PREREQUISITE CONFIDENTIALITY

John P. Nestor and E. S. Lee

Computer Systems Research Institute
University of Toronto
6 King's College Road
Toronto, Ontario
Canada M5S 1A4

{nestor,stew}@hub.utoronto.ca

ABSTRACT

We introduce a new definition of confidentiality. It is demonstrated that this new definition, called prerequisite confidentiality, is more effective than previous definitions.

We have developed a modelling scheme that is based upon event systems in order to study prerequisite confidentiality. The structure of the event traces is captured by formal languages and grammars. This provides a convenient and mathematically well-founded means for dealing with component specifications. The externally visible behaviour of a component, including causal relationships between events, and possible nondeterminism, is successfully modelled using the approach. It is then possible to restrict the grammatical specification in such a way that the desired confidentiality property is satisfied. Since all of the grammars used in the specification technique fall into a particular class, we show that it is always possible to construct a recognizer that can be used to identify valid event sequences or determine whether an event sequence satisfies a desired property.

1 INTRODUCTION

When the first computer systems were designed and built, it was relatively easy to verify that the systems had all of the properties intended by their designers. One could merely exercise a system through manual testing. As the complexity of the systems increased, and as different types of systems were interconnected, a convincing demonstration that a system had certain properties became difficult. Researchers turned to a variety of formal modelling techniques to describe systems and the desired properties. These techniques were often based on mathematical proofs that were used to increase confidence that the system goals were met.

CCS '94- 11/94 Fairfax Va., USA

No modelling strategy can provide complete assurance that a real system behaves in a certain way. The act of modelling itself involves a variety of abstractions of the real system in order to create a workable model. It remains possible that an abstraction made for modelling purposes will conceal some detail that will allow for invalid operation when the modelled system is implemented. The key is to show that the degree of abstraction chosen is appropriate and does not adversely affect the result of the exercise.

The first step in developing a good modelling technique for computer systems is to create a representation that reflects how a system is "put together" from its basic components. The second step is to create a means of describing the way that these components communicate or otherwise interact with each other and the surrounding environment.

In this paper, we introduce a new definition of confidentiality. It is demonstrated that this new definition, called prerequisite confidentiality, is more effective than previous definitions.

We begin our description of prerequisite confidentiality by introducing a modelling scheme that is based upon event systems. The structure of the event traces is captured by formal languages and grammars. This provides a convenient and mathematically well-founded means for dealing with component specifications. The externally visible behaviour of a component, including causal relationships between events, and possible nondeterminism, is successfully modelled using the approach. It is then possible to restrict the grammatical specification in such a way that the desired confidentiality property is satisfied. Since all of the grammars used in the specification technique fall into a particular class, we show that it is always possible to construct a recognizer that can be used to identify valid event sequences or determine whether an event sequence satisfies a desired property.

Given such a representation, the notion of prerequisite confidentiality can easily be expressed and its composability properties investigated and proven.

The remainder of this section introduces event systems and the grammatical class that we will employ in our specification. Section 2 introduces the component specification method, and presents several sample components. Section 3 considers some decidability issues. In Section 4, a general notion of properties is introduced. Section 5 then considers prerequisite confidentiality as an example of such a property. Section 6 compares prerequisite confidentiality with several existing approaches. Finally, Section 7 provides a summary and some conclusions.

1.1 EVENT SYSTEMS

One way of representing the behaviour of a system uses event systems. In this approach, the significant happenings in the modelled system are represented by a time-ordered sequence of events. The event system then describes the set of all valid event sequences.

The modeller is free to choose those events that are of interest and ignore those that are not relevant to a particular analysis. What is considered "important" depends on the particular situation being analyzed.

In addition, event systems allow for the representation of computer and other "real world" systems at a wide range of levels of detail. For example, events might map to the change of digital levels on IC pins, or to the execution of individual statements in a C program, or to the invocation of an entire program. The choice as to the appropriate level of detail depends on the goal of the particular modelling effort being undertaken.

In such a representation, the absolute timing of a particular event is not important, but its timing relative to the occurrence of other events in the sequence is significant. This relativistic timing notion corresponds closely to the practical operation of most real computing systems. For example, a typical C program running on a multi-user system is usually guaranteed to execute its statements in a particular order[1], but the absolute timing of the execution of each statement usually does not greatly concern the user. If one wishes to introduce the notion of absolute timing into the event trace formalism, this can generally be done by creating "clock-tick" events.

