Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5540--05-8919

A Formal Syntax and Semantics for the GSPML Language

JOHN P. MCDERMOTT

Center for High Assurance Computer Systems
Information Technology Division

October 31, 2005

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
31-10-2005	Memorandum Report	Jan 2005 - Aug 2005

4. TITLE AND SUBTITLE

A Formal Syntax and Semantics for the GSPML Language

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62235N

6. AUTHOR(S)

John P. McDermott

5d. PROJECT NUMBER
IT-235-007

5e. TASK NUMBER
T036-04

5f. WORK UNIT NUMBER
WU-55-6475

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory, Code 5540
4555 Overlook Avenue, SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5540--05-8919

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR / MONITOR'S ACRONYM(S)

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report formally defines both a syntax and a semantics for the GSPML visual language. The syntax is defined with a hypergraph grammar and the semantics is defined with Plotkin style structural operational semantics. A decorated trace semantics is derived from the labeled transition system of the structural operational semantics. GSPML itself is motivated by shortcomings in the visual security modeling capabilities of the Model Driven Architecture.

15. SUBJECT TERMS

Information system survivability; Security protocol; Structural operational semantics; Hypergraph grammar

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			John P. McDermott
Unclassified	Unclassified	Unclassified	UL	60	19b. TELEPHONE NUMBER *(include area code)* (202) 404-8301

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

CONTENTS

A Formal Syntax and Semantics for the GSPML Language

John McDermott
Code 5542

October 17, 2005

1 Introduction

This report formally defines both a syntax and a semantics for the GSPML visual language [1]. The syntax is defined with a hypergraph grammar [2, 3] because a hypergraph grammar suits the visual nature of GSPML. Structural operational semantics [4] has been chosen as the primary semantic approach because it carries more information about the visual constructs of GSPML. We also give a trace semantics for GSPML. GSPML itself is motivated by shortcomings in the security modeling capabilities of the model-driven architecture approach.

The concept of *symbol* can be subtle and we avoid it here by heuristic notions of *text symbols* and *diagram symbols*. Symbols represent ideas; the symbols can be abstract or concrete. Distinguishing text symbols from diagram symbols by their use as part of an alphabet does not help, as we could define an alphabet of what were intuitively diagram symbols. Definitions based on some restriction on bits used to represent text symbols is also problematic in a world of scalable, colored, mappable fonts. In the end we appeal to the intuitive notion of rectangular regions or simple geometric shapes drawn as part of a diagram being the "diagram" symbols and the "text" symbols being the ones used to name processes, events, and sets.

1.1 Motivation

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 MDA is UML 2.0 [5]. 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 visual modeling in this paradigm. This report argues that there are. Without the necessary security-specific visual modeling, model-driven approaches will produce no better security than present practice.

One of the most significant security-specific aspects of software development is the *security protocol*. Security protocols are sequences of allowable interactions between

Manuscript approved September 7, 2005.

principals. A principal is an entity that participates in a security system. Security protocols are not necessarily about cryptography; some security protocols involve no cryptography at all. A good security protocol has desirable consequences for every possible trace; not all security protocols are good ones. For this reason, we want to be able to define all possible traces in a graphical language.

The UML candidates for visual modeling of security protocols all have shortcomings. Existing alternatives outside of UML also have problems, for various reasons. Some of the difficulties are visual modeling issues and others are semantic issues. One of the most critical semantic requirements for modeling security 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 the entire security protocol is necessary for security. Another highly desirable modeling feature is event-based modeling, as opposed to state-based modeling. 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 events and traces of a protocol.

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 better supported by 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 communication patterns between processes and abstract away from details of internal computations.
- 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.

The UML candidates for visual modeling are either not well-defined or they fail to be comprehensive or concise. Visual modeling candidates outside UML are well-defined but are either state-based or fail to be concise or comprehensive. The visual interfaces to current security protocol modeling tools also do not provide a formalism that satisfies all of our criteria. These candidates are not necessarily bad but are not suited to visual security protocol modeling, according to one or more of the criteria above. We make these statements without explanation here but present a detailed justification in Section 5.

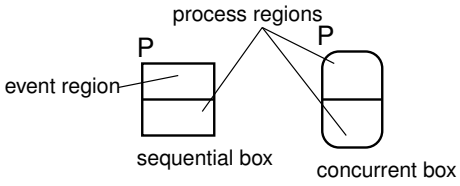

Figure 1: Basic Boxes of GSPML

2 The GSPML Alternative

The goal of GSPML 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 semantics that differ from the semantics of GSPML defined here.

The GSPML alternative is well-defined, event-based, compositional, comprehensive, and concise. GSPML semantics is based on the semantics of the Communicating Sequential Processes (CSP) process algebra. We chose CSP rather than another process algebra because CSP has been used extensively to model security protocols. CSP itself is usually explained visually by ad-hoc labeled transition system (see Sections 2 or 5.2) depictions or by "wiring diagrams" that indicate the synchronization between processes. For the reasons discussed in Section 5.2, these ad-hoc diagrams also are not suitable for visually modeling security protocols. GSPML is not intended to be a "visual CSP" as we have no basis for claiming that generality. We do mean for it to be able to model most security protocols as a single diagram, though not necessarily a diagram that fits the format of this page. Appendix C provides an example model of a complex security protocol in a single diagram.

Our presentation here in Section 2 does not define a semantics for the language but provides an introduction and demonstrates the applicability of GSPML. The formal GSPML semantics are presented in Appendix B.

2.1 Basic Boxes

In GSPML, every process is defined by either a *process box* (also referred to as a *box* when simplicity is desired) or possibly a *process box name* (*box name*). Every GSPML model is a collection of nested process boxes.

There are two major distinctions between boxes: *sequential boxes* and *concurrent boxes*, as show in Figure 1. A sequential box has rectangular corners and models sequential processes. A concurrent box has round corners and models concurrent processes. Boxes have *process regions* that may contain other GSPML boxes. Sequential boxes may also have *event regions*.

A 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 process name in it. The innermost process regions of a well-formed GSPML diagram must contain box names only (not

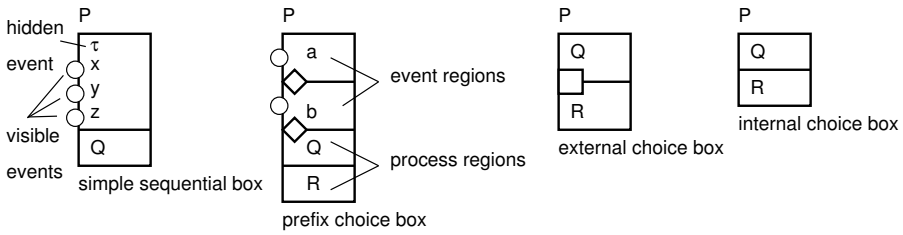

Figure 2: Sequential Boxes of GSPML

boxes), to rule out unbounded nested visual recursion. If this were not required, a box with unbounded nesting of other boxes would be considered well-formed.

Events must appear as part of some sequential box, that is, never as part of a concurrent box. Only one event in one sequential box happens at any point time, unless the event is *shared* among concurrent processes. Shared events happen simultaneously and must have the same name everywhere they happen. Events are atomic but their names may carry compound information, as indicated by a dotted notation. For example, a single event that models the transmission of a message from entity *a* to entity *b* may be named *send.a.b*.

Events may be *visible* or *hidden*. Visible events appear in the traces generated by a box and may be shared between boxes. Hidden events may influence the behavior of a box but do not appear in its traces; they represent internal behavior and may not be shared. One practical application of hidden events is modeling nondeterminism.

2.2 Sequential Boxes

A *simple sequential box*, as shown in Figure 2 is divided into two regions: an event region located above a process region. The event region of a simple sequential box defines the order of events in a sequential process. Events are listed from top to bottom, in the order they must occur. Each visible event is defined by a round *event symbol* on the left hand boundary of the event region. All events must have names listed to the right of the event symbol, inside the event region. Hidden events are represented by an event name in the event region but have no event symbol on the left boundary of the event region.

Figure 2 shows a simple sequential box that defines visible events *x, y, z* and hidden event τ, that must be ordered as $\tau \rightarrow x \rightarrow y \rightarrow z$. The process region below the event region of a simple sequential box must contain either a process name or another process box. In either case, the process identified in the process region follows the last event defined in the event region. So the simple sequential box in Figure 2 has the semantics of the CSP process $P = \tau \rightarrow x \rightarrow y \rightarrow z \rightarrow Q$.

Each time an event occurs in a GSPML model the corresponding sequential box is replaced by a box containing all of the following events. That is, in the simple sequential box of Figure 2, after the events τ and x, the box *P* is replaced with a simple sequential box having the event y above the event z in its event region and the process *Q* in its process region. When all of the events of a box's event region are consumed, the

resulting box is taken from the process region. So simple sequential box *P* eventually is replaced by box *Q*.

There are three other forms of sequential process boxes: *prefix choice boxes*, *external choice boxes*, and *internal choice boxes*, as shown in Figure 2.

A prefix choice box offers choice over initial events. The choice then defines the subsequent process. A prefix choice box comprises at least two and at most finitely many event regions located above a corresponding number of process regions. Each event region of a prefix choice box must contain only one event. Choice of event *i* results in a process that does event *i* and then acts like the process in the *i*th process region of the box. Prefix choice boxes are denoted visually by the diamond on the lower left boundary of each event region. The prefix choice box *P* shown in Figure 2 either does event *a* first and then acts like process *Q* or does event *b* and then acts like process *R*.

An external choice box offers choice over processes. An external choice box has no event regions but comprises at least two and at most finitely many process regions, that is, there are no event symbols on the left boundary of any region of an external choice box. External choice boxes have no event regions because choice is based on process names, not first events. External choice boxes are denoted visually by this absence of events and by the external choice square on the boundaries between process regions. The external choice box *P* depicted in Figure 2 either acts like process *Q* or like process *R*, as chosen by the environment of box *P*.

An internal choice box differs from an external choice box by its relationship with its environment; that is, an external choice box allows the environment to choose the process, but an internal choice box makes the choice itself, without regard for the consequences to its environment. An internal choice box is denoted visually by having only process regions just like an external choice box, but without the square symbols on its region boundaries.

2.3 Concurrent Boxes

There are two kinds of concurrent boxes: *interleaving boxes* and *parallel boxes*. A basic interleaving box is a concurrent box (round corners) with two process regions. An interleaving box denotes interleaved execution of the processes inside its process regions. That is, each process executes concurrently with no communication. At any point in time, one event from either the top region or the bottom region takes place, but never from both, unless some form of enclosing parallel box adds communication. Figure 3 shows a basic interleaving box. A basic parallel box looks like a basic interleaving box with an *interface port* symbol between its process regions. The interface port symbol means that there is a non-empty set of events that are shared between the processes inside the two process regions of the parallel box. The shared events must have the same names. Shared events are connected by *synchronization lines*. Synchronization lines may be drawn anywhere that suits visual clarity, but must pass through the interface port that defines the sharing. Figure 3 shows a parallel box with the event *y* shared between sequential boxes P_1 and P_2. The synchronization line is drawn outside of the parallel box to emphasize that it may be routed anywhere, as long as it passes

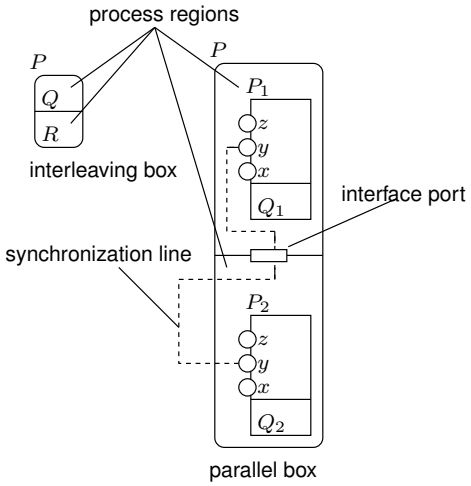

Figure 3: Concurrent Boxes of GSPML

through the corresponding interface port. In GSPML synchronization lines are only drawn because of an interface port in a parallel box.

The meaning of the parallel box *P* shown in Figure 3 is that the *x* and *z* events are distinct in each process region of parallel box *P*. That is, for example, if box P_1 does an *x* and box P_2 also does an *x* there are two *x* events in the trace of concurrent box *P*. On the other hand, if box P_2 in the bottom process region of concurrent box P does a *y* then box P_1 on the top also participates in the same *y* event: there is only one *y* in the combined trace of top-level box P.

2.4 Indexed Boxes

Prefix choice, external choice, internal choice, interleaving, and parallel boxes have an *indexed form*. *Indexed prefix choice* boxes are also referred to as prefix choice boxes, for simplicity. Their form is shown in Figure 4. The meaning of prefix choice box *P* from Figure 4 is that an initial event *x* is chosen from set *X*. After this event *x* then the box acts like the process *Q(x)* which is selected by the value of event *x*. The indexed form of a prefix choice box is denoted visually by a double diamond symbol on its event region boundary and the fact that it has only one process region.

The other forms of indexed boxes use an index to identify each box in a finite set of boxes. Each box P_j from the set is defined for $j \in J$, where J is a finite index set. An *indexed external choice box* is denoted visually by a double square symbol on the lower boundary of its upper process region and by the fact that it has only two process regions. The description of the indexing set is placed next to the double square. The lower process region contains a single GSPML box describing the indexed set of boxes. An indexed external choice box allows its environment to choose a process based on the index value. In the example indexed external choice box *P* of Figure 4, the environment chooses a value *j* out of the index set *J*. The value *j* designates the box *Q(j)* that will be

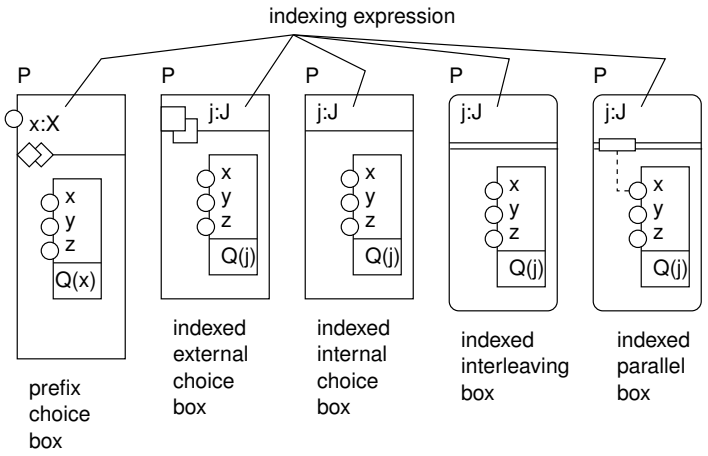

Figure 4: Indexed Boxes of GSPML

chosen. An *indexed internal choice box* acts like the indexed external choice box but the box rather than the environment chooses the value j out of the index set J. Indexed internal choice boxes differ visually from indexed external choice boxes by having no double square symbol on their process region boundary. The *indexed interleaving box*, does not offer a choice of boxes, but instead gives the modeler a way to compactly describe the interleaving of a large number of similar boxes, each distinguished by values chosen from the index set. An indexed interleaving box is denoted visually by a double line separating its two process regions. The top process region will contain the name of an index set and the bottom process region will contain a single GSPML box representing the indexed set of boxes, as shown in Figure 4. Finally, there is an *indexed parallel box* form of the parallel box. Indexed parallel boxes only have meaning when the interface port is understood to communicate the same events to all boxes in the indexed set. Any event from that interface happens in all of the boxes. An event not in the interface happens in only one of the boxes. An indexed parallel box looks like an indexed interleaving box, but has an interface port on the double line. Synchronization lines are drawn from the interface port to the single box in the lower process region. The synchronization lines may be used to indicate which events are communicated by the interface port.

