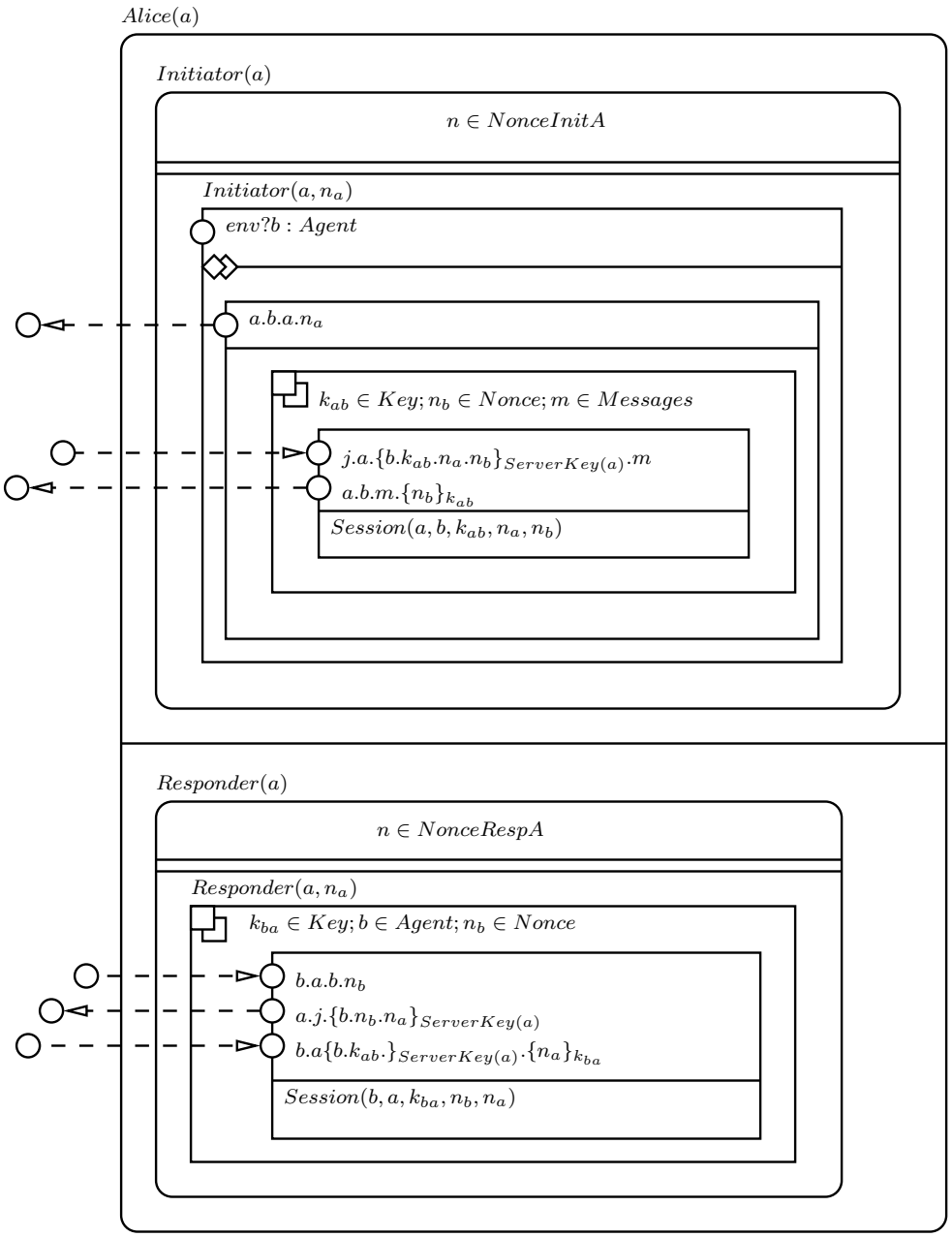


Figure 9: Yahalom: The Single User Alice

allow its environment (i.e. the human user) to chose the responder.