Another way of describing the relativistic timing of event traces relates to the possible "real" executions that such traces capture. An event trace places no constraints on the absolute timing of particular events. All executions with the same relative ordering of events are captured and represented by a single event sequence. Thus, when we consider the properties of an event sequence, we are really considering the properties of a whole class of executions. This is represented graphically in Figure 1. In this case, the labelled circles represent different events, and their absolute timing displayed on the scale from left to right. The first three sequences have identical event trace representations, "A, B, C, B", while the fourth sequence differs.

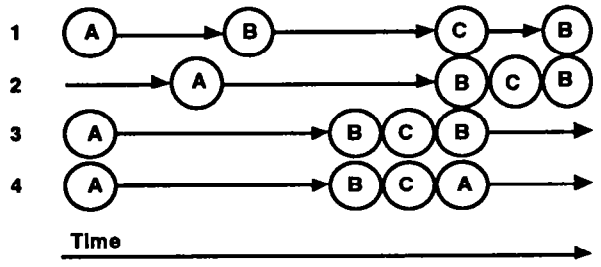

Figure 1: Relativistic Timing of Event Traces

One advantage of the event trace approach is that it describes a system based only upon the externally visible behaviour. As a result, it does not implicitly constrain the internal behaviour of the modelled system in the same way that other methods (such as state machines) sometimes do.

There has been fairly extensive research in the area of event (or trace-based) systems. Many of these are based upon Hoare's notion of Communicating Sequential Processes (CSP) [Hoa78]. In contrast with these approaches, we wish to develop a representation for event systems where the primary goal is that it must be possible to construct a recognizer (or parser) that can determine whether a given event sequence is valid. This will be accomplished through the use of formal languages and grammars.

1.2 FORMAL LANGUAGES AND GRAMMARS

Formal languages and grammars form the basis of our event-system model. This section will very briefly introduce some of the standard definitions and notation.[HU69,HU79]

A *grammar* G is denoted by the tuple $\langle V_N, V_T, P, S \rangle$. The symbols V_N, V_T, P, and S are, respectively, the *non-terminal symbols*, *terminal symbols*, *productions*, and *start symbol*.

The union $V_N \cup V_T$ is often denoted by V. Let V^* denote the Kleene closure of V and $V^+ = V^* \setminus \{\varepsilon\}$. The set of productions P consists of expressions of the form $\alpha \rightarrow \beta$, where α is a string in V^+ and β is a string in V^*. Finally, S is always a symbol in V_N.

We now consider the language that a grammar $G = \langle V_N, V_T, P, S \rangle$ generates. We first need to define the

[1] This would not be the case with programs coded in a language that supports parallel execution.

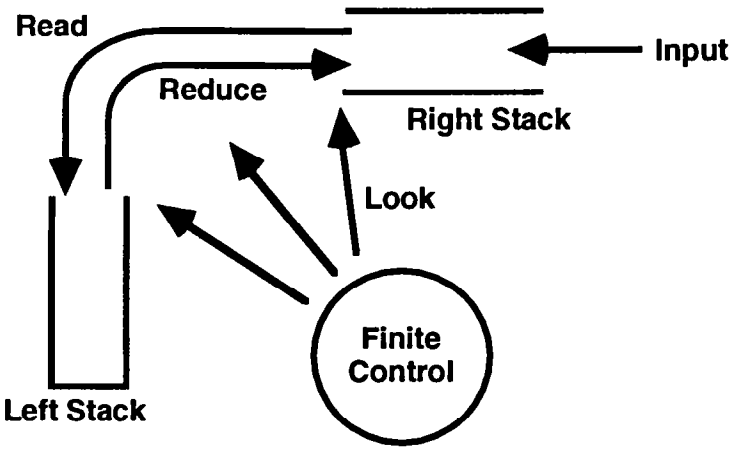

Figure 2: Two Stack Machine

relations $\underset{G}{\Rightarrow}$ and $\overset{*}{\underset{G}{\Rightarrow}}$ between strings in V^*. Specifically, if $\alpha \to \beta$ is a production of P and γ and δ are any strings in V^*, then[2] $\gamma\alpha\delta \underset{G}{\Rightarrow} \gamma\beta\delta$. We say that the production $\alpha \to \beta$ is applied to the string $\gamma\alpha\delta$ to obtain $\gamma\beta\delta$. Thus $\underset{G}{\Rightarrow}$ relates two strings exactly when the second is obtained from the first by the applications of a single production.