3 Hypergraph Grammar Syntax

We use a hypergraph grammar to define the syntax of GSPML because its diagrams are represented syntactically as hypergraphs [2, 3]. A *hypergraph* is a generalization of a graph where the edges (called *hyperedges*) can be attached to any fixed number of nodes. A hyperedge *visits* these nodes. A hyperedge has connection points, called *tentacles*, that attach to the nodes it visits. A (GSPML) diagram component is represented by a hyperedge with each tentacle representing a specific attachment point

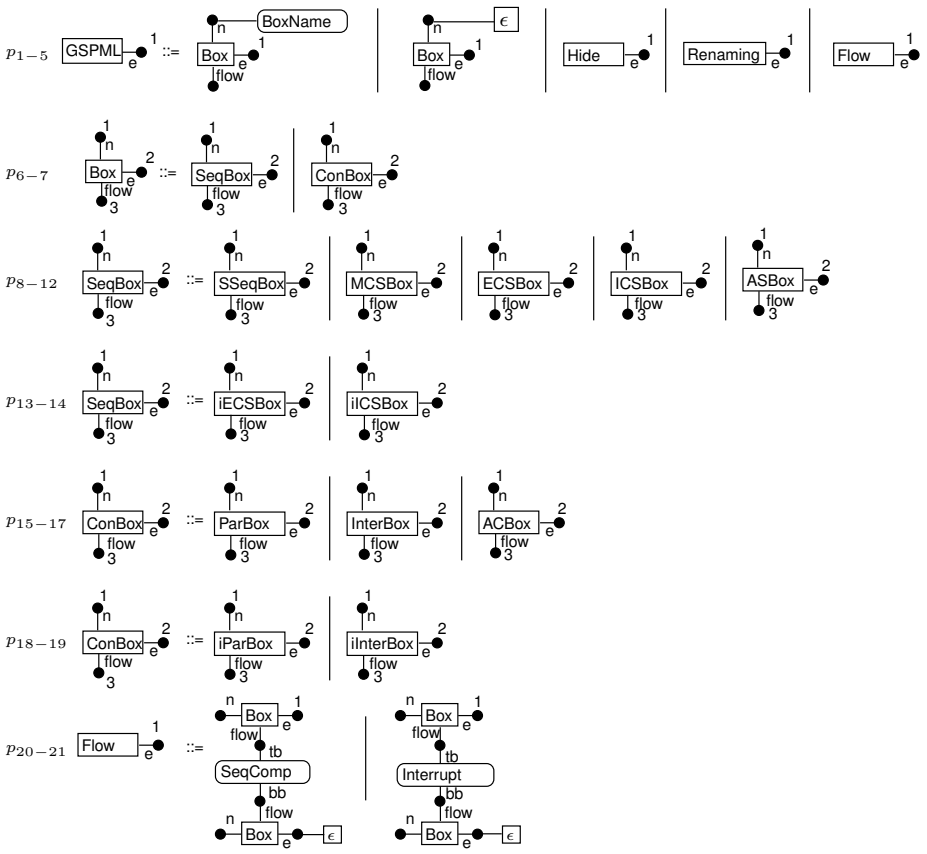

Figure 5: Top Level of the GSPML Hypergraph Grammar

for connecting diagram symbols to each other. For example, in the simple sequential box of Figure 24(a) the event region is represented by a hyperedge that has an attachment points *bottom boundary* that connects to the *top boundary* attachment point of the process region below it.

A hypergraph grammar G is a quadruple $G = (N, T, P, S)$ where the finite sets N and T contain the nonterminal and terminal hyperedge types, the finite set P contains the *hypergraph productions* of G, and S is the *starting hypergraph*. Each hypergraph production $p \in P$ of hypergraph grammar G is an algebraic hypergraph transformation rule $(L \xleftarrow{f_\ell} F \xrightarrow{f_r} R)$. Hypergraphs L, R and F of hypergraph production p are the left-hand side (lhs), right-hand side (rhs), and *interface*. Mappings f_ℓ and f_r are hypergraph morphisms, with f_ℓ being into hypergraph L, that is, interface hypergraph F is a sub-hypergraph of lhs L. Figure 5 presents the top-level productions of the GSPML hypergraph. The productions combine symbols and layout from conventional BNF grammar with visual depictions of the hypergraphs L, R and F. In each production, hypergraph L appears to the left of a conventional BNF production symbol ::=

and hypergraph R appears to its right. Alternative productions are indicated by vertical bars as in conventional BNF production rules. For example, the first line of Figure 5 shows productions p_1 through p_5 as alternatives. The interface hypergraph F is implicitly defined by the labeled nodes and the hyperedges contained in both the left side and right side of the visual depiction of the production rule. So the interface hypergraph F for production rule p_1 in Figure 5 consists of a single node labeled 1.

Nonterminals are depicted as rectangles, e.g. nonterminal *Box* in the rhs of production p_1. Terminals are shown as ovals, e.g. terminal *BoxName* in the rhs of production p_1. The interface of a production is implicitly defined by the numbered nodes that appear in both the lhs and rhs hypergraphs of a production, e.g. nodes 1 and 2 of production p_1 shown in Figure 5.

Production p is applied to hypergraph H by finding left hand side L as a subgraph of H and replacing sub-hypergraph $L \setminus F$ with sub-hypergraph $R \setminus F$. The replacement is oriented according to interface F and mapping f_r. This *step* of a derivation results in hypergraph H'. The hypergraph language $L(G)$ defined by hypergraph grammar G is the set of all hypergraphs H that contain only terminal hyperedges, and can be derived from start S in a finite number of steps.

The GSPML hypergrammar uses a special nonterminal hyperedge ϵ to indicate that no diagram component is to be attached, in a production. For example, production p_{22} shown in Figure 9 in Appendix A uses an ϵ nonterminal hyperedge at attachment point *tb* (top boundary) of the EventRegion terminal, to indicate that no diagram component is to appear above the event region, in this production. In contrast, production p_{32} of Figure 10 has the same event region terminal hyperedge attachment point *tb* associated with the interface node number 1 of the MCSC nonterminal, to show that process regions may appear above event regions in an application of menu choice convention (refer to Figure 27 for the semantics of menu choice).

4 Structural Operational Semantics

Structural operational semantics (SOS) [4] is a well-understood approach and has a significant literature. SOS is used to provide a model-theoretic interpretation for a specification language or a process algebra. Structured operational semantics generates an LTS in which the configurations are closed terms over a first-order single-sorted signature. The transitions between configurations are defined by a *transition system specification* or TSS. A TSS is a signature with a set of proof rules called *transition rules*. If the TSS transition rules have only positive premises (see Def. 5) then the LTS defined by the TSS is simply the transitions derivable from the transition rules.

None of the SOS literature address semantics for diagrams as opposed to text symbols. The approach taken here is to define the semantics of GSPML via a TSS defined on the visual box symbols of GSPML. That is, the transition rules are diagrams rather than text. A GSPML transition between LTS configurations steps from one diagram to another.

Correct depiction of a box is defined not only by the hypergraph grammar of Appendix A but also by the way boxes are drawn in the transition rules of Appendix B. For example, the question of whether to draw an event symbol as a circle, a square, or

a triangle is answered by the way the applicable boxes in Figures 24, 25, 26, and 39 are drawn in Appendix B. The visual characteristics of a diagram that are defined by the TSS transition rules are

- *Corner Shape*: The shape of box corners but not the size or aspect ratio of the box;
- *Diagram Symbols*: The shape of diagram symbols that are not boxes;
- *Text Symbols*: The meaning of text symbols associated with a box.
- *Placement*: Aspects of placement not easily described by the hypergraph grammar. For example, the placement of event symbols at the *left boundary* attachment point of an EventRegion terminal is show in Figure 24 and cannot be understood as such directly from the hypergraph grammar.

An example of corner shape is that sequential boxes must have rectangular corners. An example of a diagram shape rule is that event symbols must be circles. An example of a placement rule is that box names must be placed outside their corresponding box, at the upper left corner.

We will point out these characteristics as they are defined by the transition rules. Only changes or new information will be presented with succeeding rules. That is, if a visual characteristic is not mentioned in a rule, then that characteristic must have been defined in an earlier transition rule.

At this point the relative sizes of the visual elements have been left undefined. The definitions do include some recommendations in terms of the em space and ex space of the font used for text symbols. In this report the term *near*, when used to define GSPML diagrams, can be taken to mean a distance between one and three em spaces.

4.1 TSS Definitions

We begin our TSS descriptions with the necessary definitions. All of them are fairly conventional for the TSS of a positive process algebra. They are stated here for completeness.

Definition 1 (Box) *A* box *is a rectangular GSPML diagram that denotes a process. We denote an unspecified box by the text symbol* $\Box$*. GSPML boxes can have* names*. We denote the specific box named* P *with first event* a *as* $\overset{P}{a\Box}$*.*

Intuitively, boxes act as the configurations of our labeled transition system, as explained in Definition 2. In a GSPML diagram, box names are denoted by text symbols near the upper left corner of the box, but outside the box. This is partially defined by production p_1 shown in Figure 8 of our hypergrammar, but the placement and textual nature of the BoxName terminal are defined by the diagrams used in the LTS of Appendix B.

Definition 2 (Labeled Transition System) *A (GSPML)* labeled transition system (LTS) *is a triple* $(\mathcal{B}, A, \rightrightarrows)$ *where*

- *$\mathcal{B}$ is a set of boxes that act as the configurations,*
- *A is a set of events,*
- *$\rightrightarrows$ is a set of binary relations $\stackrel{a}{\rightarrow}$ of the form $\stackrel{a}{\rightarrow} \subseteq \mathcal{B} \times \mathcal{B}$.*

Visible events from set A are denoted in lower case; the letter τ is used to denote hidden events. The letter μ is used to denote an event of unspecified visibility.

This report follows the usual practice of writing $\langle \stackrel{P}{\Box}, \stackrel{P'}{\Box} \rangle \in \stackrel{a}{\rightarrow}$ as $\stackrel{P}{\Box} \stackrel{a}{\rightarrow} \stackrel{P'}{\Box}$. These binary relations are the *transitions* of our LTS. We also use the notation $\stackrel{P}{\Box} \stackrel{a}{\not\rightarrow}$ to state that there is no box $\stackrel{P'}{\Box}$ such that $\stackrel{P}{\Box} \stackrel{a}{\rightarrow} \stackrel{P'}{\Box}$. Notice that by this definition a transition changes one GSPML diagram to another GSPML diagram. Intuitively, each transition results in a new drawing.

The GSPML LTS is *finitely branching*: for every box $\stackrel{P}{\Box}$ there are only finitely many transitions $\stackrel{P}{\Box} \stackrel{a}{\rightarrow} \stackrel{P'}{\Box}$ out of box $\stackrel{P}{\Box}$. The GSPML LTS is *regular*: it is finitely branching and each box can only transition into finitely many other boxes. The GSPML LTS is *finite*: it is regular and there is no infinite sequence of transitions $\stackrel{P_0}{\Box} \stackrel{a_0}{\rightarrow} \stackrel{P_1}{\Box} \stackrel{a_1}{\rightarrow} \ldots$.

Definition 3 (Signature) *Let V be a countably infinite set of variables, ranged over by x, y, z. Then a* signature Σ *is a set of* function symbols *disjoint from V, together with an* arity *mapping that assigns a natural number $ar(f)$ to each function symbol f.*

The arity of a function represents the number of arguments it has. A function symbol of arity zero is a *constant*, function symbols of arity one are called *unary*, and function symbols of arity two are called *binary*.

Definition 4 (Term) *The set $\mathbb{T}(\Sigma)$ of* open terms *over a signature Σ, ranged over by t, u, is the least set such that:*

- *each $x \in V$ is a term;*
- *if f is a function symbol and $t_1, t_2, \ldots, t_{ar(f)}$ are terms then $f(t_1, t_2, \ldots, t_{ar(f)})$ is a term.*

$T(\Sigma)$ denotes the set of ground terms *over Σ. Ground terms do not contain variables.*

A *substitution* is a mapping $s : V \rightarrow \mathbb{T}(\Sigma)$. A substitution is closed if it maps each variable in V to a ground term in $T(\Sigma)$.

Definition 5 (Transition System Specification (TSS)) *Let Σ be a term algebra signature, and let $\stackrel{P}{\Box}$ and $\stackrel{Q}{\Box}$ range over $\mathbb{T}(\Sigma)$. A* transition rule *ρ is of the form H/α, with H a set of* premises $\stackrel{P}{\Box} \stackrel{a}{\rightarrow} \stackrel{Q}{\Box}$ *and α the* conclusion *of the form $\stackrel{P}{\Box} \stackrel{a}{\rightarrow} \stackrel{Q}{\Box}$. A* transition system specification *is a set of transition rules.*

The left-hand side of a transition rule ρ is called the *source of* ρ and the right-hand side of ρ is called the *target of* ρ. A transition rule ρ is closed if it does not contain variables. Transition rules are often written in the form $\frac{H}{\alpha}$; they are drawn in the same style in Appendix B. The TSS in this report is *positive*; it contains only positive transition rules. Positive transition rules do not contain premises of the form $\overset{P}{\Box} \overset{a}{\not\rightarrow}$, i.e. claiming that there are no a-transitions out of box $\overset{P}{\Box}$.

Definition 6 (Literal) *A* literal *is a transition* $\overset{P}{\Box} \overset{a}{\rightarrow} \overset{P'}{\Box}$ *where* P *and* P' *range over the set of closed terms* $T(\Sigma)$.

