Specifying and Analyzing Security Automata using CSP-OZ*

David Basin
Department of Computer Science
ETH Zurich
8092 Zurich, Switzerland
basin@inf.ethz.ch

Ernst-Ruediger Olderog
Department of Computing Science
University of Oldenburg
26129 Oldenburg, Germany
olderog@informatik.uni-oldenburg.de

Paul E. Sevinc
Department of Computer Science
ETH Zurich
8092 Zurich, Switzerland
paul.sevinc@inf.ethz.ch

ABSTRACT

Security automata are a variant of Büchi automata used to specify security policies that can be enforced by monitoring system execution. In this paper, we propose using CSP-OZ, a specification language combining Communicating Sequential Processes (CSP) and Object-Z (OZ), to specify security automata, formalize their combination with target systems, and analyze the security of the resulting system specifications. We provide theoretical results relating CSP-OZ specifications and security automata and show how refinement can be used to reason about specifications of security automata and their combination with target systems. Through a case study, we provide evidence for the practical usefulness of this approach. This includes the ability to specify concisely complex operations and complex control, support for structured specifications, refinement, and transformational design, as well as automated, tool-supported analysis.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages

General Terms

Security, Languages, Verification

Keywords

CSP-OZ, security automata

1. INTRODUCTION

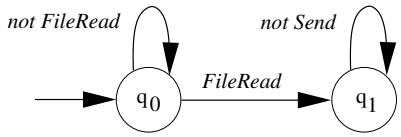
Security automata were introduced by Schneider [32] as a means for characterizing the class of security policies enforceable by mechanisms that work by monitoring system execution. These automata are a variant of Büchi automata that accept finite and infinite sequences of input symbols. They execute in tandem with a target system, restricting the target to those executions permitted (i.e., accepted) by the automaton. In addition to using security automata as a theoretical construct to *characterize* the class of security policies enforceable by monitoring, Schneider observed that, when specified using guarded commands, security automata also provide a useful notation to *specify* such policies.

Our focus in this paper is on this second aspect: how can security automata be specified in a practical way? More generally, how can one specify monitor-like enforcement mechanisms and their combination with target systems? Any proposed solution should scale to complex, real-world systems. Ideally, it should embody proven techniques from the world of specification languages (namely, abstractions for specifying, structuring, and composing designs) and support for refinement, transformation, and other kinds of reasoning about specifications. In this paper, we show how the specification language CSP-OZ [12] can be used to meet all of these requirements.

Our contributions are both theoretical and practical. Theoretically, we formally connect Schneider's concepts to the theory of Communicating Sequential Processes (CSP) [21] and thereby also to CSP-OZ. To begin with, we relate the acceptance condition of security automata to the trace model of CSP. In doing so, we prove that trace refinement in CSP is sufficient for checking safety properties of security automata. Afterwards, we show that CSP's parallel composition with synchronizing events provides a formal counterpart for Schneider's notion of the tandem simulation of a security automaton with a target system. We then prove a general result relating systems secured with security automata to unprotected systems: when restricted to a common interface, the secure system is a refinement of the insecure system.

Practically, we show how a combination of CSP with Object-Z [35] can be used as a specification language for specifying and reasoning about security automata. Our thesis is that the resulting language, CSP-OZ, is very well suited for specifying and reasoning about complex security automata and their combination with large-scale systems. This in-

*This work was partially supported by the Zurich Information Security Center. It represents the views of the authors.



state vars *state*:{0,1} **initial** 0

transitions
not *FileRead* ∧ *state* = 0 → **skip**
FileRead ∧ *state* = 0 → *state* := 1
not *Send* ∧ *state* = 1 → **skip**

Figure 1: Security automaton specified graphically and as guarded commands

cludes the ability to specify concisely complex operations and complex control, support for structured specifications, refinement, and transformational design, as well as automated, tool-supported analysis.

We present a case study in banking that provides support for the above thesis. We specify a (simplified) secure bank as the composition of an insecure bank with a security automaton and we analyze the resulting combination using a model checker. Afterwards, we use transformational development to explore and formally relate different system architectures. For example, we relate an insecure bank with a secure bank (the latter is a refinement of the former) and show that there are different equivalent ways of structuring an architecture for authentication, authorization, and access control. This shows that CSP-OZ is a good starting point not only for specifying security automata, but also for exploring and interrelating different points in the security-design space. In a companion paper [34], we have applied CSP-OZ to a larger example, specifying a system for secure document processing.

The remainder of the paper is organized as follows. In Section 2, we provide background on security automata and CSP-OZ. In Section 3, we formally relate CSP-OZ specifications to security automata, show how to combine these specifications with those of target systems, and establish general results about such formalizations. In Section 4, we present our case study and, in Section 5, we discuss related work and draw conclusions.

2. BACKGROUND

Security automata are a variant of Büchi automata [36] that accept safety properties. In [32], a security automaton is a four-tuple $\mathsf{A} = (Q, S, I, \delta)$, where: Q is a countable set of *automaton states*; $S \subseteq Q$ is a set of *initial states*; I is a countable set of *input symbols*; and $\delta : Q \times I \to 2^Q$ is a *transition function*, where 2^Q denotes the power set of Q. As usual, δ can be extended to a function $\delta^* : Q \times I^* \to 2^Q$ such that, for $w \in I^*$ a finite sequence of input symbols, $\delta^*(q, w)$ records the set of states reachable from q by iteratively applying δ to each input symbol in w.

There are two primary differences with other presentations of Büchi automata. First, the set of states and inputs may be infinite. The infinite-state set allows one to model enforcement mechanisms as infinite-state systems, which is a common abstraction of many computer programs. Similarly, the infinite set of input symbols allows one to model the monitoring of a target system that itself is infinite-state or can react to infinitely many events. Second, a different acceptance condition is used. Namely, to process a (finite or infinite) sequence $s_1 s_2 \cdots$ of input symbols, the *current state* set Q' of the security automaton starts equal to S. As each input symbol s_i is read, the security automaton changes Q' to $\bigcup_{q \in Q'} \delta(q, s_i)$. If Q' is ever empty (indicating no transition), then the input is rejected and otherwise the input is accepted. Under this definition, security automata can accept both finite and infinite input sequences.[1] We write $\mathcal{L}_\infty(\mathsf{A})$ to denote the set of all input sequences accepted by A. In [32], it is noted that $\mathcal{L}_\infty(\mathsf{A})$ is a safety property in the sense of Alpern and Schneider [2, 3]. Furthermore, let $\mathcal{L}(\mathsf{A})$ denote the set of all *finite* input sequences accepted by A.

Figure 1 provides a simple example, taken from [32], of a security automaton represented in two equivalent ways. The automaton specifies a security policy prohibiting the execution of *Send* operations after a *FileRead* has been executed. The left-hand side specifies the automaton graphically, using predicates as syntactic sugar to denote sets of events. For example, the predicate *not FileRead* is satisfied by any input symbols (modeling system execution steps) that are not file-read operations. Formally, this automaton accepts all input sequences that are either (i) finitely or infinitely many *not FileRead* operations, or (ii) finitely many *not FileRead* operations followed by one *FileRead* operation, or (iii) sequences of kind (ii), followed by finitely or infinitely many *not Send* operations. The right-hand side provides an alternative representation of the automaton as a set of guarded commands. In this example, there is no essential difference between the two. But in general, there may be other state variables, in which case the guarded commands describe something more akin to an extended state machine [38], where state variables may be updated during transitions. Hence, analogously to the way that extended state machines may specify transition systems substantially more compactly than ordinary automata, specifying automata using guarded commands may result in significant savings over pictures or explicitly enumerated transition tables.