The outer structure of the *Initiator* box in Figure 9 is similar to the responder's structure. An interleaving box models concurrent runs of the protocol using different initiator nonces $n_a$. Within the interleaving of runs defined by possible nonces the menu choice box models the initiator's choice of responder.

Within the process region of this menu choice box that selects a user $b$ we find a simple sequential box for each possible choice of user received from channel *env*. This simple sequential box transmits the applicable nonce to the chosen user's responder. The process region of this simple sequential box uses an external choice box to select the box that finishes the initiator's part of a single run, given the nonce $n_b$ chosen and returned by the responder $b$. Once the responder has chosen a nonce $n_b$ the rest of the initiator becomes a single run via a sequential box, just like the server and responder boxes seen earlier.

## 3.7 Modeling a Non-Cryptographic Protocol

We can demonstrate the versatility of GSPML by modeling a non-cryptographic security protocol, and use this second example as an opportunity to introduce further GSPML notation. In contrast to the preceding example of the Yahalom protocol, *information flow security protocols* do not involve cryptography. Intuitively, an information flow security protocol involves a resource that is shared between two environments *High* and *Low*. The resource is supposed to provide shared service to both High and Low but prevent information from flowing from High to Low.

The problem is not as easy as it looks and is still a research topic. One of the most difficult parts of the problem is defining absence of information flow. There are simple definitions of an information-flow-secure resource shared between High and Low that are clearly effective but inhibit or preclude functionality. For example, if the allowable security protocol provides no services to the High environment, then the shared resource in question will be information flow secure. The difficulty is getting a less restrictive definition of an allowable protocol that still has acceptable information flow properties. GSPML can both define allowable information flow and model the protocols.

We now define information flow security, for a simple service protocol. The definition of information flow security is taken from Ryan and Schneider [28]. The definition is not the best proposed by Ryan and Schneider [28] but is chosen because it can show GSPML notation we have not seen yet. Readers interested in information flow security can refer to McLean [20].

The most significant difference in this example is that we are now modeling a relation between two GSPML models. In our case, the relation is *equivalence*[3] between the unnamed box of Figure 10(a) and the unnamed box on of Figure 10(b).

The new feature of GSPML used in this example is that both parts of Figure 10 use a *hiding* box. A hiding box makes events inside it invisible to its

[3]For definitions of information flow security, the specific kind of equivalence is significant, but a discussion of that would detract from our main point.

environment. A hiding box is distinguished by its *strikethrough symbols*; the strikethrough symbols indicate the events that are to be hidden in the enclosed box. Outside the hiding boxes of Figure 10 events $a_{low}$ and $b_{low}$ are visible but events $a_{high}$ and $b_{high}$ are visible. Inside the hiding box, all four events are visible when they take place.

Another difference in GSPML notation of Figure 10 is that the synchronization lines are not directed. This shows that the direction of communication or synchronization is not significant in this diagram. The synchronized events happen together in each connected box, but there is no sense of source and destination.

The box construction on the right side of Figure 10 is using synchronization with the constant box $STOP_H$ to block $H$ (i.e. High) events in the box $Protocol$. When boxes are synchronized via a parallel box but the combination reaches a point in its execution where one of them is not prepared to synchronize then the combination blocks. In this case, since box $STOP_H$ has precisely all the $H$ events of box $Protocol$ only $L$ events happen in the combination.

Figure 10 uses the process box *Protocol* to define the service itself and the two models containing copies of *Protocol* to define information flow security for the service. Essentially, the GSPML of Figure 10 defines information flow security for *Protocol* as the condition that any trace of *Protocol* with the $H$ events hidden (i.e., Figure 10(a)) is the same as any trace of *Protocol*, with its $H$ events blocked and then hidden (i.e., Figure 10(b)). The implications of this definition may be understood by considering that an arbitrary intruder box may be inserted as synchronized with the *Protocol* box inside each model of Figure 10; thus there is potential for different behavior to be visible between the two parts of Figure 10. For example, an intruder added to Figure 10 can use the external choice semantics of *Protocol* to choose the second inner box of *Protocol* (the one that does $b_{high}$) but then only request event $a_{high}$, resulting in a failure.