Suppose that $\alpha_1, \alpha_2, \ldots, \alpha_m$ are strings in V^*, and $\alpha_1 \underset{G}{\Rightarrow} \alpha_2, \alpha_2 \underset{G}{\Rightarrow} \alpha_3, \ldots, \alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m$. Then we say[3] $\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_m$. We say that for two strings α and β that $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ if we can obtain β from α by application of some number of productions in P. By convention, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$ for each string α.

We define the *language generated by* G (denoted by $L(G)$) to be $\left\{ w \mid w \text{ is in } V_T^* \text{ and } S \overset{*}{\underset{G}{\Rightarrow}} w \right\}$. That is, a string is in $L(G)$ if and only if the string consists solely of terminals and can be derived from S. Two grammars G^1 and G^2 are *equivalent* if $L(G^1) = L(G^2)$.

[2]Say $\gamma\alpha\delta$ *directly derives* $\gamma\beta\delta$ in grammar G.

[3]Say α_1 *derives* α_m in grammar G.

1.3 TURNBULL'S PARSING

In 1974, Turnbull proposed a new hierarchy of languages, defined in terms related to formal languages and their equivalent finite automata.[Tur74,TL79] The classes of grammars were defined according to the ease by which sentences generated by the grammars could be parsed. Instead of basing the classes in terms of the form of rewriting rules, the new classes were defined in terms of constraints on the form and structure of canonical derivations.

To this end, Turnbull developed a new parser model called a *deterministic two stack machine* or D2SM, that is displayed in Figure 2.

The finite control is always in one of several possible states. Initially, the input is pushed into the Right Stack (which acts as an output restricted queue) and the control is in its initial state. The Left Stack is initially empty. At each step in its operations, the two stack machine may perform one of the following operations:

- *Read* — remove a symbol from the left end of the Right Stack, push that symbol into the Left Stack and change state.
- *Look ahead* — change state after looking at a bounded number of symbols at the left end of the Right Stack.
- *Reduce* — pop some symbols from the Left Stack, push some symbols into the left end of the Right Stack, and change state. The new symbols in the Right Stack have replaced the popped symbols and may be read or looked at in future steps.

- *Accept* — if the Left Stack is empty and the Right Stack contains the goal symbol, then accept the input.
- *Reject* — if none of the above moves can be made, the reject the input as being erroneous.

The D2SM model is an extension of the $LR(k)$ model[Knu69] and has all of the standard properties of $LR(k)$ parsers. Such parsers:

1. halt on all inputs and accept exactly those strings that are generated by the grammar;
2. use a parsing algorithm that is deterministic and requires no backtracking;
3. perform a single left to right scan of the input; and,
4. detect errors as soon as possible in the parse.

Turnbull's approach to subdividing the class of type-0 grammars[4] led to the development of the deterministic regular parsable (DRP) class. It was proven that DRP languages were a proper superset of $LR(k)$ languages and suggested that DRP languages form a broad subset of type-0 languages. Turnbull proved that for any D2SM, a corresponding DRP grammar exists. He also demonstrated the converse. Turnbull then went on to describe a parser generator algorithm for DRP grammars, and showed that many of the techniques for handling $LR(k)$ grammars were applicable to DRP grammars.

2 COMPONENT REPRESENTATION

This section will introduce our component representation. We will employ grammars that are members of Turnbull's DRP class. The representational framework for components will be described first, followed by a number of component examples.

2.1 REPRESENTATIONAL FRAMEWORK

The first step in developing our modelling methodology is to present a technique for representing the basic components of a system. These components are conceptual objects that might map directly to parts of a real system (like hardware or software components) or might be related to abstract requirements placed on the system. The chosen modelling technique should provide a mathematically tractable means for dealing with the complexity of an event system representation of components and systems.