While Schneider's examples are elegant, they are also relatively simple.[2] They require neither complex control structures nor complicated functions over rich (e.g., recursive) data types. Moreover, each is small enough to be specified and understood as a simple list of guarded commands. Schneider suggests that policies may be combined by conjunction, which corresponds to conjoining security automata (via a product construction), whereby an execution is rejected if and only if it is rejected by any automaton in the

[1]Note that the acceptance of an infinite sequence $s_1 s_2 \cdots$ does not imply that there exists an infinite sequence of consecutive transitions in A, i.e., that $q_{i+1} \in \delta(q_i, s_i)$ for all $i \geq 1$ and $q_1 \in S$. However, if the transition function exhibits *bounded nondeterminism*, i.e., the range of δ is always a finite set, then such an infinite transition sequence does exist by König's lemma. The results in this paper do not depend on the existence of such infinite transition sequences.

[2]We will compare with other alternatives in Section 5.

collection. But conjunction is only one way of combining specifications and there are substantially more expressive starting points for formalizing processes than guarded commands; recall the desiderata for practical specification languages mentioned in the introduction. This was our motivation for investigating alternative specification techniques and their advantages.

CSP-OZ is a specification language for reactive systems [12, 13] that combines two views of such systems: a control view specified using the notation of Communicating Sequential Processes (CSP) [21, 31] and a data view specified using Object-Z (OZ) [9, 35]. Each CSP-OZ specification can be seen as a structured way of presenting a possibly infinite-state process that manipulates a possibly infinite data space. We briefly describe some of the main characteristics of this language and its sublanguages. The references should be consulted for further details.

CSP was introduced by Hoare [20, 21]; its central concepts are synchronous communication along channels between different processes, parallel composition, and hiding of internal communication. Syntactically, CSP processes are built up from several sets of symbols. The starting point is the set $(a, b \in)\ \Sigma$ of visible *events*. Structured events of the form (c, v), where c is a channel of some type T and v is a data value from T, are called *communications*. For a tuple $v = (v_1, ..., v_n)$, with $n \geq 1$, the communication (c, v) is also written as $c.v_1. \ ... \ .v_n$. Subsets (A, B) of Σ are called *alphabets*. Besides events, there is the symbol $\tau \notin \Sigma$ representing an *internal* or silent action of a process. The set of *actions* is given by $(\alpha, \beta \in)\ Act = \Sigma \cup \{\tau\}$. We also need the set $(X, Y \in)$ *Idf* of (*process*) *identifiers* that can appear as names of CSP processes.

We consider the set $(P, Q \in)$ *Proc* of CSP processes defined by the BNF syntax

$$P \ ::= \ \text{STOP} \ \big| \ a \to P \ \big| \ P \sqcap Q \ \big| \ P \ \Box \ Q \\ \big| \ P \ [|\ A\ |] \ Q \ \big| \ P \setminus A \ \big| \ X \, .$$

STOP represents a process that cannot engage in any action. The process $a \to P$ first engages in an event a and then behaves like P. This *prefix operator* is a restricted form of sequential composition. The process $P \sqcap Q$ represents *internal nondeterministic choice* and can choose to behave like P or Q, without influence of the environment (e.g., a user or another process). In contrast, in the *alternative composition* $P \ \Box \ Q$, the environment selects by its first communication whether this process behaves like P or Q. In the *parallel composition* $P \ [|\ A\ |] \ Q$, the processes P and Q run in parallel, but must synchronize on all events in the *synchronization alphabet* A. The *hiding operator* $P \setminus A$ conceals all events in the alphabet A. These events are transformed into internal actions τ. Finally, a process identifier X represents a *call* of a process P defined by a corresponding equation $X = P$. Processes are often specified by systems of mutually recursive equations where the identifiers have *parameters* ranging over data values: $X(v_1, ..., v_n) = P$. By convention, the unary prefix operator $a \to$ binds stronger than the binary operators $\sqcap$, $\Box$, and $[|\ A\ |]$.

Associative operators like $\sqcap$ and $\Box$ can be *replicated*. For example, $\Box\ a : A \bullet a \to P(a)$ defines a process ready to engage in any communication a in the alphabet A and then behave as $P(a)$. We will also employ standard notation for output and input on channels: *output* $c!v \to P$ stands for $c.v \to P$ and *input* $c?x \to P(x)$ stands for $\Box\ v : T \bullet c.v \to P(v)$, where T is the type of channel c and $P(v)$ results from $P(x)$ by substituting the value v for the variable x. Boolean expressions e over such variables can be used as *guards*: $e \ \& \ P$ equals P if e evaluates to true, otherwise it equals STOP. The channel alphabet $\{|\ c\ |\}$ denotes the event set $\{c.v \mid v \in T\}$. For multiple channels $c_1, ..., c_n$, the notation $\{|\ c_1, ..., c_n\ |\}$ is defined accordingly.

CSP has a rich theory comprising an operational, trace, failures-divergences, and algebraic semantics with consistency proofs [6, 28, 31]. The *operational semantics* assigns to each process P a nondeterministic automaton $\mathcal{A}(P) = (Q_P, S_P, I_P, \delta_P)$, where: Q_P, the set of states, is the set of CSP processes; S_P, the set of initial states, is $\{P\}$; I_P, the set of input symbols, is $\Sigma \cup \{\tau\}$; and δ_P is the nondeterministic transition function. δ_P is given by an inductively defined relation $\longrightarrow$: $Q_P \times I_P \times Q_P$, whose rules describe process execution, namely $q' \in \delta_P(q, a)$ iff $q \xrightarrow{a} q'$. All states in $\mathcal{A}(P)$ are considered accepting.

Note that the components of $\mathcal{A}(P)$ have the same type as those of security automata. However, automata in CSP only accept *finite* sequences over the set Σ, called *traces*. For a nondeterministic automaton A, define its *(automaton) trace semantics* as $\mathcal{T}(\mathsf{A}) = \{w - \tau \mid w \in \mathcal{L}(\mathsf{A})\}$, where $\mathcal{L}(\mathsf{A})$ is the language of classical automata theory where all states are considered accepting and $w - \tau$ denotes the sequence w with all occurrences of τ deleted. The *(CSP) trace semantics* assigns to each process P the set of traces $\mathcal{T}(P) = \mathcal{T}(\mathcal{A}(P))$. A process Q *refines* a process P *in the trace model* $\mathcal{T}$, abbreviated $P \sqsubseteq_{\mathcal{T}} Q$, if $\mathcal{T}(Q) \subseteq \mathcal{T}(P)$. Informally, Q refines P if Q is more deterministic than P, i.e., if Q has fewer traces than P. Trace refinement is insensitive to nondeterminism and divergence. To make finer distinctions, the *failures-divergences semantics* $\mathcal{FD}$ of CSP was developed [6, 31], with its associated notion of refinement, $P \sqsubseteq_{\mathcal{FD}} Q$. For $\mathcal{M} \in \{\mathcal{T}, \mathcal{FD}\}$, we write $P =_{\mathcal{M}} Q$ if $P \sqsubseteq_{\mathcal{M}} Q$ and $Q \sqsubseteq_{\mathcal{M}} T$.