Adding specific details to the protocol (i.e. the *Protocol* box) is a key step in modeling information flow security. The example uses two simple events $a$ and $b$ while a more realistic example might use events like *create-channel, start-channel, stop-channel, clear-channel* and *delete-channel* for a multilevel boundary controller. Some specifications of *Protocol* will define sets of traces (and failures) that result in equality and others, sometimes surprisingly, will not. The process of designing a suitable information flow security protocol involves trade offs between the specification of *Protocol*, the protocol itself, and the pair of enclosing security definition boxes.

Figure 10 also demonstrates the compositional nature of GSPML models. If the service defined by the box named *Protocol* is to have another security property besides information flow security, then the box named *Protocol* can be removed unchanged from the hiding and blocking equivalence and placed in a model for that property.

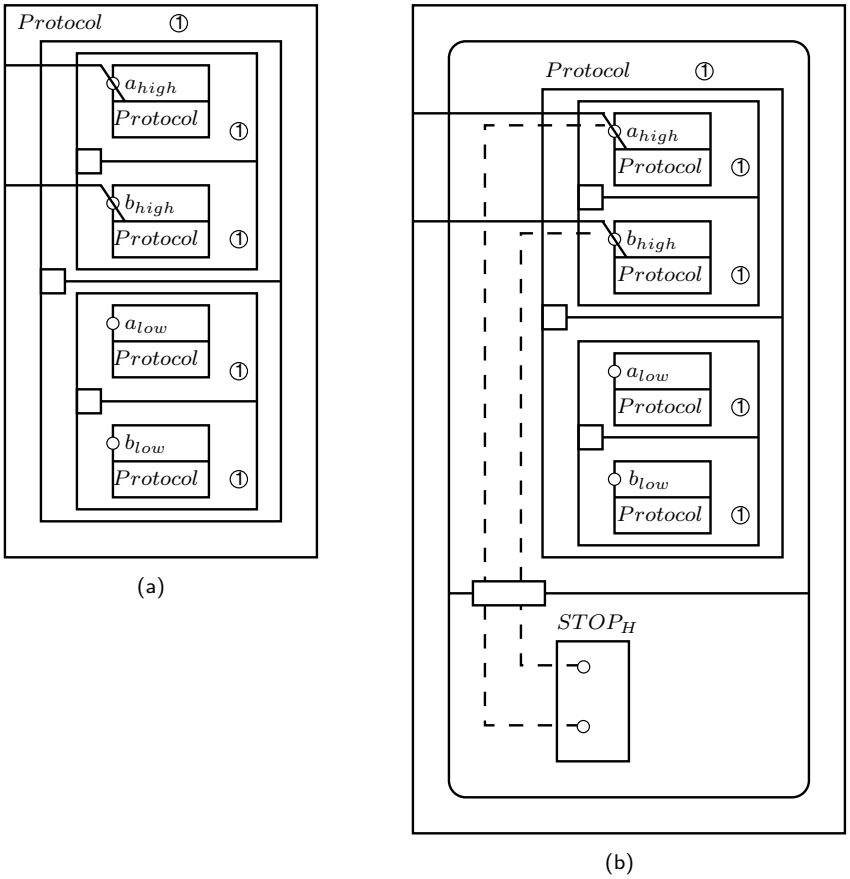


Figure 10: Modeling an Information-Flow-Security Protocol

# 4 Conclusions

Our first conclusion is that visual modeling does not magically make complex security issues simple. The two examples were chosen because they are as complex and realistic as can be presented in a brief paper. The GSPML depiction cannot remove inherent complexity from a security protocol, but it can present security protocols in a clear visual form.