We will use the term *component* for each representational element in our modelling methodology. Viewed from the external environment, a component has the ability to accept input events in several distinct streams, and to generate several distinct streams of output events. When components are interconnected, the outputs of one component may become the input of another. Unattached streams of inputs come from the environment, while unattached streams of outputs are sent to the environment.

A component operates by accepting one or more *input events*, and possibly responding by generating one or more *output events*. Conceptually, input events to a component originate in the environment or from another component. Output events are generated by the component and sent somewhere else (either to another component or to the external environment). Each stream of input events will be called a *component input*, and each stream of output events will be called a *component output*. In modelling a particular system, the component inputs and outputs may not necessarily correspond to physical channels. The occurrence of certain input or output events might denote the completion of abstractly-defined operations or requirements.

Each component in our representation will have an associated *component grammar*. Sentences in the language generated from the component grammar will be valid event sequences formed from a properly ordered succession of input and output events for the specified component.

We will use the term *complete transaction* to describe a complete set of input events and the corresponding, causally related output events at a particular component. A *partial transaction* is one in which output events are pending, or in which only a partial set of inputs has been provided to a component. Our component grammars will generate sentences (i.e., valid event sequences) that are made up of zero or more complete transactions. We denote those grammatical productions that specify complete transactions as *transaction productions*.

A component may be composed of a number of causally independent substructures referred to as *blocks* (see Figure 3). A block can be thought of as a sub-component that has been composed with another sub-component, with no connections between them. Each block receives input events and generates output events independently of other blocks within the component. The events that make up a complete transaction are specific to each block. In this sense, the activities of one block have no effect whatsoever on the activities of another block within the component.

Although we consider the component to be the smallest, stand-alone element in our modelling methodology, it is necessary to maintain a component's block structure in order to allow composition with other components. This is

[4]Type-0 grammars sometimes referred to as *semi-Thue*, *phrase structure*, or *unrestricted* grammars. They are the largest family of grammars in the Chomsky hierarchy.[Cho56,Cho59] All productions are of the form $\alpha \rightarrow \beta$, where α and β are arbitrary strings of grammar symbols, with $\alpha \neq \varepsilon$.

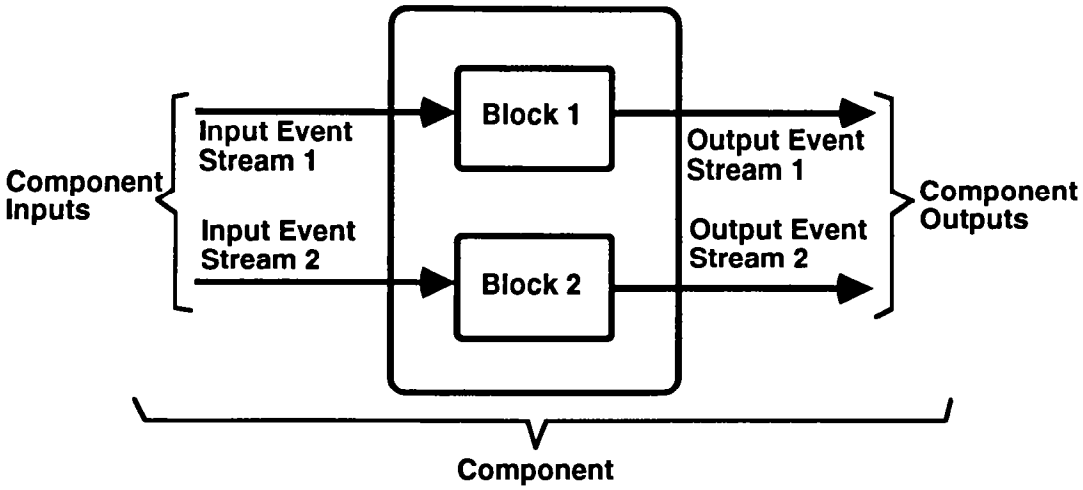

Figure 3: A Component with Two Blocks

described in [Nes93] when the composition procedure is introduced.

In order to describe component properties, we must add some semantic information about the causal relationships between events to the component grammar. This information comes in the form of the binary *prerequisite relation* between input and output events. The relation is used to associate output events with the input events that caused them.