For A as above let its *τ-closure* be the automaton $\mathsf{A}_{-\tau} = (Q, S', I', \delta')$, where:

- $S' = \bigcup_{q_0 \in S,\, w \in \{\tau\}^*} \delta^*(q_0, w)$,
  the set of states reachable from S via zero or more τ-actions;
- $I' = I \setminus \{\tau\}$; and
- $\delta' : Q \times I' \to 2^Q$, with $\delta'(q, a) = \bigcup_{w \in I^*,\, w - \tau = a} \delta^*(q, w)$.

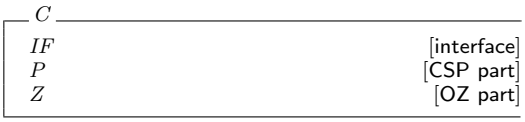
Taking the τ-closure of an automaton does not change its trace semantics, i.e., $\mathcal{T}(\mathsf{A}) = \mathcal{T}(\mathsf{A}_{-\tau})$ for every automaton A.

The FDR (Failures-Divergence Refinement) *model checker* [15, 30] provides tool support for CSP. Given finite-state processes P and Q, FDR can automatically check whether the refinement relations $P \sqsubseteq_{\mathcal{T}} Q$ or $P \sqsubseteq_{\mathcal{FD}} Q$ hold. Additionally, FDR can check whether a finite-state process is deadlock-free, divergence-free, or deterministic (intuitively,

one that can never reach a state where it can accept or refuse the same event). Other properties can be checked with FDR, provided they can be reduced to refinement checks using *test processes*. We will see an example of this in Section 4.

Z is a language based on set theory and predicate logic for specifying data, state spaces, and state transformations [22, 40]. It comprises the mathematical tool kit, a collection of convenient notation and definitions, and the schema calculus for structuring state spaces and their transformations. *Object-Z* is an object-oriented extension of Z [9, 35]. It includes the concepts of classes, instantiation, and inheritance, although in this paper we only use the concept of classes. Z and Object-Z both have a formal notion of data refinement.

CSP-OZ [12, 13, 14] integrates CSP and Object-Z. Here we just describe its main features. Example specifications are given in Section 4. The central notion of CSP-OZ is that of a class, consisting of an interface, a CSP part, and an OZ part. The specification of a class C takes the following form.



The *interface IF* declares channel names and types used by the class. The *CSP part P* constrains the possible communication sequences along the interface channels using CSP process notation. P may consist of multiple processes defined by CSP process equations, one of which is a distinguished process named `main`, which denotes the initial process. The *OZ part Z* comprises a nameless *state* schema describing the state space, a schema `Init` constraining the initial state, and communication schemas `com_`c describing the transformation of the state space induced by communicating along an interface channel c.

The Semantics of CSP-OZ is defined by a transformation into CSP [12, 13]. Thus, each CSP-OZ specification denotes a process that can be analyzed in the trace or failures-divergences model of CSP. Hence, provided all classes specified use only finite-state CSP processes and finite data, the FDR model checker can be used to prove refinement properties of the CSP-OZ specification [14].

For a given class C, the transformation maps the OZ part of C into a CSP process that runs in parallel and communicates with the CSP part of C. Formally, C is transformed into the process

$$proc(C) = \texttt{main} \ [|\ Ev\ |]\ OZMain$$

that consists of the parallel composition of `main` calling the main process in the CSP part and a parameterized process *OZMain* representing the semantics of the OZ part of C. The parallel composition synchronizes on the alphabet *Ev* of all events common to the CSP and the OZ part. The recursive process equations defining *OZMain* are given in Appendix A.

3. SPECIFYING SECURITY AUTOMATA AND THEIR COMBINATION

In this section, we describe how we use CSP-OZ to specify security automata, target systems, and their combination. In doing so, we formally relate CSP-OZ specifications with Schneider's security automata and prove general results about the relationship between protected and unprotected systems.

3.1 Specifying security automata

We define a mapping from each CSP-OZ class C to a security automaton. Take the transformational semantics of C, which yields a CSP process of the form $proc(C) = \texttt{main}\ [|\ Ev\ |]\ OZMain$. We now define the security automaton specified by C as $\mathcal{A}(proc(C))_{-\tau}$, i.e., the τ-closure of the automaton given by the operational semantics of CSP.

In this translation, we map the CSP process $proc(C)$ to the security automaton $\mathcal{A}(proc(C))_{-\tau}$, but these have different semantics. Recall that the trace semantics of a CSP process P is the set of *finite* traces accepted by the automaton $\mathcal{A}(P)$, where all states are considered accepting. In contrast, security automata A are a variant of Büchi automata that accept sets $\mathcal{L}_\infty(\mathsf{A})$ of finite and *infinite* sequences. Fortunately, the semantics are in tight correspondence for security automata defined by CSP processes.

PROPOSITION 1. *For a CSP process P and a sequence $w \in \Sigma^* \cup \Sigma^\omega$: $w \in \mathcal{L}_\infty(\mathcal{A}(P)_{-\tau})$ iff $\mathcal{T}(P)$ contains all finite prefixes of w.*

This proposition establishes that we can indeed view a CSP-OZ class with its trace semantics as a structured specification of a security automaton with a different, but equivalent, notion of acceptance. Moreover, it is easy to show the converse: any security automaton can be represented in CSP-OZ, indeed directly in CSP itself.

Security automata accept safety properties, as noted in Section 2. Since we can specify security automata in CSP-OZ or just CSP, we can use trace refinement to check these properties. For instance, to check whether a security automaton specified by a class C satisfies a safety property specified via the process P, we check the refinement $P \sqsubseteq_{\mathcal{T}} proc(C)$. This is sufficient because, by Proposition 1, we have that

$$\mathcal{L}_\infty(\mathcal{A}(P)_{-\tau}) \supseteq \mathcal{L}_\infty(\mathcal{A}(proc(C)_{-\tau})) \text{ iff } \mathcal{T}(P) \supseteq \mathcal{T}(proc(C)).$$

In contrast, the above semantics are not fine enough to detect properties such as deadlock or divergence freedom. If an automaton can nondeterministically choose between a successful transition and a deadlock or divergence, only the successful transition will be represented in the trace. This is the case for both Schneider's semantics $\mathcal{L}_\infty(\mathcal{A}(proc(C))_{-\tau})$ and our trace semantics $\mathcal{T}(proc(C))$. However, in our setting, we can check these properties by analyzing the automaton $\mathcal{A}(proc(C))$ using the richer failures-divergences semantics.

3.2 Combination with target systems

In [32, p. 40], Schneider describes how security automata are to be combined with a target system via a simulation: *"The target is executed in tandem with a simulation of the security automaton. In particular, initialization or creation*

of the target causes an initialized instance of the security automaton simulation to be created. And each step that the target is about to take generates an input symbol, which is sent to that simulation." In this setting, input rejected by the security automaton causes termination of the target. Hence, policy enforcement is by *execution cutting*.