Definition 7 (Proof) *Let T be a TSS. A* proof *of a closed transition rule* H/α *from T is a well-founded, upwardly branching tree whose nodes are labeled by literals, such that*

- *the root of the tree is* α.
- *if B is the set of labels of nodes directly above a node with label* ℓ *then either*
  - *B is empty and* $\ell \in H$*, or*
  - K/ℓ *is a closed substitution instance of a transition rule in T.*

If a proof of H/α exists then H/α is *provable* from T. This is usually written as $T \vdash H/\alpha$. The meaning of the GSPML transition system specification T of Appendix B) is defined as the LTS made up of all provable transitions defined on the rules of T.

Behavioral equivalence is a critical part of the meaning of GSPML. We need to be able to know or say if two boxes define the same behavior. There is a rich body of literature defining behavioral equivalence for conventional LTS's and it applies to GSPML's LTS semantics. To demonstrate this, we present, for GSPML, two of the most important forms of LTS equivalence: bisimulation [6, 7] and ready simulation [8, 9].

Definition 8 (Bisimulation) *A binary relation* $\mathcal{R}$ *on the boxes of an LTS is a* simulation *if whenever* $\overset{P_1}{\Box}\ \mathcal{R}\ \overset{Q_1}{\Box}$ *and* $\overset{P_1}{\Box} \overset{a}{\rightarrow} \overset{P_2}{\Box}$ *then there is also a transition* $\overset{Q_1}{\Box} \overset{a}{\rightarrow} \overset{Q_2}{\Box}$ *such that* $\overset{P_2}{\Box}\ \mathcal{R}\ \overset{Q_2}{\Box}$*. If simulation* $\mathcal{R}$ *is symmetric then it is a* bisimulation.

Two boxes $\overset{P}{\Box}$ and $\overset{Q}{\Box}$ are *bisimilar* if $\overset{P}{\Box}\ \mathcal{R}\ \overset{Q}{\Box}$; we denote this *bisimulation equivalence* as $\overset{P}{\Box} \Leftrightarrow \overset{Q}{\Box}$.

Definition 9 (Ready Simulation) *A simulation* $\mathcal{R}$ *on the boxes of an LTS is a* ready simulation *if whenever* $\overset{P}{\Box}\ \mathcal{R}\ \overset{Q}{\Box}$ *then if* $\overset{P}{\Box} \overset{a}{\not\rightarrow}$ *we also have* $\overset{Q}{\Box} \overset{a}{\not\rightarrow}$.

Trace semantics are an alternative to LTS semantics. Instead of defining processes as labeled transition systems, trace semantics defines processes strictly in terms of their external properties: the sequences of actions a process is prepared to perform or refuse. Trace semantics is the usual semantics for CSP, our chosen basis for GSPML, but trace

semantics do not directly show the evolution of a box from one form to another. For this reason, we chose to formally define GSPML boxes as LTS's.

It is also possible to define trace semantics from LTS semantics [10]. Because of this, we can define a GSPML trace semantics from our box-based LTS.

Definition 10 (Trace) *Given a (GSPML) LTS, a sequence* t *of its visible events*

$$t = a_0 \dots a_n, \qquad n \in \mathbb{N}$$

is a trace *of box* $\overset{P_0}{\square}$ *if there exist boxes* $\overset{P_0}{\square}, \dots, \overset{P_n}{\square}$ *such that* $\overset{P_0}{\square} \overset{a_1}{\rightarrow} \overset{P_1}{\square} \overset{a_2}{\rightarrow} \dots \overset{a_n}{\rightarrow} \overset{P_n}{\square}$. *This can be written more compactly as* $\overset{P_0}{\square} \overset{t}{\rightarrow} \overset{P_n}{\square}$. *For a box* $\overset{P}{\square}$ *we define*

$$\mathsf{initials}(\overset{P}{\square}) = \{a \in A | \exists \overset{Q}{\square} \in \mathcal{B} : \overset{P}{\square} \overset{a}{\rightarrow} \overset{Q}{\square}\}$$

Traces alone can be used to define equivalence between boxes and this is sometimes preferred for security protocol modeling. There are finer definitions of trace equivalence and these apply to GSPML semantics. The initials $\mathsf{initials}(\overset{P}{\square})$ of boxes $\overset{P}{\square}$ can be used to define a *decorated traces system* or DTS [10] that provides a more precise form of equivalence.

Definition 11 (Decorated Traces System) *A* decorated traces system *is a triple* $(A, \mathcal{T}, f)$ *where*

- A *is a set of events.*
- $\mathcal{T}$ *is a set of traces.*
- $f : \mathcal{T} \rightarrow 2^A$ *is a function from traces to sets of actions.*

Function f is used to "decorate" each trace with a set of pertinent events. Different notions of decoration define distinct forms of process equivalence.

Like labeled transition equivalences, there is also a rich body of literature defining decorated trace equivalences; for example see the results of Bloom et al. [11]. To connect these results to GSPML, we present one important form of decoration here, *stable failures*. Stable failures not only account for possible deadlock or other forms of error they also deal with impact of hidden events and thus nondeterminism:

Definition 12 (Stable Failure) *A box* $\overset{Q}{\square}$ *is* stable *if* $\overset{Q}{\square} \overset{\tau}{\not\rightarrow}$*; intuitively box* $\overset{Q}{\square}$ *does not have the potential to change its initials by participating in a hidden event* τ*. Event set* X *is a* refusal *of stable box* $\overset{Q}{\square}$ *if* $\overset{Q}{\square} \overset{\mu}{\not\rightarrow}$*, for any event* $\mu \in A$*. Event set* X *is a refusal of box* $\overset{P}{\square}$ *if the empty trace* $\langle\rangle$ *leads box* $\overset{P}{\square}$ *to some stable box* $\overset{Q}{\square}$ *that has event set* X *as its refusal. A pair* (t, X) *with trace* $t \in \mathcal{T}$ *and event set* $X \subseteq A$ *is a* stable failure *of box* $\overset{P}{\square}$ *if*

- *there is a stable box* $\overset{Q}{\square}$ *such that trace* t *leads box* $\overset{P}{\square}$ *to stable box* $\overset{Q}{\square}$*, i.e.* $\overset{P}{\square} \overset{t}{\Rightarrow} \overset{Q}{\square}$*,*
- *event set* X *is a refusal of box* $\overset{P}{\square}$*.*

4.2 Conventions

To permit clearer drawings, GSPML includes the application of certain *conventions*. Conventions define shortcuts or simplified ways of drawing a GSPML box. A convention can be defined in terms of the structural operational semantics presented here, but visually it is simpler. An example convention is that GSPML allows a list of events to be drawn together in the event region of a sequential box while the meaning is only defined in terms of boxes with a single (first) event (see Figure 25 in Appendix B). The vertical list of events can be understood as nesting of single-event boxes.

5 Related Work

In this section we present a detailed application of our criteria: *event-based, composable, comprehensive, concise,* and *well-defined*, to other candidates for visual modeling of security protocols. Using the criteria, 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.

5.1 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 life-lines 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* [12]. 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 (Basic Process Algebra) process algebra of Bergstra

and Klop [13] 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 [14] of Harel's elegant *statechart* [15] 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 events, 2) the semantics are run-to-completion which poses problems for modeling some forms of recursion (Tenzer and Stevens [16] provide good examples of this), and 3) some of the events are not atomic [17]. 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.

5.2 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 Γ 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" [15]. Labeled transition systems are not concise. Current LTS work is turning to algebraic treatments to overcome this difficulty. *Petri nets* were developed by Carl Petri [18] for formal modeling of concurrency, nondeterminism, and communication. Petri nets are well-defined 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 [17], 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 protocols. 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 [19] 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) [20] as part of the Concurrency Factory tool [21]. The GCCS visual notation is based on Milner's CCS [22] 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 [23], a modeling technique based on combinations of the CCS, CSP [24], and Circal [25] 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 [26] extended the *DrawNET* tool to model the PEPA [27]

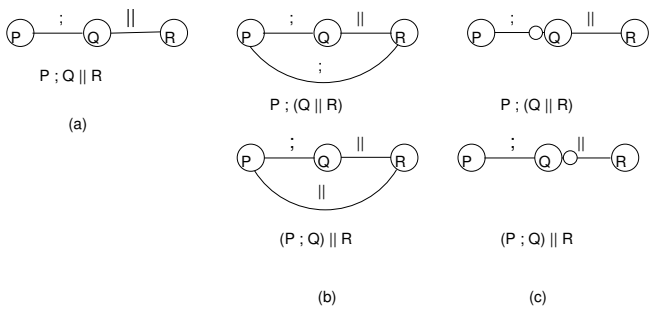

Figure 6: Resolving Compositional Ambiguity in gCSP

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. [28, 29] 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 difficult to present complex process algebra relationships graphically. Figures 6 and 7 illustrate this difficulty in the gCSP notation. Figure 6 shows the process algebra fragment $P; Q \| R$ which is an ambiguous term specifying the sequential (via the ; operator) and parallel (via the $\parallel$ operator) composition of processes P, Q and R. Figure 6 (a) shows how this ambiguity can be drawn in gCSP. Figure 6 (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 6 (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 7. However, it is the lack of explicit events that renders gCSP unsuitable for security protocol modeling.

5.3 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

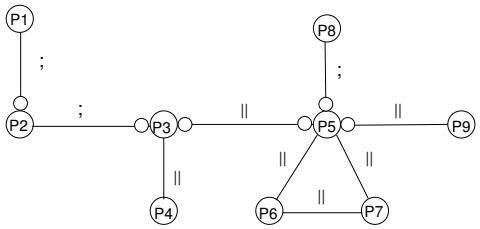

Figure 7: Complex Composition in gCSP

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 Mu-CAPSL, its group multicast protocol version, is a formal language for specifying cryptographic security protocols [31]. 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 [32]. Convince uses a text-based formal language based on BGNY [33] 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) [34]. 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 [35]. 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.

5.4 UML-Based Security Modeling

Some work has been done on security modeling with UML. Epstein and Sandhu [36] show how UML can be used to model RBAC policies. Jürjens [37] has proposed *UMLsec* as a means of annotating UML with stereotypes and tagged values, to specify security requirements. Basin, Doser, and Lodderstedt [38] have extended UML, via stereotypes, to *SecureUML*. The SecureUML language can be used to specify access control requirements on UML Class Diagrams and UML Statemachines. None of this work covers security protocol modeling. Nevertheless, it supports our observation that bare UML does not treat security issues adequately.

6 Conclusions

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

Our first 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 our criteria. There are well-defined visual formalisms outside of the MDA/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 in at least one criterion.

A comment on our first conclusion is that all of the modeling approaches considered in Sections 5.1 and 5.2 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 [39], GSPML is not suited to verification or analysis of protocols but should be used as part of a protocol analysis tool as considered in Section 5.3

Our second 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 5.3, 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 specialist to generalist, 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.

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 [40] between language complexity and the complexity of visual models drawn in the language.

References

[1] J. McDermott, "Visual security protocol modeling," in *Proc. New Security Paradigms Workshop*, Lake Arrowhead, California, USA, September 2005.

[2] M. Minas, "Hypergraphs as a uniform diagram representation model," *Lecture Notes in Computer Science*, vol. 1764, pp. 218–295, 1998, selected paper from the 6th International Workshop on Theory and Application of Graph Transformations.

[3] ——, *Diagrammatic Representation and Reasoning*. Springer-Verlag, 2001, ch. Specifying diagram languages by means of hypergraph grammars.

[4] G. Plotkin, "A structural approach to operational semantics," Computer Science Department, Aarhus University, Tech. Rep. Report DAIMI FN-19, 1981.

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

[6] R. Milner, "Calculi for synchrony and asynchrony," *Theoretical Computer Science*, vol. 25, pp. 267–310, 1983.

[7] D. Park, "Concurrency and automata on infinite sequences," in *LNCS*, P. Deussen, Ed. Springer-Verlag, 1981, vol. 104, pp. 167–183, from Fifth GI Conference, Karlsruhe, Germany.

[8] B. Bloom, S. Istrail, and A. Meyer, "Bisimulation can't be traced," *JACM*, vol. 42, pp. 232–268, 1995.

[9] K. Larsen and A. Skou, "Bisimulation through probablistic testing," *Information and Computation*, vol. 94, 1991.

[10] R. Eshuis and M. Fokkinga, "Comparing refinements for failure and bisimulation semantics," *Fundamenta Informaticae*, vol. 52, no. 4, pp. 297–3231, 2002.

[11] B. Bloom and R. van Glabbeek, "Precongruence formats for decorated trace semantics," *ACM Transactions on Computational Logic*, vol. 5, no. 1, pp. 26–78, January 2004.

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

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

[14] D. Harel and E. Gery, "Executable object modeling with statecharts," *IEEE Computer*, vol. 30, no. 7, July 1997.

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

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

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

[18] C. Petri, "Kommunikation mit automaten," Ph.D. dissertation, Bonn: Institut für Mathematik, 1962, available as Technical Report RADC-TR-65–377, vol. 1, 1966, pages:supl. 1, English Translation.

[19] R. Walters, "Automating checking of models built using a graphically based formal modelling language," *Journal of Systems and Software*, vol. 71, no. 1, pp. 55–64, 2005.

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

[21] 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 *Proc. DIMACS Workshop on Specification of Parallel Algorithms*, G. Belloch, K. Chandy, and S. Jagannathan, Eds. AMS, May 1994.

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

[23] A. Cerone, "From process algebra to visual language," Software Verification Research Centre, The University of Queensland, Queensland 4072, Australia, Tech. Rep. 01-36, October 2001.

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

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

[26] 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.

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

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

[29] 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.

[30] E. Saul and A. Hutchison, "Enhanced security protocol engineering through a unified multidimensional framework," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, January 2003.

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

[32] 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.

[33] 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.

[34] C. Meadows, "The NRL Protocol Analyzer: an overview," *The Journal of Logic Programming*, vol. 26, no. 2, pp. 113–131, 1996.

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

[36] P. Epstein and R. Sandhu, "Towards a UML based approach to role engineering," in *Proc. Fourth ACM Workshop on Role-Based Access Control*, Fairfax, Virginia, USA, October 1999.

[37] J. Jürjens, "UMLsec: extending uml for secure systems development," in *Proc. UML 2002*, Dresden, Germany, September 2002.

[38] D. Basin, J. Doser, and T. Lodderstedt, "Model driven security for process-oriented systems," in *Proc. Eighth ACM Symposium on Access Control Models and Technologies*, Como, Italy, June 2003.

[39] S. Schneider, "Verifying the correctness of authentication protocols in CSP," *IEEE Transactions on Software Engineering*, vol. 24, no. 9, pp. 741–758, September 1998.