Some complex concepts can be understood more quickly by visual means. Visual descriptions are sometimes preferable to text-based notation. GSPML provides those benefits for security protocols.

Our second conclusion is that GSPML is a modeling language that meets the security protocol modeling criteria: event-based, compositional, comprehensive, concise, and well-defined. There is no other visual modeling technique that satisfies all of these criteria. The current Model Driven Architecture does not provide security-specific modeling facilities and its general modeling facilities fail to satisfy one or more of the security protocol modeling criteria. There are well-defined visual formalisms outside of the UML that could be used to model security protocols: labeled transition systems, Harel's original statecharts, and Petri nets. However, each of these three is also lacking according to at least one criterion.

A comment on our second conclusion is that all of the modeling approaches considered in Sections 2.1 and 2.2 are useful and in some cases superior to GSPML, for applications other than security protocol modeling. For instance, a lack of states and other internal computational details makes GSPML less suitable for modeling object-oriented implementations. GSPML is for modeling and defining protocols visually. Other than through some visual form of the rank function approach [31], GSPML is probably not suited to verification or analysis of protocols but should be used as a front-end for a protocol analysis tool as considered in Section 2.3

Our third conclusion is that, from a visual modeling point of view, the idea of a security protocol should be generalized to any form of interaction between security principals. The proposed notation should be security or protocol specific, rather than specialized to only cryptographic protocols.

Our final conclusion regards the application of GSPML. Security protocol design and modeling is usually considered a security specialist responsibility and outside the expertise of a general software developer. Why then would we need a modeling language just for security protocols? There are three reasons: 1) security specialists benefit from visual modeling, as demonstrated by the visual components of the tools described in Section 2.3 above, 2) a visual presentation may be more useful to software developers who have to implement the security protocol and thus serve as a bridge from security specialist to other developers, 3) many security protocols fail because they are used in new or different environments; GSPML models may reveal the impact of the new environment more clearly than a text-based model.

The diagrams of GSPML are compatible with several forms of process algebra. That is, the semantic differences in various process algebras are not great

enough to require substantially different diagrams. In our experience, we have used GSPML to visually depict security protocol models of both classical CSP and PEPA stochastic process algebra semantics. It should be possible to use CCS [25] or ACP [2, 1] semantics for GSMPL diagrams. All three process algebras are Turing complete, so there is no real issue of expressiveness in one semantics over another.

Our future work on GSPML will include further prototyping and application, to validate the syntax and semantics. We will also continue to analyze the visual aspects of the language, to improve the balance [35] between language complexity and the complexity of visual models drawn in the language.

# References


[1] J. Baeten and W. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[2] J. Bergstra and J. Klop. Fixed point semantics in process algebra. Technical report, Mathematical Centre, Amsterdam, 1982.

[3] S. Brackin. A HOL extension of GNY for automatically analyzing cryptographic protocols. In *Proc. 9th IEEE Computer Security Foundations Workshop*, Kenmare, County Kerry, Ireland, 1996.

[4] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London*, (426):233–271, 1989.

[5] A. Cerone. From process algebra to visual language. Technical Report 01-36, Software Verification Research Centre, The University of Queensland, Queensland 4072, Australia, October 2001.

[6] I. Cervesato and C. Meadows. A fault-tree representation of NPATRL security requirements. In *Workshop on Issues in Theory of Security 2003*, 2003.

[7] R. Cleaveland, X. Du, and S. Smolka. GCCS: A graphical coordination language for system specification. In *4th International Conference on Coordination Models and Languages*, pages 284–298, Limassol, Cyprus, 2000.