Automata specified in CSP-OZ can also be combined with target systems in the way described above. In this paper, however, we will explore another possibility, where we work entirely at the specification level. Namely, using CSP's parallel composition, we can provide a formal account of how an unprotected target system *UnpSys* is combined with a security automaton *SecAut* to yield a secure system *SecSys*:

$$SecSys = UnpSys \ [|\ A\ |]\ SecAut\,.$$

Here, we choose the synchronization set A such that *SecAut* obtains all relevant information of *UnpSys* and blocks all transitions of *UnpSys* that are unsafe. The synchronization set provides a formal counterpart of Schneider's account of the target system generating input for the security automaton. One possibility is to require that the security automaton synchronizes with, and hence accepts, *all* transitions of the target system. This corresponds to taking $A = \alpha(UnpSys)$, where $\alpha(UnpSys)$ denotes the set of all events possible for the process *UnpSys*. Alternatively, the security automaton can be specified to monitor only some subset of target system transitions by choosing a smaller synchronization set. Moreover, the security automaton can introduce additional events not present in the target system. This may be used to specify security mechanisms that react to new events, such as user inputs. For example, the security automaton in our bank example will introduce events for entering the *pin* and *tan*.

In both systems and their specifications, it is useful to distinguish between the functionality provided by the system and that presented to the user via a restricted interface. In our setting, both unprotected systems and protected systems may contain operations that should not be accessible to the user. Hence we may choose to hide events in either *UnpSys* or *SecSys*. For example, in the bank specification, the event of executing a requested funds transfer will be hidden from the user.

The above provides a formal account of both security automata and their combination with target systems in CSP-OZ and, via translation, in CSP. Hence all methods and tool support that these languages provide for analysis, refinement, and transformation can be applied to these models. Moreover, based on the semantics of CSP we can prove a general refinement result, once and for all, relating insecure and secure systems. Namely, when restricted to a common interface, the secure system is a refinement of the insecure system. We proceed by first stating several properties of refinement and trace equivalence.

LEMMA 1. *Let P and Q be CSP processes, with $A = \alpha(P) \cap \alpha(Q)$ and $B \subseteq \Sigma$. Then the following refinement relationships hold in the trace semantics of CSP:*

(1) If $\alpha(Q) \subseteq \alpha(P)$ then $P \sqsubseteq_{\mathcal{T}} P \ [|\ A\ |]\ Q$.

(2) If $\alpha(P) \cap B = \varnothing$ then
$(P \ [|\ A\ |]\ Q) \setminus B =_{\mathcal{T}} P \ [|\ A\ |]\ (Q \setminus B)$.

Property (2) also holds in the failures-divergences semantics, i.e., with $=_{\mathcal{FD}}$ instead of $=_{\mathcal{T}}$. Property (1) does not hold in the failures-divergences semantics because $P \ [|\ A\ |]\ Q$ might deadlock even if P is deadlock-free.

PROPOSITION 2. *Let U and S be CSP processes, with $A = \alpha(U) \cap \alpha(S)$ and $B = \alpha(S) \setminus \alpha(U)$. Then the following trace refinement holds:*

$$U \ \sqsubseteq_{\mathcal{T}}\ (U \ [|\ A\ |]\ S) \setminus B\,.$$

PROOF. We apply Lemma 1:

$$\begin{array}{ll} & U \\ \sqsubseteq_{\mathcal{T}} & \quad \{\text{by (1), since } \alpha(S \setminus B) \subseteq \alpha(U)\} \\ & U \ [|\ A\ |]\ (S \setminus B) \\ =_{\mathcal{T}} & \quad \{\text{by (2), since } \alpha(U) \cap B = \varnothing\} \\ & (U \ [|\ A\ |]\ S) \setminus B) \end{array}$$ □

Taking $U = UnpSys, S = SecAut$, and *SecSys* as above, Proposition 2 yields $UnpSys \ \sqsubseteq_{\mathcal{T}}\ SecSys \setminus B$. Thus, by construction, when restricted to a common interface, the secure system is a trace refinement of the unprotected target system.

4. A BANKING EXAMPLE

We illustrate our approach by modeling part of the IT infrastructure of a simple bank, which offers customers operations for accessing accounts and transferring funds. We first model an insecure version of the bank as one without any security measures beyond a simple, passwordless, login mechanism. Afterwards, we secure the bank with a security automaton that runs in parallel and synchronizes with the bank.

4.1 Modeling the bank and its security policy

Our model consists of two processes running in parallel: the bank and the security automaton. The formal specification starts by introducing the sets

$$[UserID, AccID, PIN, TN], \quad Val : \mathbb{P}\,\mathbb{Z}, \quad Sum : \mathbb{P}\,\mathbb{N}$$

needed to specify the parameters in the bank's interface, which provides operations for *login* and *logout*, for checking the *balance* of an account, for *requesting the transfer* of some sum of money from one account to another, for *executing* the transfer, and for *aborting* the transfer. When defining the channel types, Z naming conventions are used, e.g., variables decorated with "?" and "!" denote inputs and outputs.

The state of a *Bank* (representing an insecure bank) maintains the set of its customers and the balances of all accounts. It only accepts logins from users registered as customers, but no credentials, like passwords, are required. After a successful login, the user may check the balance of any account and transfer funds. The bank processes only transfer requests between two different accounts when the amount

to be withdrawn does not exceed the balance of the debiting account. Therefore a transfer is modeled by two operations: *transferReq*, which requests the transfer, and *transferExec*, which transfers the requested funds only if the request succeeds. We will explain the significance of *abort* shortly.

The CSP part of *Bank* models the sequencing constraints of the operations *login*, *logout*, *balance*, *transferReq*, *transferExec*, and *abort*. For example, *balance* is only possible after a successful *login*. Whereas the data values of the operation parameters u, s, $a1$, and $a2$ are determined by the user via *UserIF*, the values of the parameters *ok* and v depend on the state of a *Bank* and are determined by the communication schemas `com_`*login*, `com_`*balance*, and `com_`*transferReq* of the OZ part. Note that the communication schemas `com_`*transferReq* and `com_`*transferExec* change *transferOK* and *bal*, respectively, of the bank's state as indicated by the Δ notation; following Z's convention, these variables are decorated by a prime to denote their value after transformation. Note too that, as specified, the bank is single threaded. Hence we do not need to consider problems of processing (ACID) transactions.

Bank

```
chan login : [ u? : UserID; ok! : 𝔹 ]
chan logout
chan balance : [ a? : AccID; v! : Val ]
chan transferReq : [ s? : Sum; a1? : AccID;
      a2? : AccID; ok! : 𝔹 ]
chan transferExec : [ s? : Sum; a1? : AccID;
      a2? : AccID ]
chan abort

main = login?u?ok → (ok & Operate
                      □ ¬ok & main)
Operate = (balance → Operate
          □ transferReq?s?a1?a2?ok →
              (transferExec!s!a1!a2 → Operate
              □ abort → Operate)
          □ logout → main)
```

	Init
$customers : \mathbb{P}\ UserID$ $bal : AccID \to \mathbb{Z}$ $transferOK : \mathbb{B}$	$true$

com_login	com_balance
$u? : UserID$ $ok! : \mathbb{B}$	$a? : AccID$ $v! : Val$
$ok! \Leftrightarrow u? \in customers$	$v! = bal(a?)$

com_transferReq	com_transferExec
$\Delta(transferOK)$ $s? : Sum$ $a1? : AccID$ $a2? : AccID$ $ok! : \mathbb{B}$	$\Delta(bal)$ $s? : Sum$ $a1? : AccID$ $a2? : AccID$
$ok! \Leftrightarrow a1 \neq a2 \land$ $bal(a1?) - s? \geq 0$ $transferOK' \Leftrightarrow ok!$	$transferOK \Rightarrow$ $bal' = bal \oplus$ $\{a1? \mapsto bal(a1?) - s?,$ $a2? \mapsto bal(a2?) + s?\}$ $\neg transferOK \Rightarrow$ $bal' = bal$