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

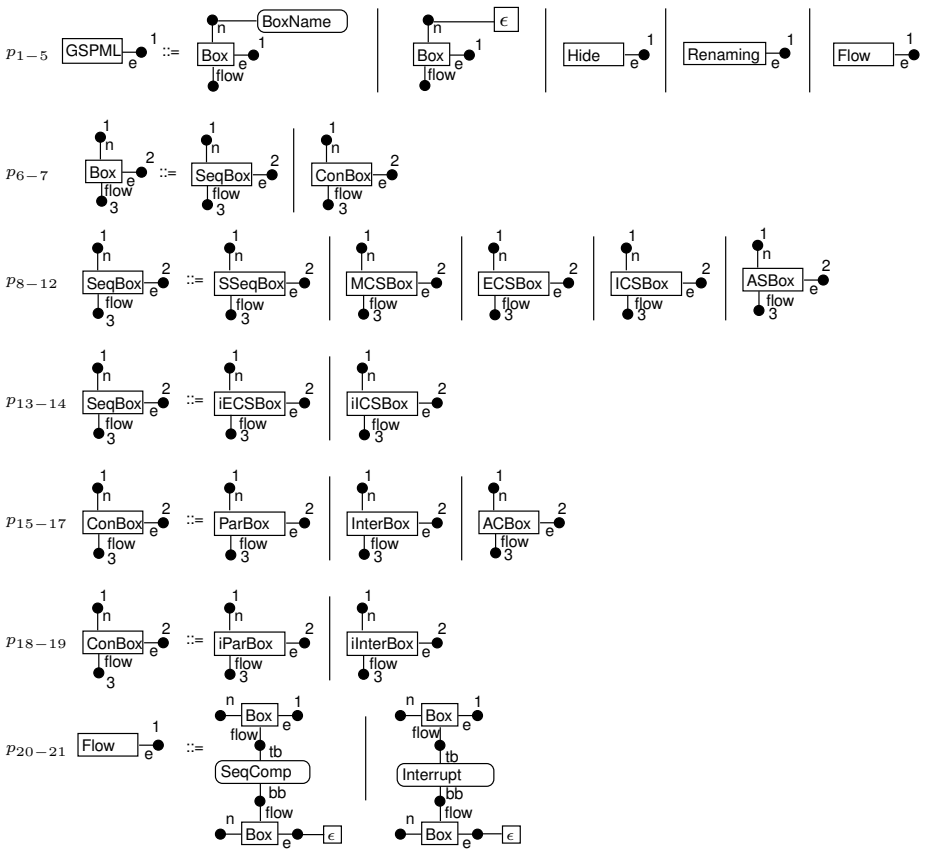

Figure 8: Top Level of the GSPML Hypergraph Grammar

A The GSPML Hypergraph Grammar Syntax

The syntax of GSPML is defined by the hypergrammar presented in this section. The syntax is given for diagrams that include the conventions (see Section 4) that are not part of the semantic definition of the language. Derivation of the simpler syntax for semantic definition is straightforward, since we only need to delete productions.

There are several intentional omissions in the grammar, to keep the size presentable for this report. The omitted productions would need to be added to a parser that used this grammar to automatically check the syntax of a model.

The syntax does not provide for hidden events. To provide for hidden events we can add another terminal *HiddenEvent* that acts as a placeholder. That is, space for the hidden event is allocated on the left boundary of the applicable event region, but no event symbol is shown. This leaves space for the event name, which will always be τ, in the list of events while allowing us to parse diagrams that contain hidden events. For each production that uses the *Event* terminal, we add another production that has the *HiddenEvent* terminal in place of the *Event* terminal.

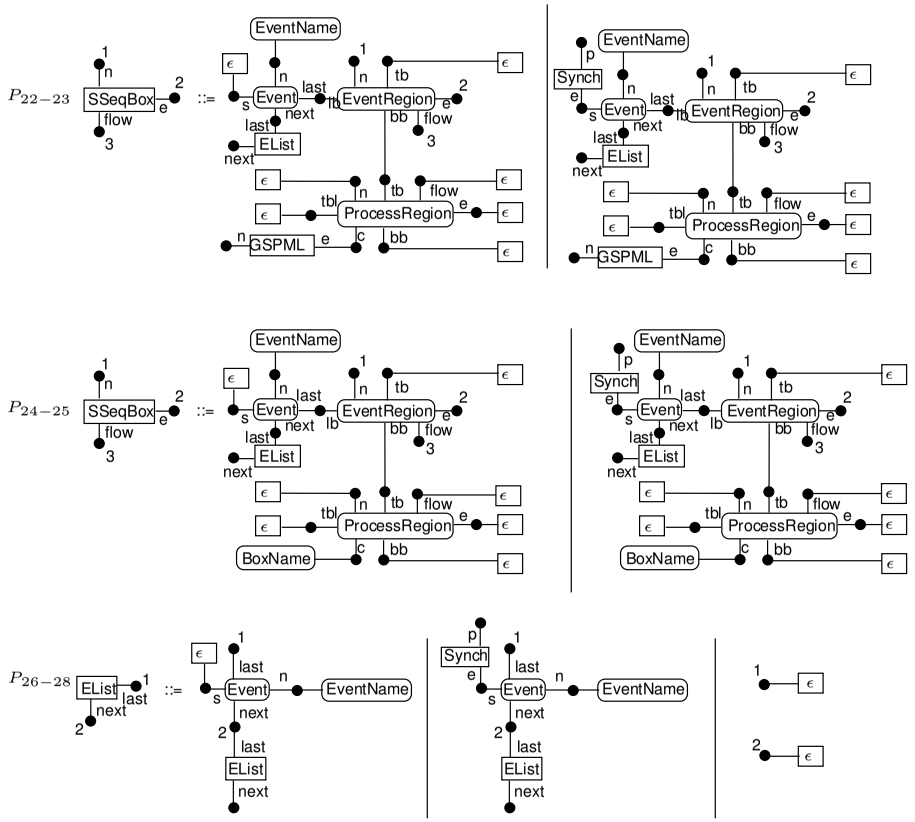

Figure 9: Syntax of the Simple Sequential Box

Attachment Point	Attaches
n	A *name* to a GSPML box or region.
e	A box or region to its containing *environment*.
c	The *contents* of an event or process region.
lb	An event or list of events to the *left boundary* of an event region.
tb	A process or event region to a process or event region's *top boundary*.
bb	A process or event region to a process or event region's *bottom boundary*.
tbl	A diamond or square symbol to the *top left boundary* of a process region.
s	A *synchronization line* to an event symbol.
r	A *interface port symbol* to a process region boundary.

Table 1: Attachment Points Used in the Grammar

No syntax is provided for backward renaming. The syntax of backward renaming is essentially the same as for forward renaming. The differences between the two are semantic.

Only half of the necessary productions for sequential composition and interrupt are show in Figures 9 through 20. The productions shown only provide for connections on the top boundary.

The following table, Table 1, describes the most important attachment points used in the grammar.