[8] R. Cleaveland, J. Gada, P. Lewis, S. Smolka, O. Sokolsky, and S. Zhang. The Concurrency Factory: practical tools for specification, simulation, verification and implementation of concurrent systems. In G. Belloch, K. Chandy, and S. Jagannathan, editors, *Proc. DIMACS Workshop on Specification of Parallel Algorithms*. AMS, May 1994.

[9] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19, 2001.

[10] F. Fabrega, J. Herzog, and J. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7:191–230, 1999.

[11] S. Gilmore and M. Gribaudo. Graphical modelling of process algebras with DrawNET. In *Proc. Workshop on Petri Nets and Performance Models (PNPM '03)*, Urbanna, Illinois, USA, September 2-5 2003.

[12] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[13] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7), July 1997.

[14] P. Henderson, R. Walters, and S. Crouch. Implementing hierarchical features in a graphically based formal modelling language. In *Proc. 28th Int. Computer Software and Applications Conf. COMPSAC '04*, pages 92–98, Hong Kong, September 2004.

[15] G. Hilderink. A graphical modeling language for specifying concurrency based on CSP. In *Proc. Communicating Process Architectures 2002*, Reading, England, September 2002.

[16] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[17] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[18] D. Jovanovic, B. Orlic, G. Liet, and J. Broenink. gCSP: a graphical tool for designing CSP systems. In *Proc. Communicating Process Architectures 2004*, Headington, England, September 2004.

[19] R. Lichota, G Hammonds, and S. Brackin. Verifying the correctness of cryptographic protocols using Convince. In *Proc. 12th Annual Computer Security Applications Conference*, San Diego, California, USA, December 1996.

[20] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Research in Security and Privacy*, Oakland, California, USA, May 1994.

[21] C. Meadows. The NRL protocol analyzer: an overview. *The Journal of Logic Programming*, 26(2):113–131, 1996.

[22] V. Mencl. Enhancing component behavior specifications with port state machines. *Electronic Notes in Theoretical Computer Science*, 101C:129–153, 2004. Special issue: Proceedings of the Workshop on the Compositional Verifications of UML Models, CVUML, Ed. F. de Boer and M. Bonsangue.

[23] J. Millen and G. Denker. CAPSL and MuCAPSL. *Journal of Telecommunications and Information Technology*, pages 16–27, March 2002.

[24] G. Milne. *Formal Specification and Verification of Digital Systems.* McGraw-Hill, 1994.

[25] R Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice-Hall, 1989.

[26] Object Management Group. *Unified Modeling Language: Superstructure, Version 2.0*, final adopted specification ptc/03-08-02 edition, August 2003.

[27] C. Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn: Institut für Mathematik, 1962. Available as Technical Report RADC-TR-65–377, vol. 1, 1966, pages:supl. 1, English Translation.

[28] P. Ryan and S. Schneider. Process algebra and non-interference. In *Proc. 12th IEEE Computer Security Foundations Workshop*, Mordano, Italy, June 1999.

[29] P. Ryan and S. Schneider. *Modelling and Analysis of Security Protocols.* Addison-Wesley, 2001.

[30] E. Saul and A. Hutchison. Enhanced security protocol engineering through a unified multidimensional framework. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[31] S. Schneider. Verifying the correctness of authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, September 1998.

[32] B. Selic. The pragmatics of model-driven development. *IEEE Software*, pages 19–25, September/October 2003.

[33] D. Song, S. Berezin, and A. Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9:47–74, 2001.

[34] J. Tenzer and P. Stevens. Modelling recursive calls with UML state diagrams. In *Fundamental Approaches to Software Engineering 2003, LNCS 2621*, pages 135–149, Warsaw, Poland, April 2003. Springer-Verlag.

[35] E. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, Connecticut, 2001.

[36] R. Walters. Automating checking of models built using a graphically based formal modelling language. *Journal of Systems and Software*, 71(1):55–64, 2005.