In this example, the (unprotected) target system is $UnpSys = Bank$. To secure this system, we specify a security automaton *SecAut*, whose states record the credentials, privileges, and the TAN (Transaction Authentication Number) lists of its customers. The credentials specify the PIN (Personal Identification Number) code of the customers, the privileges specify which actions (balance check, transfer) a user can perform on a given account, and the TAN list contains the transaction authentication numbers to be used (each only once) for each transfer operation. Formally, the set of actions is specified by

$$Actions :== Balance \mid Transfer\,.$$

After a successful login, the security automaton authenticates the user by checking whether the PIN code entered on the channel *pin* of *SecUserIF* agrees with the user's credentials. When this is the case, the security automaton enters the loop *SecOperate*, where it stays until the user logs out. To check an account's balance, the security automaton checks whether the user has the corresponding privilege to carry out the *Balance* action on the account $a?$, input by the user. To transfer funds, the security automaton requires that the user has the *Transfer* privilege for the account $a1?$ to be debited and moreover it checks whether the TAN entered by the user agrees with the next number on the TAN list.

SecAut

```
chan login, logout, balance, transferReq,
     transferExec, abort                    [types omitted]
chan pin : [ p? : PIN; ok! : 𝔹 ]
chan tan : [ t? : TN; ok! : 𝔹 ]

main = login?u?ok → (ok & Identify
                      □ ¬ok & main)
Identify = pin?p?ok → (ok & SecOperate
                        □ ¬ok & Identify)
SecOperate = balance → SecOperate
           □ transferReq → TanCheckExec
           □ logout → main
TanCheckExec =
     tan?t?ok → (ok & transferExec → SecOperate
                □ ¬ok & abort → SecOperate)
     □ logout → main
```

$credentials : customers \to PIN$ $privileges : \mathbb{P}(UserID \times AccID \times Actions)$ $tanlist : customers \to \mathbb{N} \to TN$ $uid : UserID$ $tid : customers \to \mathbb{N}$

Init
$\forall\, uid : customers \bullet tid(uid) = 0$

com_login	com_pin
$\Delta(uid)$ $u? : UserID$	$p? : PIN$ $ok! : \mathbb{B}$
$uid' = u?$	$ok! \Leftrightarrow (p? =$ $credentials(uid))$

com_balance	com_transferReq
$a? : AccID$	$a1? : AccID$
$(uid, a?, Balance) \in$ $privileges$	$(uid, a1?, Transfer) \in$ $privileges$

com_tan

$\Delta(tid)$
$t? : TN$
$ok! : \mathbb{B}$

$ok! \Leftrightarrow (t? = tanlist(uid)(tid(uid)))$
$(ok! \Rightarrow tid' = tid \oplus \{uid \mapsto tid(uid) + 1\}) \land$
$\quad (\neg ok! \Rightarrow tid' = tid)$

The secure system is specified by

$$SecSys = Bank \ [|\ A\ |] \ SecAut\,.$$

Here $A = \{|\ login, balance, transferReq, transferExec, abort, logout\ |\}$. To the user, a constrained view of the system may be presented, for example, $SecSys \setminus B$, where the operations in $B = \{|\ transferExec, abort\ |\}$ are hidden.

4.2 Formal analysis

Using the transformational semantics of CSP-OZ outlined in Section 2, we can perform a formal analysis of the bank example. First, we perform basic "sanity checks", in particular checking the system against different *use cases* in the form of traces. For example, for suitable choices of the data domains,

$$SecSys \sqsubseteq_{\mathcal{T}} login.u1.true \rightarrow pin.p1.true \rightarrow transferReq.3.ac1.ac2.true \rightarrow STOP$$

describes a partial run of the secure system.

Second, we check several *general properties* of the specifications based on their failures-divergences semantics: *Bank*, *UnpSys*, *SecAut*, and *SecSys* are all *deadlock-free*, *livelock-free*, and *deterministic*.

Third, we check *trace refinement* properties. Note that $UnpSys \not\sqsubseteq_{\mathcal{T}} SecSys$, i.e., the secure system does not refine the unprotected system. This is simply because *SecSys* has new communication capabilities along the channels *pin* and *tan*, which are not part of *UnpSys*. However, if we hide these channels then $UnpSys \sqsubseteq_{\mathcal{T}} (SecSys \setminus \{|\ pin, tan\ |\})$, i.e., the secure system, restricted to a common interface, refines the unprotected system.

Finally, we check two *security properties* of *SecSys*. Informally stated they are:

(1) No *balance* check occurs before a sequence of successful *login* and *pin* entries.

(2) No *transferExec* occurs before a successful *tan* entry.

We formalize these properties using CSP test processes. For example, property (2) is checked by the test process $P2$ specified as follows:

$$\begin{aligned} P2 &= tan?t.true \rightarrow P2T \\ &\quad \Box\ (\ \Box\ x : A2 \bullet x \rightarrow P2) \\ P2T &= transferExec?s?a1?a2 \rightarrow P2 \\ &\quad \Box\ (\ \Box\ x : A2 \bullet x \rightarrow P2) \end{aligned}$$

with $A2 = \{|\ login, balance, transferReq, abort, logout, pin\ |\} \cup \{tan.t.false \mid t \in TN\}$. Then the trace refinement $P2 \sqsubseteq_{\mathcal{T}} SecSys$ shows that (2) holds.

By restricting the bank's data to range over finite domains, we have used FDR to check all of these properties automatically. See Appendix B for details. The third property though does not actually require model checking as it follows directly from Proposition 2.

4.2.1 Transformation

The combination CSP-OZ also allows us to perform semantic preserving transformations. In particular, we can *eliminate* the parallel composition between the target and security automaton by using a transformation akin to Milner's *expansion law* [25]. Applying this transformation to $SecSys0 = Bank \ [|\ A\ |] \ SecAut$ yields a CSP-OZ specification where all communication schemas of *Bank* and *SecAut* with the same name are conjoined. For example, eliding the declaration parts, we obtain

com_balance

...

$val! = bal(a?)$
$(uid, a?, Balance) \in$
$\quad privileges$

com_transferReq

...

$ok! \Leftrightarrow bal(a1?) - s? \geq 0$
$transferOK' \Leftrightarrow ok!$
$(uid, a1?, transfer) \in$
$\quad privileges$

where the unprotected bank operations and security checks concerning the *privileges* appear as conjuncts next to each other.

In general, we can use such transformations to explore different designs, which may reflect different ways of securing systems, e.g., corresponding to different security architectures or security design patterns. This simple example shows the relationship between two common security architectures. The security automaton represents a *policy decision point*. Parallel composition represents a system where decisions about allowed operations are separated from the operations themselves and the *policy enforcement point* is represented implicitly, not by a process, but rather by the operational semantics of synchronous communication. This architecture can be directly refined to a concrete platform with support for *declarative access control*, where, for example, the middleware serves as a policy enforcement point by intercepting method calls and querying the policy decision point to check whether the resulting step is allowed. In contrast, when we transform the specification by eliminating parallel composition, the security checks are moved to, and incorporated in, the respective (previously insecure) operations. This factorization of the policy decision point into the individual methods is in the style of *programmatic access control*, where each method takes responsibility itself for checking whether it is authorized by the security policy.