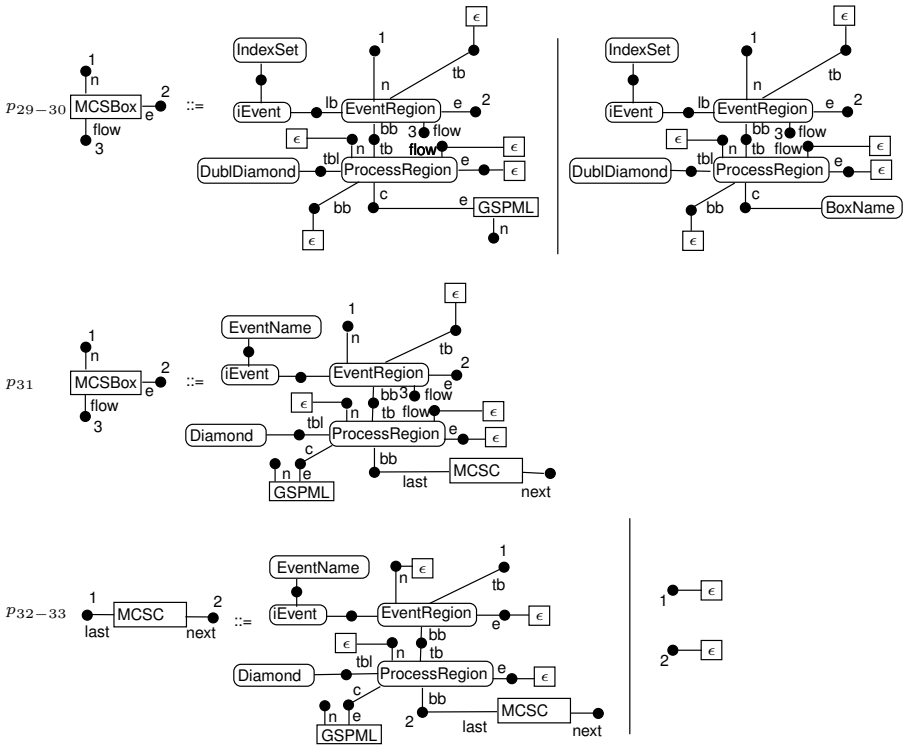

Figure 10: Syntax of the Menu Choice Sequential Box

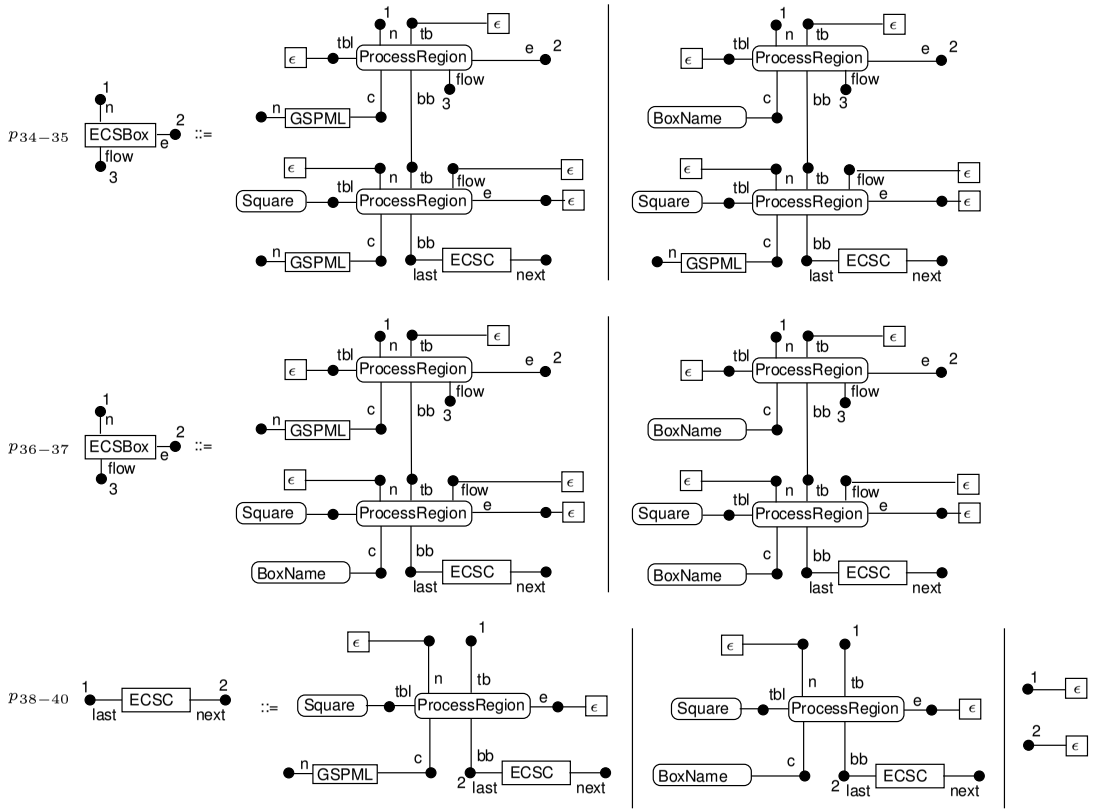

Figure 11: Syntax of the External Choice Sequential Box

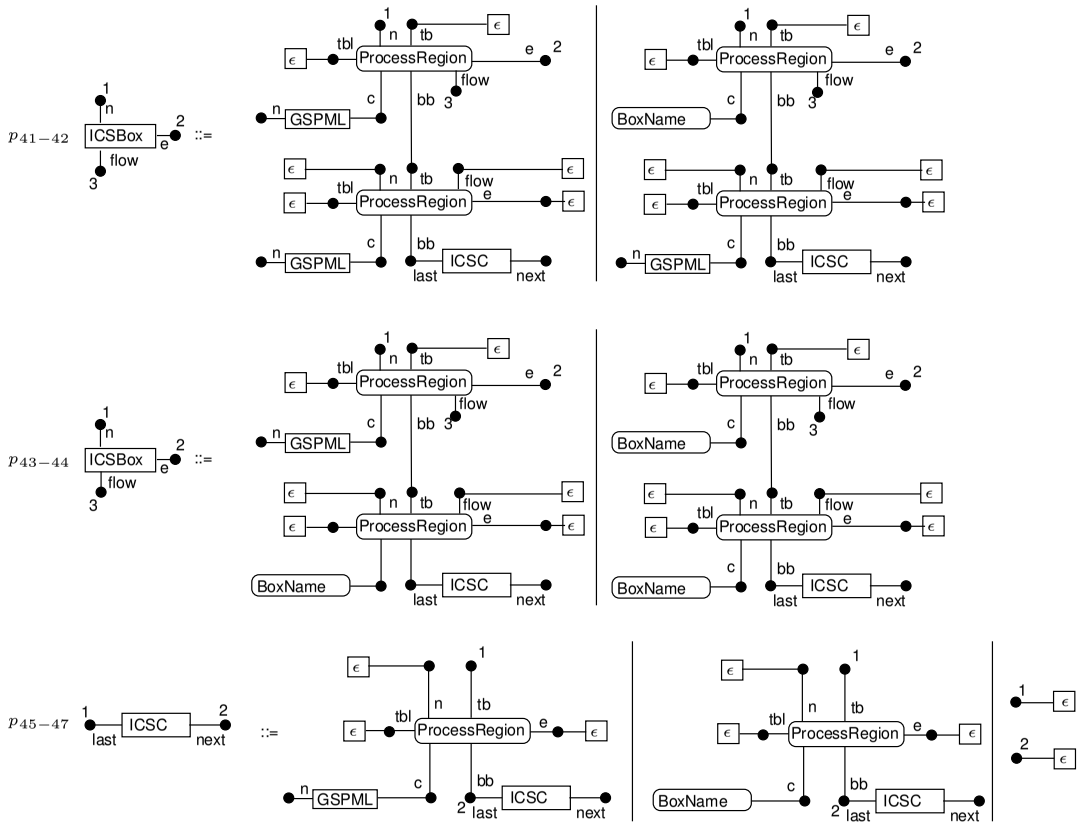

Figure 12: Syntax of the Internal Choice Sequential Box

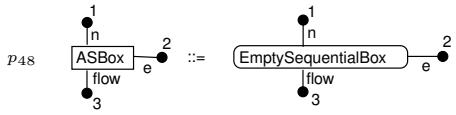

Figure 13: Syntax of the Abstract Sequential Box

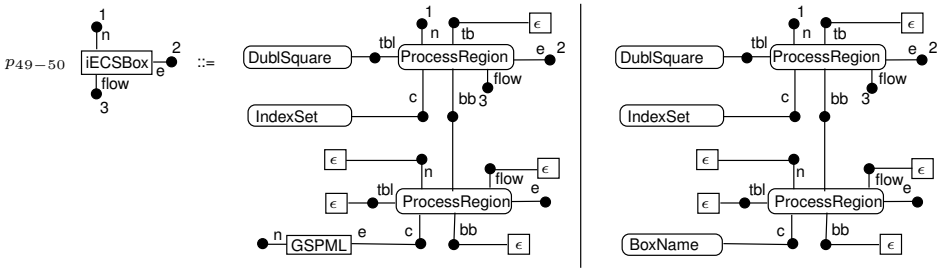

Figure 14: Syntax of the Indexed External Choice Sequential Box

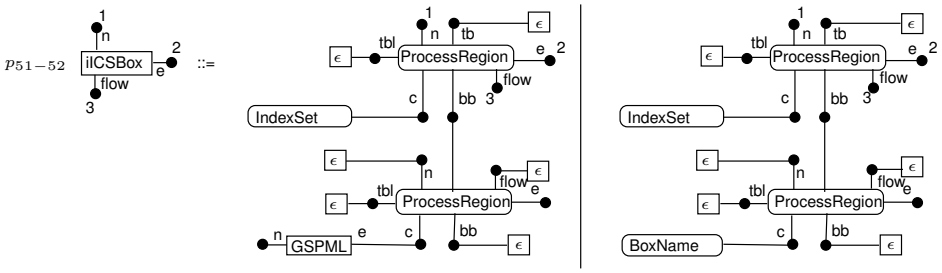

Figure 15: Syntax of the Indexed Internal Choice Sequential Box

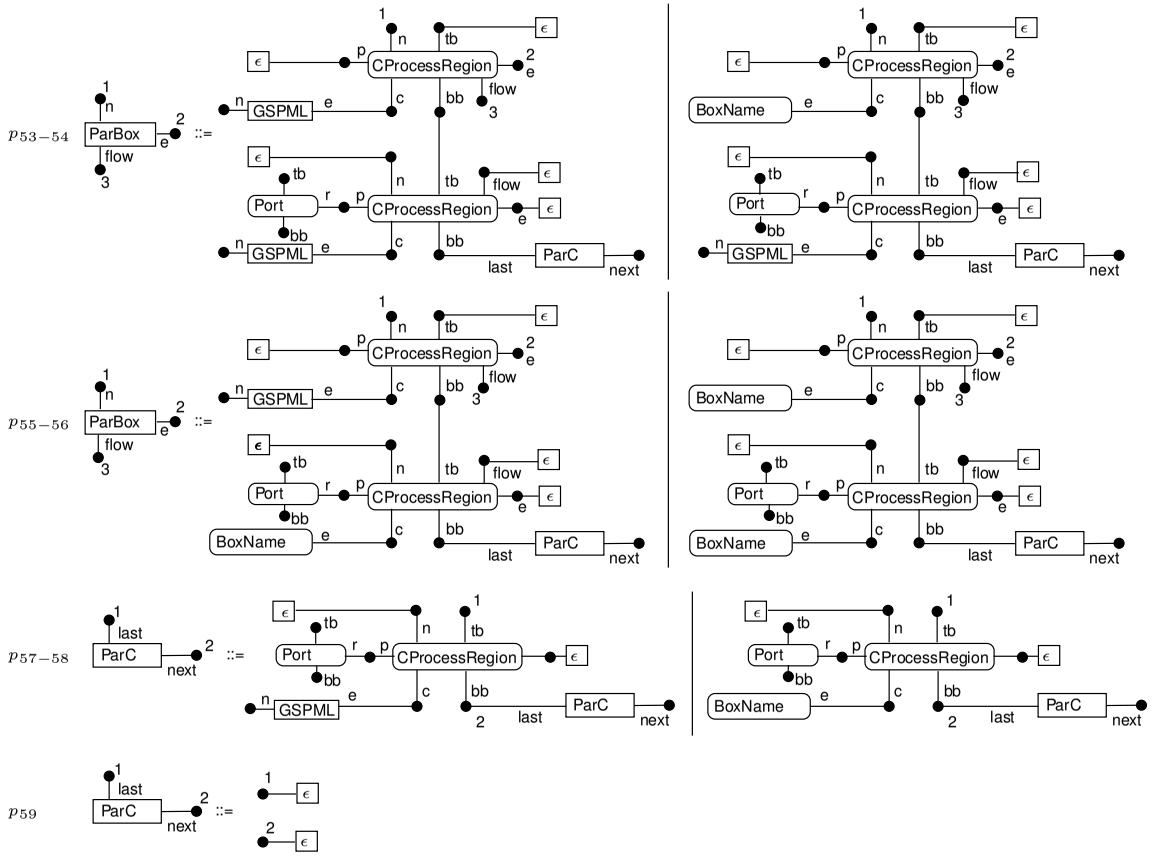

Figure 16: Syntax of the Parallel Concurrent Box

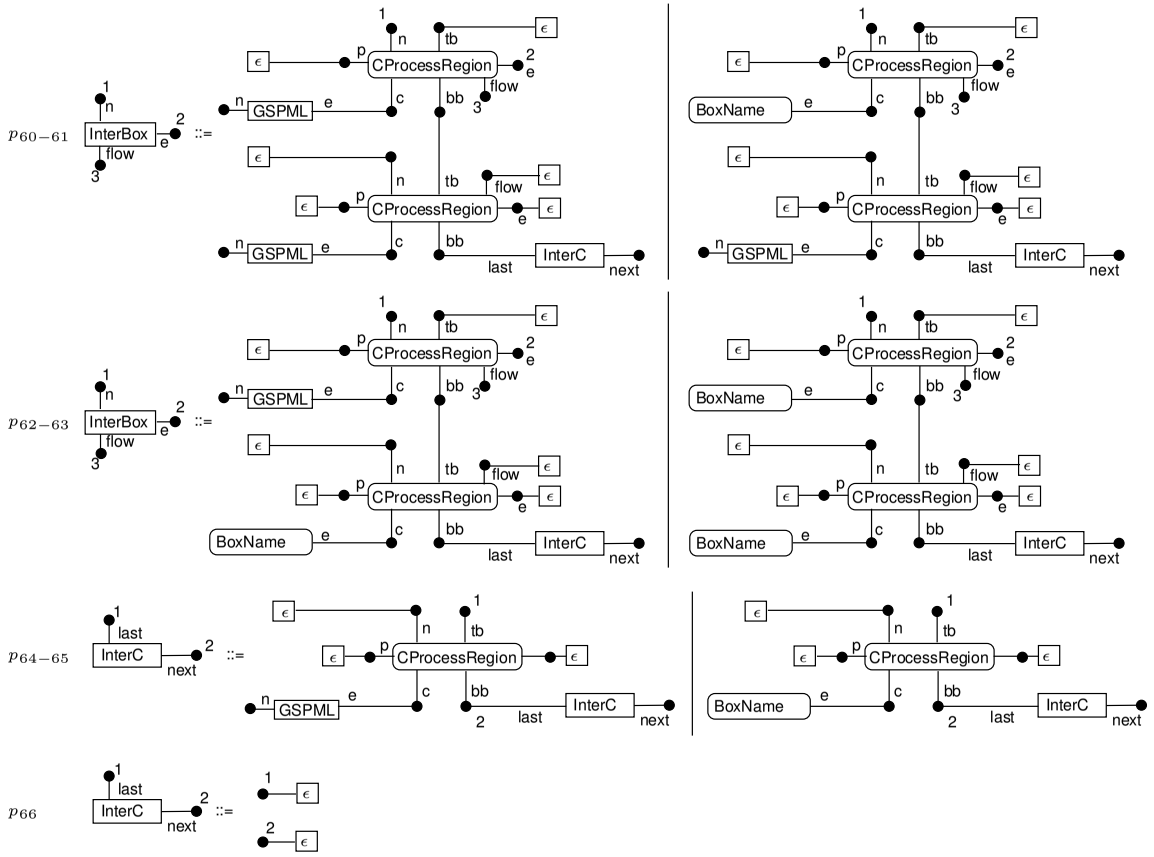

Figure 17: Syntax of the Interleaving Concurrent Box

Figure 18: Syntax of the Abstract Concurrent Box

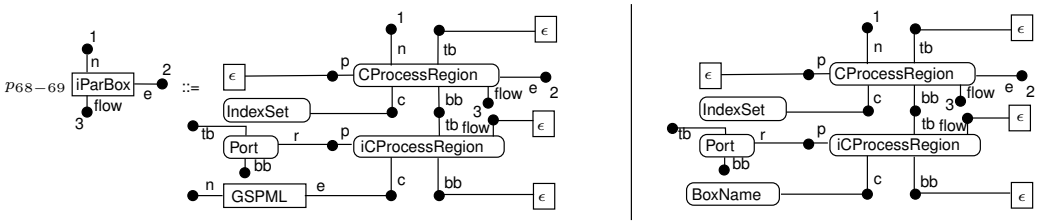

Figure 19: Syntax of the Indexed Parallel Concurrent Box

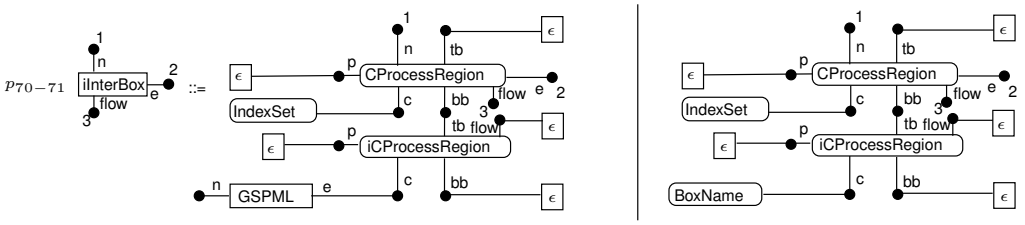

Figure 20: Syntax of the Indexed Interleaving Concurrent Box

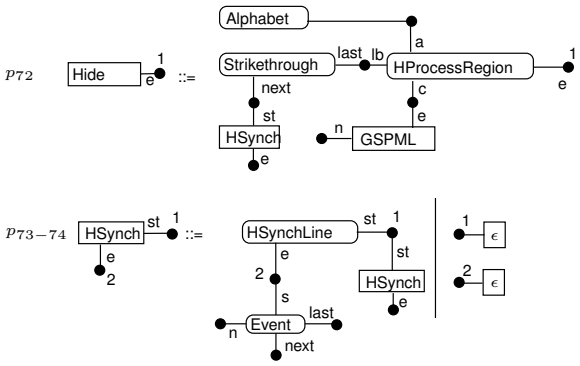

Figure 21: Syntax of Hiding

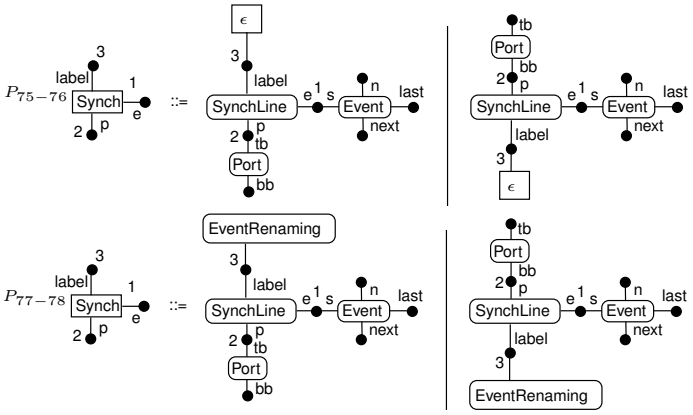

Figure 22: Syntax of Synchronization Lines

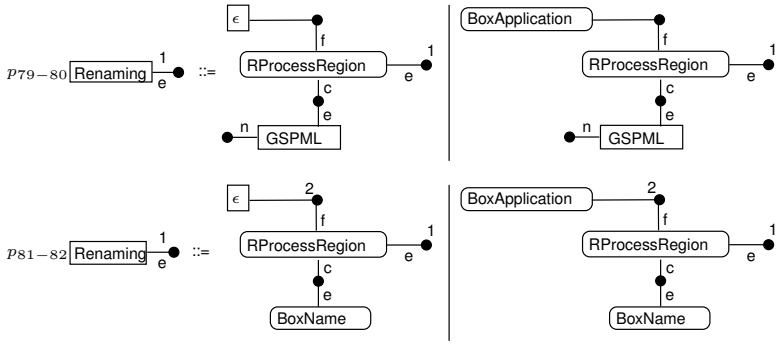

Figure 23: Syntax of Forward Renaming

B The GSPML TSS

This appendix describes the visual TSS that defines the semantics of GSPML. It assumes that the reader has already seen Sections 3 and 4.

B.1 Sequential Boxes

This section defines GSPML semantics for simple sequential boxes, the event region convention, menu choice, recursion, external choice and internal choice.

B.1.1 Simple Sequential Boxes

The *Event Region Rule*, shown as Figure 24, defines the most basic semantics of GSPML. It is always the case that a box may perform its event, so there is no premise. Events and event names are depicted as shown in the event region of the rectangular box of Figure 24. The syntax is defined by the hypergraph grammar productions of Figure 9. When the event takes place, the left-hand box is replaced by the box named or drawn in the process region of the left-hand box. Figure 24 shows two transition rules: Event Region Rule (a) and Event Region Rule (b). The (a) form of the rule defines transitions for box names and the (b) rule defines transitions for boxes. Event Region Rule (b) also requires that the innermost process region of a GSPML diagram contain a box name rather than a box.

The Event Region Rule also defines the following visual characteristics:

- *Corner Shape*: A simple sequential box must have rectangular corners.
- *Diagram Symbols*: The box has an event symbol for its event, if the event is visible. The event symbol is a circle. The recommended diameter of the circle is one em space of the font used for text symbols.
- *Text Symbols*: The box may have a box name and an event name for its event.
- *Placement*: The event symbol (represented by the *Event* terminal of the hypergraph grammar) must be placed on the left boundary of the event region. An event symbol should not touch the upper or lower boundary of the event region. The box name should be placed within two em spaces of the upper left corner of the box, outside its boundary.