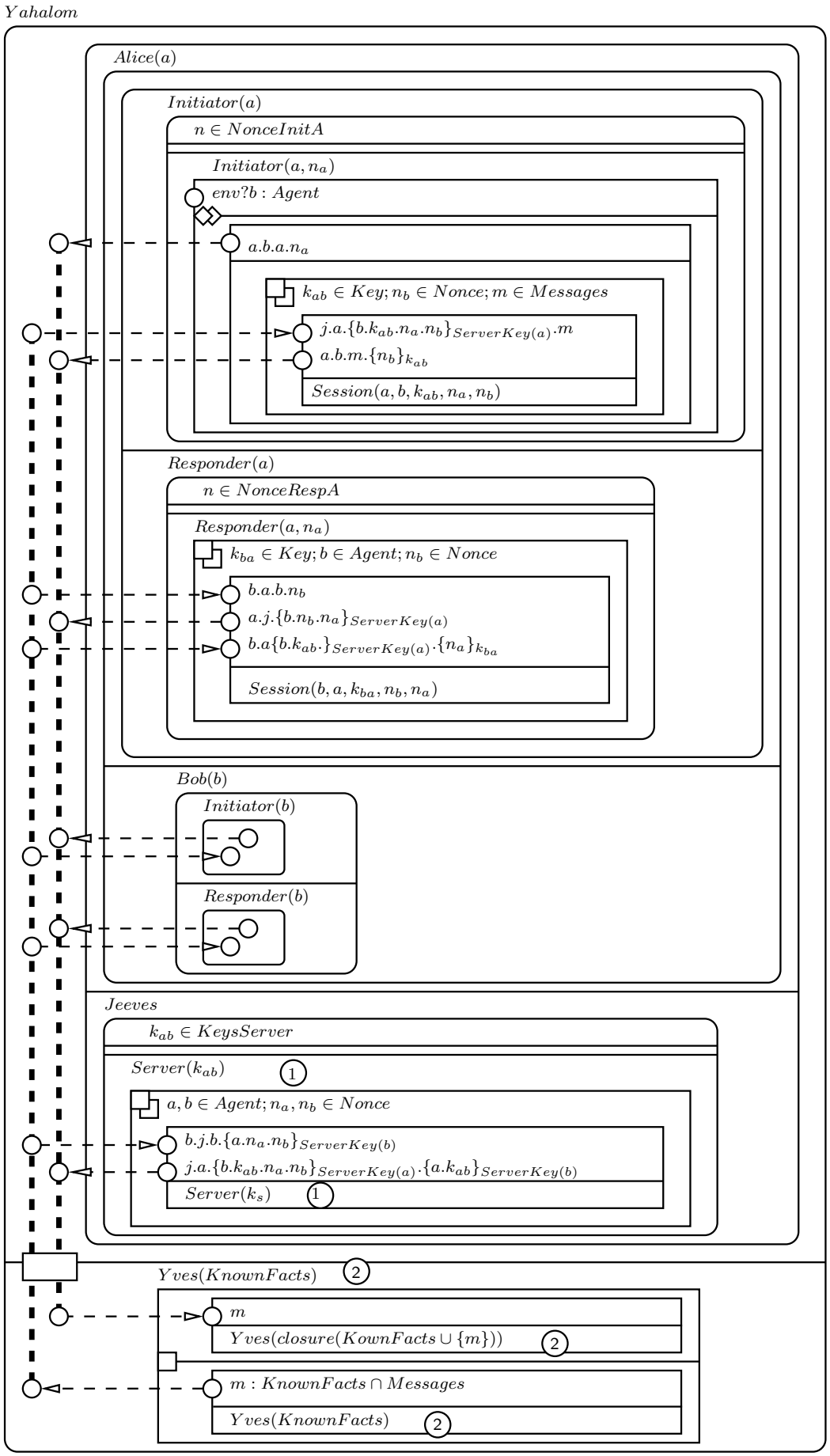


Figure 11: Yahalom: The Complete Protocol Model