4.3 Discussion

In a companion paper [34], we have applied CSP-OZ to specifying access control for documents, modeled as attributed trees. The full specification comprises around 25 pages and is given in [33]. While we have not performed any formal analysis of the resulting specification, the case study demon-

strates that our specification approach scales to realistic examples.

The banking example illustrates how we can use different features of CSP-OZ to specify and reason about security automata and their composition with systems. The following characteristics are important for this application.

- The ability to specify event-based and state-based behavior as complementary aspects of the overall system behavior.
- Support for the parallel composition of specifications with synchronization on common events.
- Notions and laws of refinement, which enable correct proofs and system transformation.
- Tool support for reasoning about specifications.

Other specification languages have some of these characteristics, but few possess all of them. One alternative is CSP ‖ B [37], which combines CSP with B machines [1]. Whereas in Object-Z, any Z predicate can be used to specify state transformations, B uses the restricted format of guarded, nondeterministic assignments. Tool support is also available for CSP ‖ B [7]. Another alternative is Circus [39], which combines CSP, Z, and concepts of the refinement calculus [27]. Circus also has a model checker [16], inspired by FDR.

We now return to the *abort* operation in the *Bank* specification and explain the reasons for it and its implications. The inclusion of *abort* is not needed for the functioning of the insecure system but it acts as a "hook" in *Bank* that prevents the secure system from deadlocking. By omitting *abort*, a deadlock can arise: if a *transferReq* is granted but an incorrect *tan* is entered, then *transferExec* blocks. Note that a deadlock would correspond to Schneider's concept of execution cutting, which is acceptable from a security point of view. Indeed, without the *abort* alternative, we still have the trace refinement $UnpSys \sqsubseteq_{\mathcal{T}} (SecSys \setminus \{\mid pin, tan \mid\})$. However, from a usability point of view, we would like our systems to be able to recover from attempted security violations.

In the ideal world, we could design unprotected systems completely independent of security concerns. The above discussion indicates that we can achieve this using execution cutting, provided we are not concerned about deadlocks. But this is usually unacceptable in practice. The alternative we have taken, using *abort*, corresponds to sending a notification to the target system or throwing an exception. But to cause a transfer of control in the target system, the target system must be designed to respond to such notification and this requires building-in appropriate functionality in the target system prior to its composition with the security automaton. This indicates why, in practice, security cannot be completely factored out during system design.

5. RELATED WORK AND CONCLUSIONS

A number of researchers have studied the expressiveness of variants of security automata and alternative enforcement mechanisms. For example, in [24] the authors formalize *edit automata*, which go beyond execution monitoring by allowing run-time program modification. Alternatively, [17] examines the class of policies enforceable by transforming target programs using rewriting. In contrast, our work is not concerned with expressiveness in the sense of enlarging the class of enforceable policies beyond those of security automata. Rather, we provide a foundation that is more expressive than security automata in terms of providing a language that is higher-level, more natural and concise, and which also supports transformation and analysis.

A related research area concerns how to map access control specifications to code that enforces the specified policy. For example, [10] specifies access control policies for Java stack inspection in the policy specification language PSLang. Policies are translated by a trusted rewriter into inline reference monitors by merging checks into the target systems. Other examples are the Polymer system [5] and Naccio [11], both of which produce Java bytecode from security specifications, again inlined into the target system. In contrast, the Ponder Policy Specification Language [8] and the Model-Driven Security approach of [4] map descriptions of access control policies into both procedural access control infrastructures (Java assertions) and declarative access control infrastructures (RBAC) for middleware platforms.

Each of the above-mentioned approaches (and there are many others) offers its own "domain specific" policy specification language along with some means of associating policies with target systems. Some of these languages, for example Polymer, are quite rich and well suited for modularly specifying and combing policies. These approaches, however, have a rather different focus than ours: mapping policies to security infrastructure as opposed to analyzing and transforming the policies themselves. As these languages are precise, and, in most cases semantically well defined, the formal analysis of policy specifications should be possible. However, this would require concrete work, in each case, to make precise the link from the domain-specific language to the appropriate reasoning theory, notions of refinement and transformational development (where they exist), and tool support.

Security automata are closely related to runtime monitors, which determine whether safety properties are violated at runtime. Research in this direction, e.g., [18, 19], has focused on the generation of efficient monitors from safety properties specified as formulas in Linear Temporal Logic and the integration of monitors in different programming languages. While this approach is well suited for monitoring properties involving simple control patterns, like precedence and bounded response, it is less well-suited for specifying security properties with complex interaction between control and data, like those arising in our document processing example.

Within the process algebra community, synchronous parallel composition is commonly used to specify systems by building up constraints on traces. Namely if P and Q are processes, then $P \ [|\ A\ |]\ Q$ specifies a process where P may only carry out those events in A permitted by Q. This was the starting point for our work. Phrased in the terminology of security automata, Q describes the security automaton and synchronous parallel composition combines the

processes into one that uses execution cutting for policy enforcement. Our contributions have been to make this correspondence exact for a rich specification language, where we build upon existing concepts, results, and tools, to show that the result can be applied to realistic examples, and to highlight some of the advantages in proceeding this way.

Our focus in this paper has been on the specification, the first stage (after requirements analysis) in system development. One direction for future work is to explore the next stage: the link to code. In contrast to the use of direct mappings, like those discussed above, we have the possibility of using refinement to establish a formal link between CSP-OZ specifications at different levels of detail. In particular, by introducing more components that run in parallel one can refine towards a distributed implementation. For a predecessor of the CSP-OZ notation, this has been shown in [29]. To proceed to an actual implementation, the approach of [26] could be pursued where CSP-OZ specifications are linked to a Java implementation by generating (as an intermediate step) assertional contracts expressed in the Java Modeling Language (JML) [23] from the specification. These contracts can then be satisfied by a Java program implementing the specification. Taken together, this would provide a formal framework for carrying out the development and analysis of security-critical systems, as well as for making a formal link to code.

Another direction, which is particularly interesting from the security standpoint, is to use the transformational approach suggested to explore the relationships between different security architectures. We have provided a simple example of how this can be done to relate declarative and programmatic access control. Other studies can be undertaken here, for example ways of distributing monitoring between clients and servers for different classes of distributed applications.

6. REFERENCES

[1] J. Abrial. *The B-Book — Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] B. Alpern and F. Schneider. Defining Liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[3] B. Alpern and F. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2:117–126, 1987.

[4] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: from UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, January 2006.

[5] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 305–314, June 2005.

[6] S. Brookes, C. Hoare, and A. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31:560–599, 1984.

[7] M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Formal Methods 2005*, volume 3582 of *LNCS*. Springer, 2005.

[8] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, London, UK, 2001. Springer-Verlag.

[9] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[10] Ú. Erlingsson and F. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[11] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[12] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 2, pages 423–438. Chapman & Hall, 1997.

[13] C. Fischer. *Combination and Implementation of Processes and Data: From CSP-OZ to Java*. PhD thesis, Bericht Nr. 2/2000, University of Oldenburg, April 2000.

[14] C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *Integrated Formal Methods*, pages 315–334. Springer, 1999.

[15] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement – FDR 2 User Manual*, May 2003.

[16] L. Freitas. *Model Checking Circus*. PhD thesis, University of York, October 2005.