B.1.2 Event Regions

Given the Event Region Transition Rule we can introduce a convention that allows us to avoid nesting for the sake of sequential events. We call this convention the *Event Region Convention*. This convention is shown as in Figure 25. Its syntax is also given by the productions of Figure 9. The premise box gives the meaning of the convention depicted as the conclusion box.

The Event Region Convention also defines the following visual characteristics:

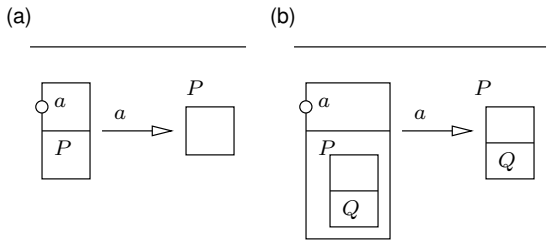

Figure 24: Event Region Transition Rule

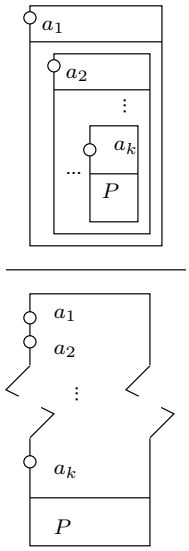

Figure 25: Event Region Convention

- *Text Symbols*: The box must have an event name for each visible event.
- *Placement*: Event symbols must be placed on the left boundary of the event region. The first event appears at the top and succeeding events are placed in order from top to bottom, with the last event at the bottom. Event symbols should be separated by a vertical space of about one ex space of the font used for text symbols.

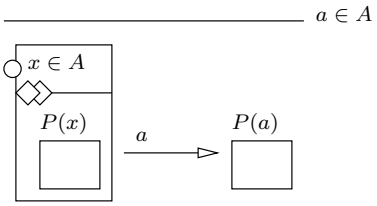

Figure 26: Menu Choice Transition Rule

B.1.3 Menu Choice

The *Menu Choice Transition Rule* allows boxes to choose between first events. Menu Choice is shown in Figure 26. The syntax is defined by the productions shown in Figure 10. The set A is a set of events and the side condition $a \in A$ reminds us that there must be a box $\overset{P(a)}{\square}$ defined for each event in A. If event set A is empty, then a menu choice sequential box is equivalent to $STOP$. Box $\overset{P(a)}{\square}$ must be distinguished by event a but a need not be the first event of $\overset{P(a)}{\square}$.

- *Diagram Symbols*: The lower boundary of the event region must have a double diamond event choice symbol. The size of each diamond should be about two em spaces.
- *Text Symbols*: The form of the event set descriptor must be $x : A$ with the event variable denoted as x and the event set as A. The event parameter in the name of the box contained in the process region must match the event variable of the event set descriptor.
- *Placement*: The event set descriptor must be placed next to the event symbol, inside the event region.

Given Menu Choice's general choice from a set, we can introduce a convention that allows us to depict small sets of events more clearly. This convention is shown in Figure 27 as the *Menu Choice Convention*. The premise box gives the meaning of the convention. Choice of event *i* results in a box that does event *i* and then acts like the process in the *i*th process region of the box. Menu Choice Convention boxes are denoted visually by the single diamond in the lower boundary of each event region. The syntax of the Menu Choice Convention is also included in Figure 10.

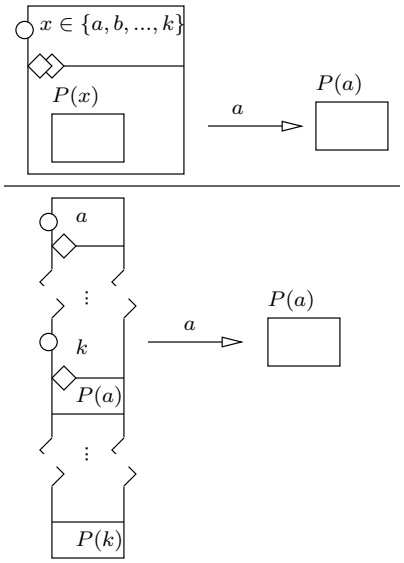

Figure 27: Menu Choice Convention

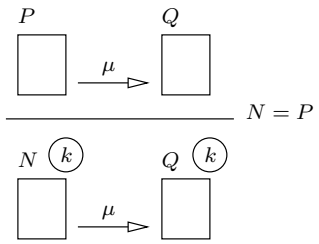

Figure 28: Recursion Transition Rule

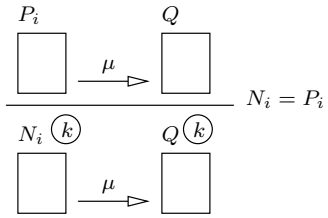

Figure 29: Mutual Recursion Transition Rule

B.1.4 Recursion

Recursion is based on box names. There is no "recursion" box or diagram symbol. A visual reminder of the designer's intent is provided by sets of small numbered circles, once placed next to each box name that defines the recursion. Simple recursion is defined in Figure 28 and general mutual recursion in 29.

- *Diagram Symbols*: Each box name used for recursion should have a corresponding recursion symbol next to it. The recursion symbol should be a circle as shown in Figure 28 with a diameter of about 1.5 em spaces. If large (i.e. more than one text character) recursion symbols are desired, then larger circles are appropriate. It is an error to have only one recursion symbol in a diagram.

- *Text Symbols*: Each recursion symbol should have a single character to identify it with the other box names and recursion symbols that participate in the recursion.

- *Placement*: Recursion symbols should be placed near the right end of the corresponding box names.

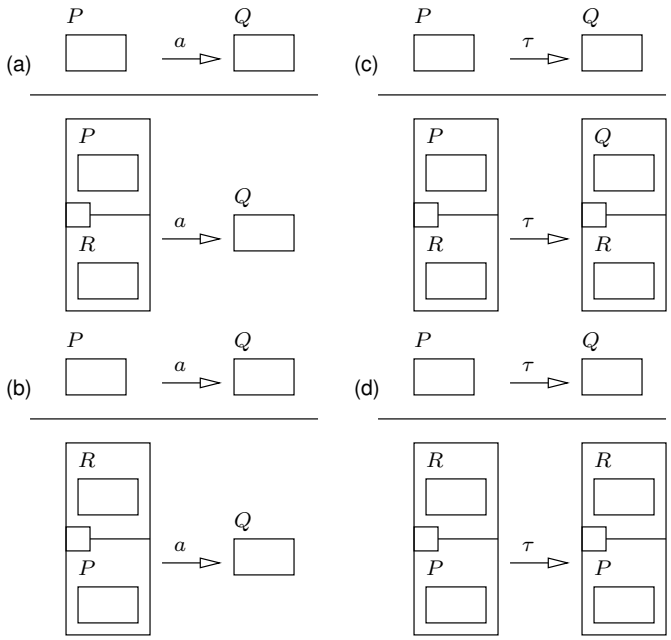

Figure 30: External Choice Transition Rule

B.1.5 Choice

Menu choice boxes define a choice of events with a single process. The other possibility is choice of process. The chief distinction is whether the environment can choose or the box chooses internally.

External choice boxes are initially able to perform the events of either process of choice: the choice is made by the environment of the box and the box's behavior is deterministic. Figure 30 shows the transition rule for external choice. The syntax of External Choice is given by Figure 11. The first pair of rules, (a) and (b), shows the visual commutativity of the choice over visible events. That is, the top-to-bottom order of the processes does not change the meaning and the result is the box from the chosen process region. The second pair, (c) and (d), shows commutativity for hidden events and also defines the fact that hidden events do not resolve the choice of box. That is, in parts (c) and (d) of the rule, the internal transition always results in another external choice box.

External choice boxes are also associative. That is, the order of nesting a choice over three boxes does not change the semantics. Figure 31 depicts this visual associativity. The external choice box of Figure 31(a) has the same meaning as the box of Figure 31(b).

- *Diagram Symbols*: The box should have a square external choice symbol as shown in Figure 30. The size of the square should be between one and two em spaces on a side.

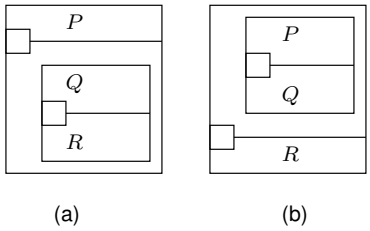

Figure 31: Associativity of External Choice

- *Text Symbols*: The external choice box may have a box name.
- *Placement*: The external choice box should be placed on the left end of the boundary between the two process regions.

Given the basic external choice box, we can apply its commutativity and associativity to define a convention for generalized indexed external choice. The meaning of this convention is shown as Figure 32.

- *Containment*: The single process region will contain a index set specification and a process box.
- *Diagram Symbols*: The box will have a double square indexed external choice diagram symbol.
- *Text Symbols*: The index set will be specified as a text symbol.
- *Placement*: The indexed external choice diagram symbol will be placed at the upper left corner of the box. The index set specification will be placed near the right side of the indexed external choice diagram symbol.

Internal choice boxes allow the depiction of nondeterminism. The meaning of an internal choice box is shown as Figure 33. The two parts to the rule show that an internal choice box may transition into either of the boxes contained in its two process regions via a hidden event. The transition takes place before any event in the environment of the internal choice box. Unlike the external choice box, the internal choice box is not initially prepared to act like either of its contained choices but instead acts as one or the other, in an unpredictable manner. Internal choice boxes are distinguished from external choice boxes by the absence of an external choice diagram symbol. Internal choice boxes have an indexing convention just like external choice. This convention is shown as Figure 34.

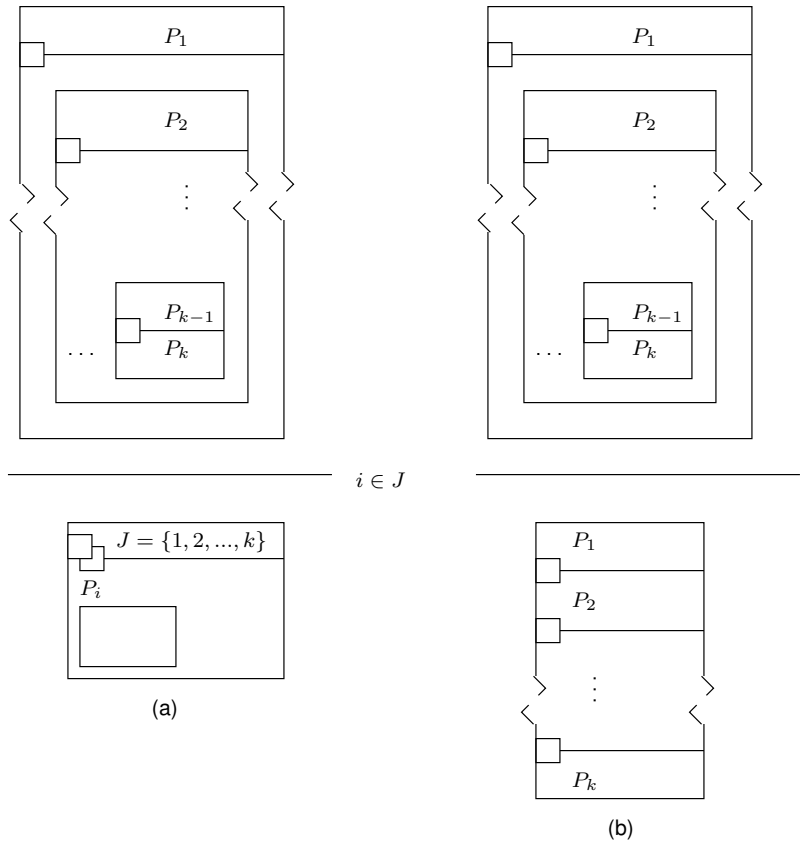

Figure 32: Indexed External Choice Convention

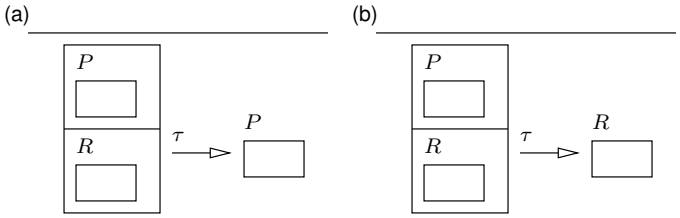

Figure 33: Internal Choice Transition Rule

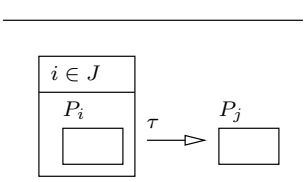

Figure 34: Indexed Internal Choice Convention

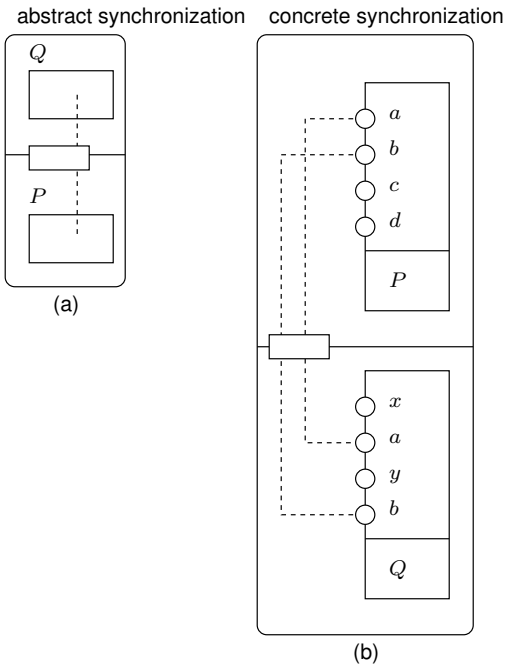

Figure 35: Depicting Abstract and Concrete Synchronization

B.2 Concurrency

Synchronization in GSPML is defined as a broadcast communication mechanism: events that are shared take place at the same time in all processes. In GSPML, synchronization is depicted with *synchronization lines* or synch lines. Synchronization lines are drawn as dashed lines. Synchronization lines are drawn to connect all instances of a shared event.

Figure 35 shows how event sharing and synchronization are depicted in GSPML. The right hand box of Figure 35 shows *concrete synchronization*. In concrete synchronization, a synchronization line is drawn connecting each pair of shared events, for all events that synchronize. If events are not connected by a synchronization line, they are not shared.

The left hand box of Figure 35 shows *abstract synchronization* where there are no event symbols connected by synchronization lines but there is an interface port symbol. In abstract synchronization, synchronization lines connecting events are not shown because some of the applicable events are contained within an abstract box. We can still have information about the events that are shared across the interface because:

1. either the interface between the two boxes inside the process regions is explicitly defined as a set of events and the name of the set is written inside the interface port symbol; or

2. there is no set name on the interface port and we assume that all visible events

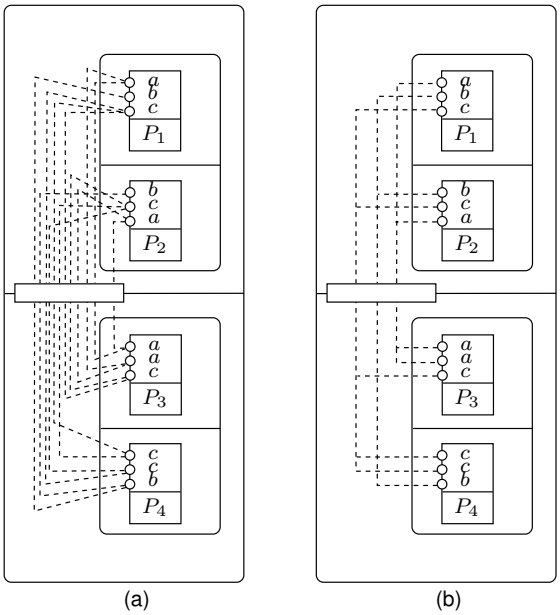

Figure 36: Simplified Depiction of Synchronization Lines

are shared across the interface.

Figure 35 shows the latter kind of abstract synchronization. Abstract synchronization allows us to show synchronization that is taking place, without having to draw a synch line for each shared event. As Figure 35(a) shows, we allow the depiction of unconnected synchronization lines, as a visual reminder that some events are shared, even though the connections are only implicit.

Figure 36 shows how synchronization lines may be depicted in a simpler way, when there is more than one pair of events defined on an event name. In Figure 36(a) full concrete synchronization is show. This is the defined semantics of GSPML synchronization lines. Notice that because sequential boxes P_1 and P_2 are combined via an interleaving box, it is possible for an event with the same name to be synchronized more than once inside a single process region. The upper process region of Figure 36(a) has synchronization of events a and b more than once. If interleaving or nested parallel boxes are used in both process regions of a parallel box, the synchronization lines can become cluttered and hard to understand. Renaming can make the situation even more complex, as multiple event names in one process region may be mapped onto a single event name in another process region. To avoid this, we collapse the multiple synchronization lines into a single branching line that connects all shared events that have the same name. So there is one line for each event name, representing multiple synchronization lines. Figure 36(b) shows this simplification, applied to the model of Figure 36(a).

B.2.1 Interleaving

Interleaving concurrency implies no synchronization except on termination. The events of two interleaved boxes happen independently but one box cannot continue after the other box has terminated. Figure 37 depicts this. Parts (a) and (b) of Figure 37 define the commutativity of an interleaving box and Figure 37(c) defines the common termination.

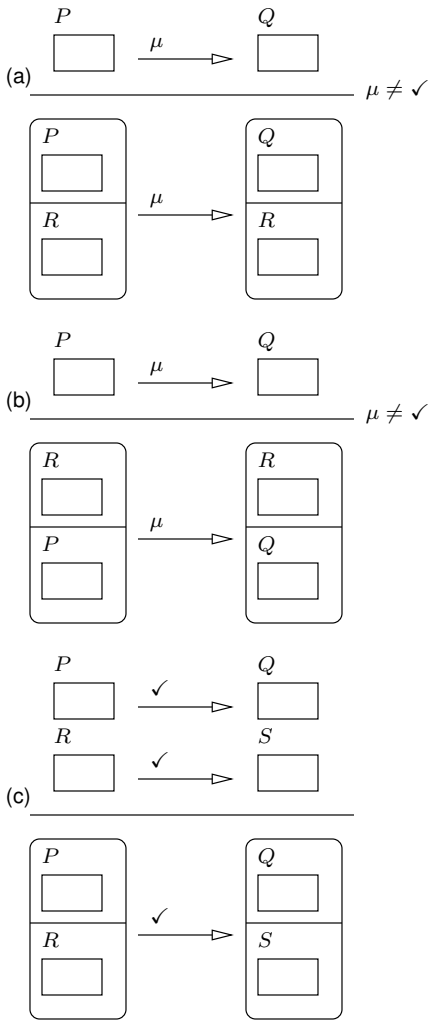

Figure 37: Interleaving Transition Rule

- *Corner Shape*: The box corners are round. The radius of the corners should be about two em spaces.

Because an interleaving box is both associative and commutative, we can use a convention to compactly depict a large number of interleaved boxes. The convention is show as Figure 38. Figure 38(a) shows the indexed interleaving convention while Figure 38(b) shows a convention that uses associativity to simplify an enumeration of interleaved boxes.

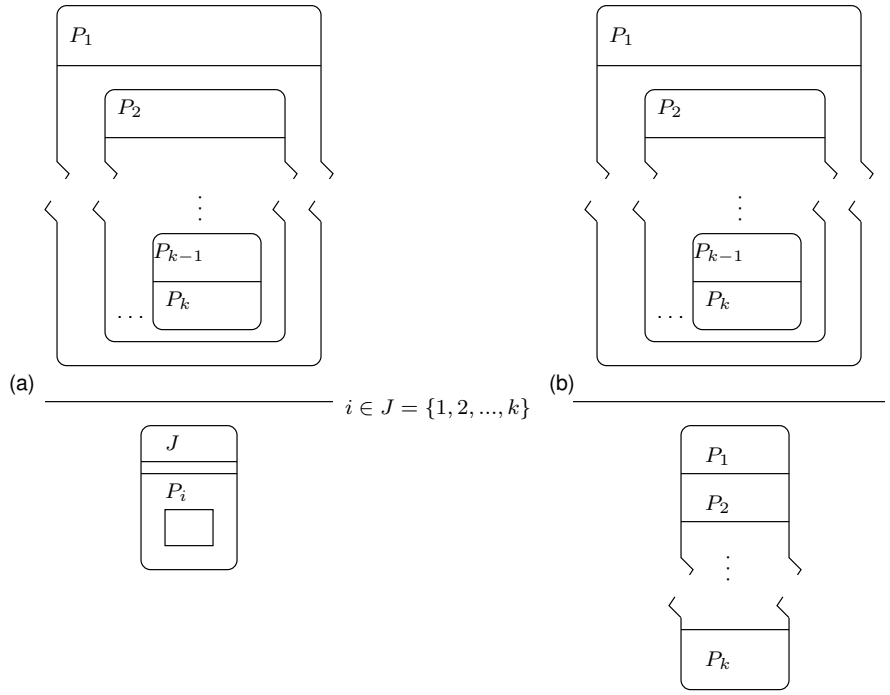

Figure 38: Indexed Interleaving Convention

- *Diagram Symbols*: An indexed interleaving box has a double line separating its two process regions.

B.2.2 Parallel

GSPML has only one form of parallel box. This parallel box is based on the CSP *interleaving parallel* operator. Parallel boxes synchronize via an *interface set*. Figure 39 defines the transition rule for interface parallel. Figure 39(a) shows that events from the interface set happen in both processes at the same time. If either box contained in a parallel box terminates then the entire parallel box terminates. Figure 39(b) and (c) show that a parallel box acts just like an interleaving box, for events not in the interface set.

- *Corner Shape*: The box corners are round. The radius of the corners should be about two em spaces.

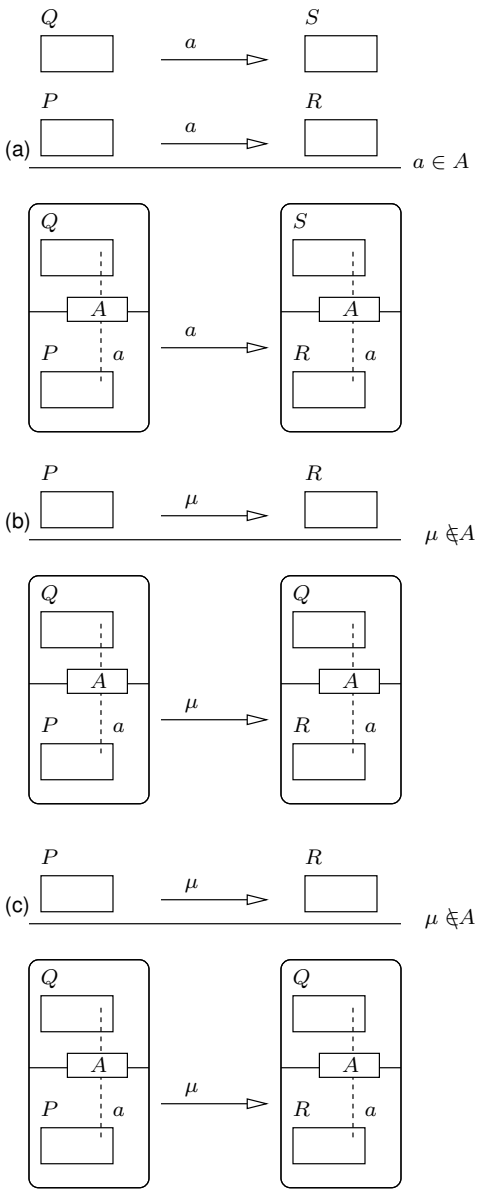

Figure 39: Parallel Transition Rule

- *Diagram Symbols*: The box will have an interface port symbol. The interface port symbol should be between two and three em spaces high and leave at least one em space around either end of the interface set name. Synchronization lines may be drawn between the event symbols of shared events. A synchronization line is drawn as a single dashed line that connects two or more events, all of which have the same name. Abstract synchronization lines may be used but will not connect event symbols.

- *Text Symbols*: The box will have an interface set name or specification. Each synchronization line may have the associated event name placed near it.

- *Placement*: The interface set specification will be placed inside the interface port symbol. The interface port symbol will be placed on the boundary between the two process regions. A synchronization line must be drawn to pass through the applicable interface port symbol.

Because indexed parallel is also commutative and associative, across an identical interface set, we define an indexed parallel convention. This is shown in Figure 40 where the interface port symbol is placed on a double line separating the two process regions.

- *Containment*:The upper process region contains the index set specification and the lower process region contains a box with a parameterized name relating to the index set specification.

- *Regions*: The box will have two process regions

- *Diagram Symbols*: An indexed parallel box has a double line separating its two process regions. The box will have an interface port symbol. The interface port symbol should be between two and three em spaces high and leave at least one em space around either end of the interface set name. Synchronization lines may be drawn between the event symbols of shared events. A synchronization line is drawn as a single dashed line that connects two or more events, all of which have the same name. Abstract synchronization lines will not connect event symbols.

- *Placement*: The interface port symbol will be placed on the boundary between the two process regions. A synchronization line must be drawn to pass through the applicable interface port symbol.

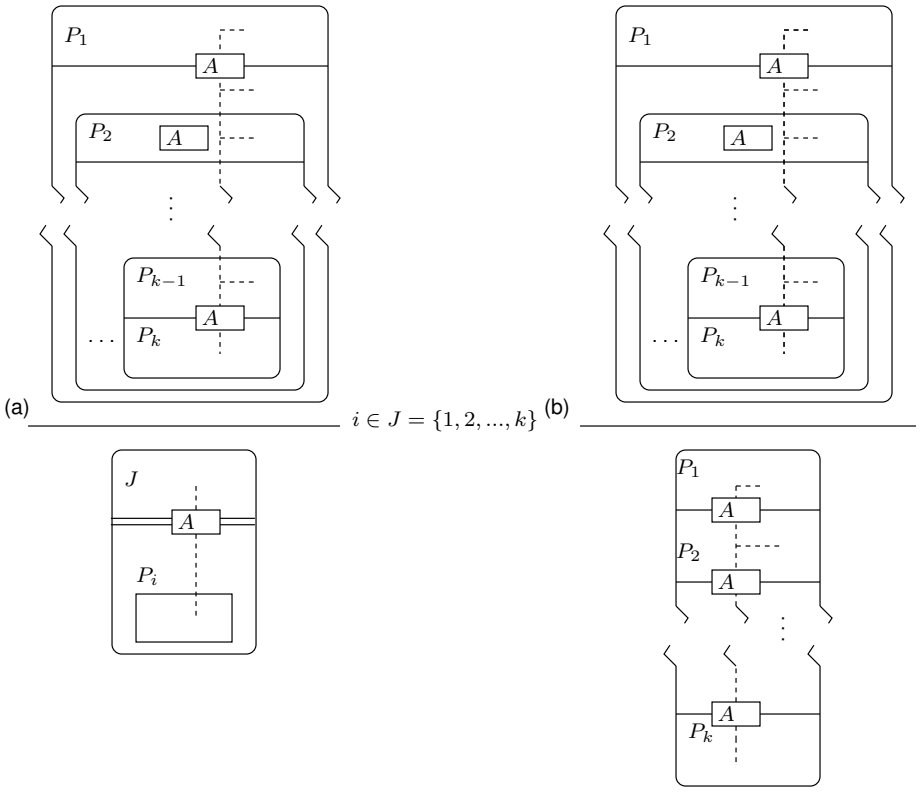

Figure 40: Indexed Parallel Convention

B.3 Abstraction

It is also useful to have some explicit abstraction operators: *hiding* and *renaming*. Abstraction by hiding allows us to conceal synchronization outside the interface of a parallel combination. Hiding is also useful in specifying security for some kinds of protocols. Renaming supports re-use of box descriptions and simplifies the presentation of complex models.

B.3.1 Hiding

Hiding removes events from the interface of a process; it makes them *internal events* to any box enclosing the hiding box. Inside the hiding box, the events are not internal events. Figure 41 shows the box symbol for hiding and defines the Hiding Transition Rule. Figure 41(a) shows the meaning of hiding, for events that are hidden and Figure 41(c) shows the meaning for events that are not hidden. Figure 41(b) shows the meaning of hiding events that are nested.

- *Corner Shape*: The corners of a hiding box are rectangular
- *Diagram Symbols*: A hiding box has one or more *hiding strikethrough symbols*, as depicted in Figure 41.
- *Placement*: Each hiding strikethrough diagram symbol must be drawn on the left boundary of the hiding box. Synchronization lines must be drawn connecting the strikethrough symbols to the event symbols for the events that are to be hidden, for concrete hiding, or terminating inside the process region of a box, for abstract hiding.

B.3.2 Renaming

Figure 42 depicts the rule for forward renaming. Forward renaming is depicted as a rectangular box drawn with dashed lines, with *renaming symbols* on its left boundary. Renaming symbols are circles of the same size as event symbols, but drawn with a dashed line. Synchronization lines for the renamed events are drawn from the renamed event through the renaming symbol. Figure 46 at the end of this report shows examples of renaming symbols. The dashed line used to draw the renaming box and its renaming symbols serves as a reminder that synchronization is likely to be affected, since that is the most frequent use of renaming. Forward renaming is accomplished by applying a renaming function f to event names a. The function application, for example $f(a) = x$, is shown next to the renaming symbol or on the affected synchronization line. The renaming function is then explicitly defined for each event name that is affected. For events that are not renamed, the meaning is the identity function. The renaming function f must be a total function on events, including hidden events τ and terminations $\checkmark$. Termination cannot be renamed and renaming may not be used to hide events; external events must be renamed to external events.