[17] K. Hamlen, J. Morrisett, and F. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[18] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Software Tools for Technology Transfer*, 6(2):158–173, 2004.

[19] K. Havelund and G. Rosu. An overview of the runtime verification tool java pathexplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[20] C. Hoare. Communicating sequential processes. *CACM*, 21:666–677, 1978.

[21] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[22] International Organization for Standardization. *Information technology – Z formal specification notation – Syntax, type system and semantics*, first edition, July 2002.

[23] G. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. Cok. How the design of JML accomodates both runtime assertion checking and formal verification. *Science Computer Programming*, 55:185–208, 2005.

[24] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, 2005.

[25] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[26] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java:

A Case Study. In E. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods*, volume 2999 of *LNCS*, pages 267–286. Springer, 2004.

[27] C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[28] E.-R. Olderog and C. Hoare. Specification-oriented semantics for communicating processes. *Acta Inform.*, 23:9–66, 1986.

[29] E.-R. Olderog and S. Rössig. A case study in transformational design of concurrent systems. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Theory and Practice of Software Development*, volume 668 of *LNCS*, pages 90–104. Springer, 1993.

[30] A. Roscoe. Model-checking CSP. In A. Roscoe, editor, *A Classical Mind – Essays in Honour of C.A.R. Hoare*, pages 353–378. Prentice-Hall, 1994.

[31] A. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall, Englewood Cliffs, NJ, first edition, 1998.

[32] F. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[33] P. E. Sevinç and D. Basin. Controlling access to documents: A formal access control model. Technical report, ETH Zurich, 2006.

[34] P. E. Sevinç, D. Basin, and E.-R. Olderog. Controlling access to documents: A formal access control model. In G. Müller and G. Schneider, editors, *Proc. of ETRICS 2006*, volume 3995 of *LNCS*, pages 352–367. Springer, 2006.

[35] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.

[36] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.

[37] H. Treharne and S. Schneider. Communicating B machines. In *ZB2002: International Conference of Z and B Users*, volume 2272 of *LNCS*. Springer, 2002.

[38] G. von Bochmann and J. Gecsei. A unified method for the specification and verification of protocols. In *IFIP Congress*, pages 229–234, 1977.

[39] J. Woodcock and A. Cavalcanti. The semantics of Circus. In *ZB2002: International Conference of Z and B Users*, volume 2272 of *LNCS*, pages 184–203. Springer, 2002.

[40] J. Woodcock and J. Davies. *Using Z — Specification, Refinement, and Proof*. Prentice-Hall, 1996.

APPENDIX

A. TRANSFORMING CSP-OZ INTO CSP

Recall from Section 2 that a CSP-OZ class C is transformed into a process $proc(C) = \texttt{main} \ [|\ Ev\ |]\ OZMain$. Here we complete this account by describing $OZMain$.

$OZMain$ is defined by recursive process equations using replicated versions of the CSP operators $\sqcap$ and $\Box$. The OZ part is represented by a loop $OZPart(st)$ that, at each iteration, is ready for communication along some channels. Which channel c is ready and its effects depend on the schema $\texttt{com\_}c$. In the definition, we use the following notation. Let st denote the list of variables declared in the *State* schema of C, and let st' be the corresponding list of primed variables. For a channel c declared in IF, let in_{op}, out_{op} denote the list of input and output parameters of this operation. Then the schema $\texttt{com\_}c$ depends on the values of st, in_c, out_c, st'. For a given list ℓ of typed variables, let $Val(\ell)$ denote the set of all corresponding lists of values that these variables may assume based on their type.

$$
\begin{aligned}
OZMain \quad &= \quad \bigsqcap v\_st : \{v\_st : Val(st) \mid Init\} \bullet \\
&\qquad OZPart(v\_st) \\
OZPart(st) \quad &= \quad \Box(c,\ v\_in_c) : \{c : IF;\ v\_in_c : Val(in_c) \mid \\
&\qquad \exists\, out_c;\ st' \bullet \texttt{com\_}c\} \bullet (\ \bigsqcap(v\_out_c,\ v\_st') : \\
&\qquad \{v\_out_c : Val(out_c);\ v\_st' : Val(st') \mid \\
&\qquad \texttt{com\_}c\} \bullet c.v\_in_c.v\_out_c \rightarrow OZPart(v\_st'))
\end{aligned}
$$

By the first equation, the process $OZMain$ can nondeterministically choose any values v_st for the state variables st that satisfy $Init$ to start with. For the $\Box$ operator in the second equation, c ranges over all channels declared in IF and v_in_c ranges over all value lists in $Val(in_c)$ such that the precondition of the communication schema for c, i.e. $\exists\, out_c;\ st' \bullet \texttt{com\_}c$, holds for these values. Finally, for any chosen c and values v_in_c, the subsequent $\sqcap$ operator is determined as follows: v_out_c ranges over all value lists in $Val(out_c)$ and v_st' ranges over all value lists in $Val(st')$ such that $\texttt{com\_}c$ holds for these values. So the $OZPart(v\_st)$ is ready for every communication event $c.v\_in_c.v\_out_c$ along a channel c in IF where for the values v_in_c the communication schema $\texttt{com\_}c$ is satisfiable for some output values v_out_c and successor state v_st'. For given input values v_in_c, any such v_out_c and v_st' can be nondeterministically chosen to yield $c.v\_in_c.v\_out_c$ and the next recursive call $OZPart(v\_st')$. Thus input and output along channels c are modeled by the subtle interplay of the CSP alternative and nondeterministic choice.