- *Diagram Symbols*: A forward renaming box has one or more renaming symbols drawn on its left boundary. Renaming symbols are the same size and shape as event symbols, but drawn with a dashed line.

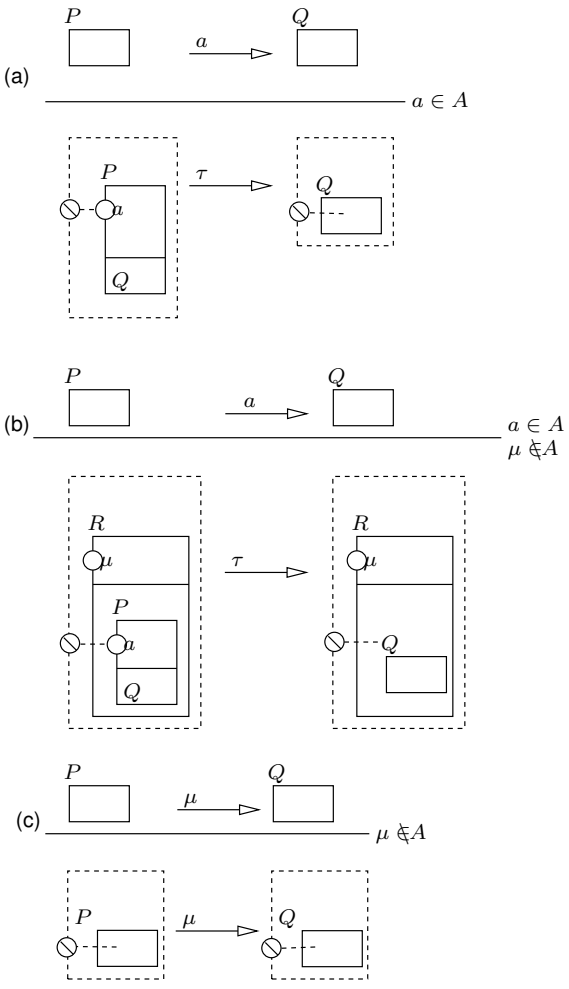

Figure 41: Hiding Transition Rule

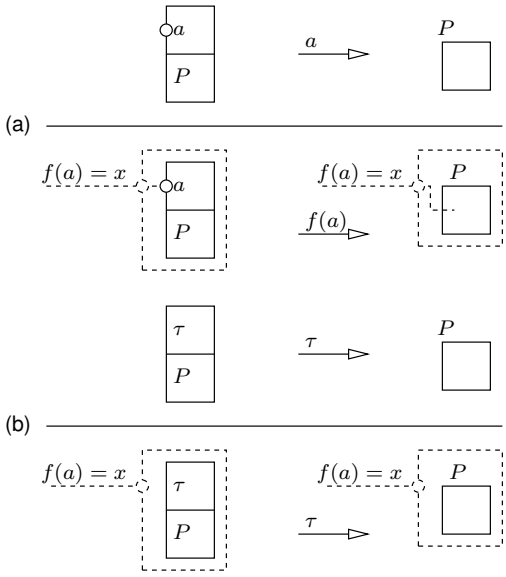

Figure 42: Forward Renaming Transition Rule

- *Text Symbols*:Synchronization lines that are affected by the renaming function must show the function application to the event name, as a label on the synchronization line. An example of this is shown in Figure 46.
- *Placement*: Renaming symbols must be placed on the left boundary of the renaming box. Affected synchronization lines must be drawn through the appropriate renaming symbol.

Figure 43 shows the transition rule for backward renaming. Backward renaming is depicted as a rectangular box drawn with dashed lines, using renaming symbols. The dashed line serves as a reminder that synchronization is likely to be affected, since that is the most frequent use of renaming.

A function f is defined but its inverse f^{-1} is applied to the box and event names. Backward renaming where f is bijective is the same as forward renaming. Affected synchronization lines are labeled with the function rule that applies to the event connected by the synchronization line.

When the inverse renaming function f is many-to-one, then backward renaming expands the interface of the box being renamed. That is, when the box named $f(P)$ can perform the event named a then the renamed box $f^{-1}(P)$ can perform any of the events that map to event a.

Visually, backward renaming is depicted in the same way as forward renaming.

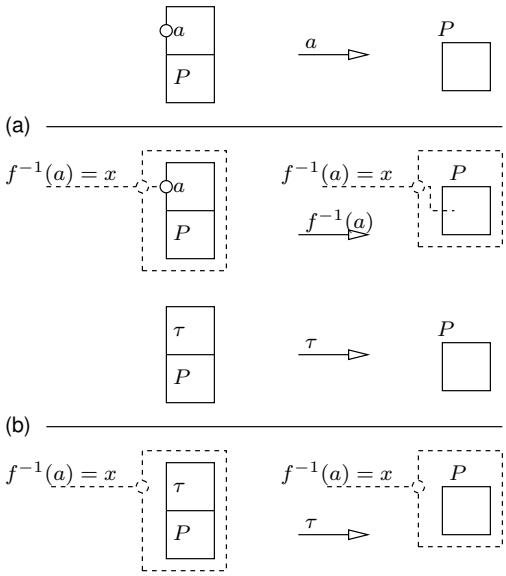

Figure 43: Backward Renaming Transition Rule

B.4 Flow Control

(Sequential) flow control supports modeling of transfer of control from one process to another as a result of the first process no longer executing, that is, the first process box is replaced by a second process box. The replacement may happen after normal termination of the first box or it may happen as a result of the first box being interrupted.

B.4.1 Sequential Composition

Sequential composition models the sequential transfer of control from a preceding process box P to a succeeding process box R, as shown by the transition rule in Figure 44. Figure 44(a) shows the transition for events other than ✓ and 44(b) shows the transition for ✓.

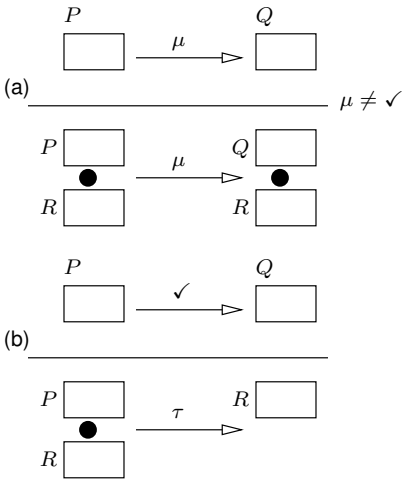

Figure 44: Sequential Composition Transition Rule

- *Diagram Symbols*: Sequential composition is denoted by a sequential composition diagram symbol. The sequential composition diagram symbol is a solid or filled circle of the same size as an event symbol
- *Placement*: The preceding box is draw above the succeeding box, as shown in 44. The sequential composition symbol should be outside the first box touching the bottom of the box. The top of the succeeding box should touch the sequential composition symbol.

B.4.2 Interrupt

Interrupt composition supports the modeling of arbitrary transfer of control from a preceding process to a succeeding one. This arbitrary transfer can model an interrupt in a protocol or it can be used to model fault events. With this construction, the interrupt takes place on the first visible event of the succeeding box. Figure 45 shows the

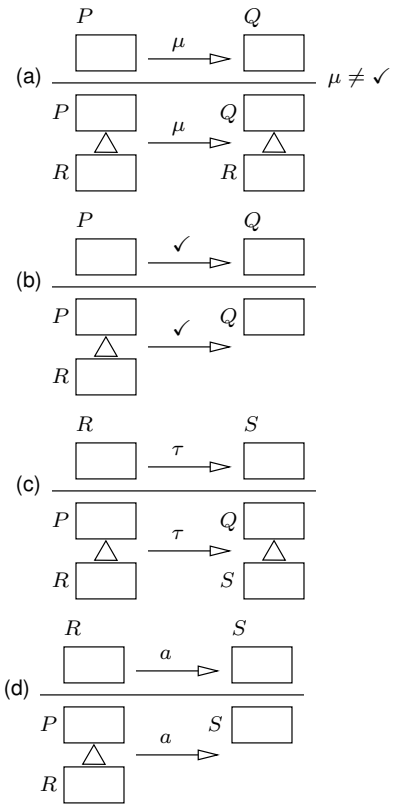

Figure 45: Interrupt Transition Rule

four-part transition rule for interrupt composition. Figure 45(a) gives the meaning of transitions where the preceding box has an event other than ✓ and Figure 45(b) gives the meaning of a normal termination event ✓ in the preceding box. Figure 45(c) shows how the succeeding or interrupting process can make progress via hidden transitions, without triggering an interruption. Figure 45(d) defines the meaning of an actual interruption.

- *Diagram Symbols*: Interrupt composition is denoted by an interrupt composition diagram symbol. The interrupt composition diagram symbol is an open or unfilled equilateral triangle. Each side of the interrupt composition symbol should be between one and two ex spaces long.

- *Placement*: The preceding box is draw above the succeeding box, as shown in 44. The interrupt composition symbol should be outside the first box touching the bottom of the box, with a vertex as shown in Figure 45. The top of the succeeding box should touch the lower edge of the interrupt composition symbol.

C Dolev-Yao Model of Yahalom Security Protocol

The diagram on the following page is a GSPML model of the Yahalom protocol with a Dolev-Yao intruder. The single diagram defines all possible interactions between multiple concurrent runs of the protocol and the intruder, including possible replay and interleaving attacks. This GSPML model is explained in detailed by McDermott [1]. It is repeated here, with a brief summary, as a convenient example of how GSPML satisfies all of our criteria for visual security protocol modeling.

The basic Dolev-Yao structure of the model is shown by the outermost parallel box named *Yahalom*. The principals are represented by the boxes *Alice*, *Bob*, and *Jeeves*. All communications between the principals is routed through the interface port of the *Yahalom* box to the intruder *Yves*. The intruder *Yves* is represented as a external choice box that uses recursion to copy every message of every run of the protocol, with the option of relaying either the proper message, any message previously seen, or an arbitrary bogus message. Both *Alice* and *Bob* use interleaving boxes to model their ability to act simultaneously in either the protocol initiator or responder roles, in distinct protocol runs.

The model achieves synchronization while retaining meaningful names for components. It accomplishes this through renaming. A renaming function f is used to rename the $send$ and $receive$ components of events in the principals, to $take$ and $fake$, respectively. The event names $take$ and $fake$ serve as intermediate channels through the interface between the protocol principals and the intruder. On the intruder's side of the interface, intruder events $learn$ and say are mapped to $take$ and $fake$. The mapping here also includes the source and destination address of each event $x.y$, to model the intruder's use of eavesdropping and replay. The source and destination become simply a chunk of data to be decoded, rearranged, and manipulated by the intruder as part of the messages it sees and saves.

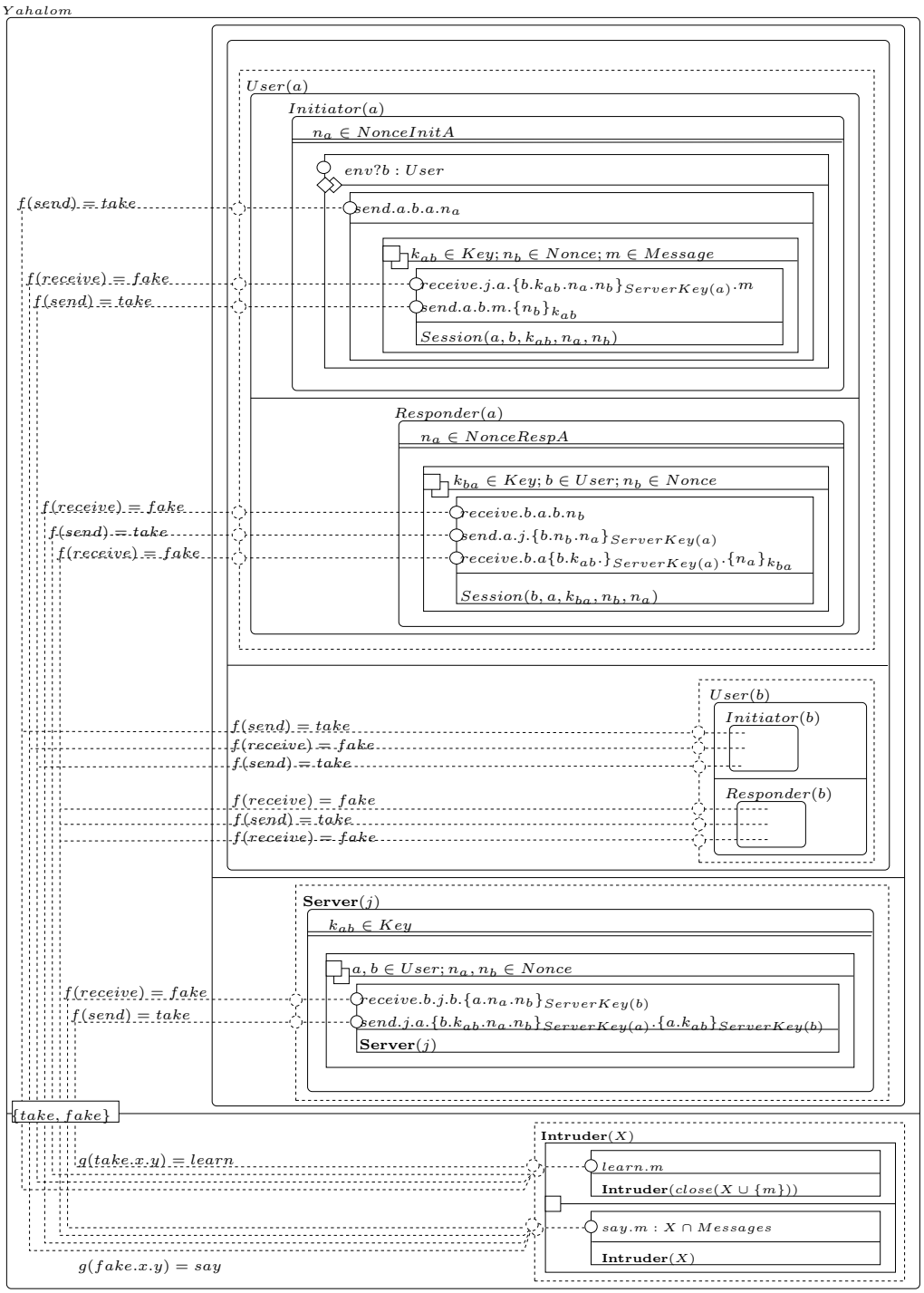

Figure 46: Yahalom: The Complete Protocol Model