B. BANK EXAMPLE IN FDR

What follows is the entire input script for the bank example in the concrete syntax of FDR. Included in this script are the various properties that have been checked.

```
    m   = \ x,S @ member(x,S)
    -- abbreviation for membership function
-- Definitions of constants
    datatype UserID = u1 | u2 | u3
    -- concrete set of user ids
    datatype AccID = ac1 | ac2
    -- concrete set of accounts
    Val = {(-6)..6}
    -- concrete set of values accounts may assume
    Sum = {1..6}
    -- concrete set of sums customers may transfer
-- CSP Part Bank
    channel login: UserID.Bool
    channel logout
    channel balance: AccID.Val
    channel transferReq:  Sum.AccID.AccID.Bool
    channel transferExec: Sum.AccID.AccID
    channel abort
    mainB = login?u?ok -> (ok & Operate
                           [] not ok & mainB)
```

```
    Operate = (balance?a?v -> Operate
              [] transferReq?s?a1?a2?ok ->
                   (transferExec!s!a1!a2 -> Operate
                    [] abort -> Operate)
              [] logout -> mainB)
-- OZ Part Bank
-- We represent the current balance bal as a set of
-- pairs (account-id, value). This requires some
-- auxiliary functions defined below:
    ValSet   = \b,a @ { v | v <- Val, m((a,v),b) }
    pick({x}) = x
    PickVal   = \b,a @ pick(ValSet(b,a))
    withdrawOK = \b,a1,a2,s @
                 not(a1==a2) and
                 (PickVal(b,a1) - s >= 0)
    upd = \b,a,v @
          let
            bminus = diff(b,{(a,vold) | vold <-Val })
          within
            union(bminus, {(a,v)})
-- The set of customers is defined as a concrete
-- subset of UserID. It appears as a global parameter
-- of the process OZB.
    cust = {u1, u2}
    OZB(bal,transferOK) =
    (m(bal,Set({(a,v)|a<-AccID,v<-Val})) and
     m(transferOK,Bool)) &
     (([] (u,ok): {(u,m(u,cust)) | u <- UserID } @
          login.u.ok  -> OZB(bal,transferOK))
      [][] (a,v) :
            {(a,PickVal(bal,a)) |
                  a <- AccID, card(ValSet(bal,a))==1} @
            balance.a.v -> OZB(bal,transferOK))
      [][] (s,a1,a2,ok):
            {(s,a1,a2,withdrawOK(bal,a1,a2,s)) |
                  s<-Sum, a1<-AccID, a2 <-AccID,
                  card(ValSet(bal,a1)) == 1 } @
            transferReq.s.a1.a2.ok -> OZB(bal,ok))
      [] transferExec?s?a1?a2 ->
          if transferOK and card(ValSet(bal,a1))==1 and
             card(ValSet(bal,a2))==1
          then
            let
              v1 = PickVal(bal,a1) - s
              v2 = PickVal(bal,a2) + s
            within
              OZB(upd(upd(bal,a1,v1),a2,v2), transferOK)
          else OZB(bal,transferOK)
      )
-- Parallel Composition of CSP and OZ part of the Bank
-- starts with the following initial balance of the
-- accounts:
    bal = { (ac1,3), (ac2,-2) }
    Bank =
      mainB
        [|{| login,balance,transferReq,transferExec |}|]
      OZB(bal,false)
-- Unprotected System
    UnpSys  = Bank
-- SecAut
    datatype Actions = Balance | Transfer
    datatype PIN = p1 | p2
    -- concrete set of pins
    datatype TN  = t1 | t2 |t3
    -- concrete set of tans
```

```
-- CSP Part SecAut
    channel pin: PIN.Bool
    channel tan: TN.Bool
    mainS = login?u?ok -> (ok & Identify
                           [] not ok & mainS)
    Identify = pin?p?ok -> (ok & SecOperate
                            [] not ok & Identify)
    SecOperate =
      balance?a?val -> SecOperate
      [] transferReq?s?a1?a2?ok -> TanCheckExec
      [] logout -> mainS

    TanCheckExec =
      tan?t?ok -> (ok & transferExec?s?a1?a2 ->
                          SecOperate
                  [] not ok & abort ->
                          SecOperate)
      [] logout -> mainS

-- OZ Part SecAut
-- The following definitions appear as
-- global parameters of the process OZS:
    priv = { (u1,ac1,Balance),
             (u1,ac1,Transfer),
             (u2,ac2,Balance),
             (u2,ac2,Transfer),
             (u3,ac1,Balance),
             (u3,ac1,Transfer),
             (u3,ac2,Balance) }
    -- concrete set of privileges
    cred(u1) = p1
    cred(u2) = p2
    -- concrete set of credentials
    N = 2
    -- N+1 is the concrete length of the tanlist
    tanlist(u1,0) = t1
    tanlist(u1,1) = t3
    tanlist(u1,2) = t2
    tanlist(u2,0) = t1
    tanlist(u2,1) = t2
    tanlist(u2,2) = t3
    -- concrete tanlist
    tid0 = { (u1,0), (u2,0) }
    -- initial tan indices

    OZS(uid,tid) =
    ( m(uid,UserID) and m(tid,Set({(u,v) |
       u <-cust, v<-{0..N} }) ) ) &
     (login?u?ok -> OZS(u,tid)
      [][] (p,ok): {(p,m(uid,cust) and p == cred(uid)) |
            p <- PIN } @ pin.p.ok -> OZS(uid,tid))
      [][] a: {a | a <- AccID,
                   m((uid,a,Balance),priv)} @
           balance.a?v -> OZS(uid,tid))
      [][] a1: {a1 | a1 <- AccID,
                     m((uid,a1,Transfer),priv)} @
           transferReq?s.a1?a2?ok -> OZS(uid,tid))
      [][] (t,ti,ok): {(t,ti,t == tanlist(uid,ti)) |
                       t <- TN, ti <- {0..N},
                       card(ValSet(tid,uid)) == 1,
                       ti == PickVal(tid,uid) } @
           tan.t.ok -> ( (ok and (ti < N)     &
                          OZS(uid,upd(tid,uid,ti+1))
                         [](not ok or (ti == N) &
                          OZS(uid,tid)) ) )
     )
```

```
-- Parallel Composition of CSP and OZ part of the SecAut
    SecAut =
      mainS
        [|{| login,pin,balance,transferReq,tan |}|]
      OZS(u3,tid0)
-- Secure System
    A = {| login, balance, transferReq, transferExec,
           abort, logout |}
    SecSys = Bank [| A |] SecAut
    -- has alphabet A union {| pin, tan |}

-------------------------------------------------------
-- Individual Traces (Use Cases): two examples
-------------------------------------------------------
    assert SecSys [T= login.u1.true -> pin.p1.true ->
                      transferReq.3.ac1.ac2.true ->
                      tan.t1.true ->
                      transferExec.3.ac1.ac2 -> STOP
    -- satisfied
    assert SecSys [T= login.u1.true -> pin.p1.true ->
                      transferReq.3.ac1.ac2.true ->
                      tan.t2.false ->
                      transferExec.3.ac1.ac2 -> STOP
    -- not satisfied: tan t2 is false
-------------------------------------------------------
-- General Properties
-------------------------------------------------------
-- Deadlock
--   Bank, UnpSys, SecAut, SecSys are all
--   deadlock free.                          -- checked
-- Livelock (Divergence)
--   Bank, UnpSys, SecAut, SecSys are all
--   livelock free.                          -- checked
-- Determinism
--   Bank, UnpSys, SecAut, SecSys are all
--   deterministic.                          -- checked
-------------------------------------------------------
-- Refinement Properties
-------------------------------------------------------
    assert UnpSys [T= SecSys
    -- not satisfied due to pin and tan
    assert UnpSys [T=  SecSys \ {|pin, tan |}
    -- checked for values up to 6
-------------------------------------------------------
-- Security Properties
-------------------------------------------------------
-- No balance check before a sequence of successful
-- login and pin, belonging to the credentials of
-- the user.
    A1 = {| transferReq, transferExec, abort, tan |}
    P1 = ([] u : { u | u <- UserID,
                       member(u,cust) } @
              login.u.true -> P1L(u))
         [](] u : { u | u <- UserID,
                        not member(u,cust) } @
               login.u.false -> P1)
         [] logout -> P1
         [] ([] x : A1 @ x -> P1)
    P1L(u) =
         ([] p : { p | p <- PIN, p == cred(u) } @
             pin.p.true -> P1LP)
         [](] p : { p | p <- PIN,
                        not(p == cred(u)) } @
               pin.p.false -> P1L(u))
         [] logout -> P1
         [] ([] x : A1 @ x -> P1L(u))
    P1LP = balance?a?v -> P1LP
           [] logout -> P1
           [] ([] x : A1 @ x -> P1LP)
    assert P1 [T= SecSys
    -- satisfied
-- No transferExec before a successful tan.
    A2 = union({| login, balance, transferReq,
                  abort, logout, pin |},
               { tan.t.false | t <- TN })

    P2 = tan?t.true -> P2T
         [] ([] x : A2 @ x -> P2)
    P2T = transferExec?s?a1?a2 -> P2
          [] ([] x : A2 @ x -> P2)
    assert P2 [T= SecSys
    -- satisfied
```