In particular, if $\langle \alpha, \beta \rangle \in prereq$, then it is possible that output event β was caused by the occurrence of input event α. We use the word "possible" because it may be the case that another input event could result in output event β, or that the occurrence of input event α might result in another output event (i.e., not β). There is a relationship between the prerequisite relation and the specification of transaction productions in the component grammar. If two events are related by the prerequisite relation, then there must exist a corresponding transaction production that couples them.

We are concerned with what is possible because, when we consider properties in Section 4, it will be necessary to constrain input/output events wherever it is possible that a particular input event caused a particular output event.

The decision as to what event pairs to place in the *prereq* relation is based upon the functional behaviour of the component being modelled. Several examples are presented in Section 2.2 to illustrate this process.

The formal definition of the above specification method can be found in [Nes93]. The result is a grammar G that generates a language whose sentences are each valid event sequences for the component. By specifying the grammar G and the prerequisite relation, we fully describe the component under the representation.

We have developed a method for composing the specifications of two or more components. Given, for example, grammars G_1 and G_2, prerequisite relations $prereq_1$ and $prereq_2$, and an interconnection specification, we can easily generate the grammar for the composite system, G_C, and the composite prerequisite relation $prereq_C$. In Section 4, this ability will allow us to consider properties of components and to prove that those properties are maintained under composition.

2.2 COMPONENT EXAMPLES

This section will take the general representation framework described in Section 2.1 and instantiate it for a number of particular component examples. The examples are not exhaustive, but are presented to demonstrate the breadth of processing elements that can be represented under this framework. The grammars used in the specifications are simplified versions of those described in [Nes93], but do demonstrate the effectiveness of using grammars to represent component behaviour.

2.2.1 BASIC COMPONENT

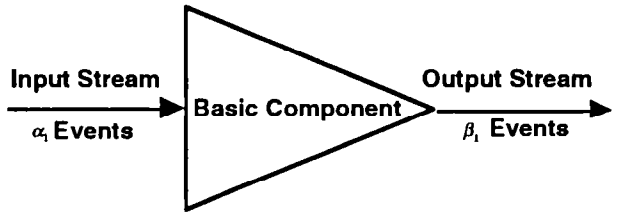

Figure 4: Basic Component

The Basic Component represents one of the simplest elements available in the abstract representational structure. A single input event, α_1, triggers its operation. After a possible processing delay, a corresponding output event, β_1, is generated. This delay may be a function of the specific input event provided as input to the component. As a result, it is possible to queue input events in a Basic Component, while awaiting the occurrence of the corresponding output. Section 2.2.3 will provide an example demonstrating how such queuing can be limited if an input buffer size restriction exists.

Because of its general nature, a Basic Component may easily be used to represent a wide variety of real processing elements. For example, an input event might represent the triggering of the start of a mathematical calculation, and an output event indicates that the operation had completed.

A possible grammatical specification for the Basic Component is as follows:[5]

$$G = \langle V_N, V_T, P, S \rangle$$
$$V_N = \{S, T, Y, Z\}$$
$$V_T = \{\alpha_1, \beta_1\}$$
$$P = \left\{ \begin{array}{l} S \rightarrow TS \\ S \rightarrow \varepsilon \\ T \rightarrow YZ \\ Y \rightarrow \alpha_1 \\ Z \rightarrow \beta_1 \\ ZY \rightarrow YZ \end{array} \right\}$$
$$prereq = \{\langle \alpha_1, \beta_1 \rangle\}$$

The component is defined to have a single input, a single output, and a single block.

The productions generate sequences with zero or more complete transactions, in which input events (denoted by α_1) are immediately followed by their corresponding output event (denoted by β_1). The prerequisite relation reflects the dependence of β_1 events on α_1 events.

The following event sequences are examples of sentences in the language generated by the production set without the "delay" production $ZY \rightarrow YZ$:

$$\varepsilon$$
$$\alpha_1\beta_1$$
$$\alpha_1\beta_1\alpha_1\beta_1\alpha_1\beta_1$$
$$\alpha_1\beta_1\alpha_1\beta_1\alpha_1\beta_1\alpha_1\beta_1\alpha_1\beta_1$$

Next we consider the impact of the delay production. Because of the possibility that the output event corresponding to a particular input event may be delayed until after an arbitrary number of new input events have occurred, a special production is required. In effect, the (context-sensitive) delay production $ZY \rightarrow YZ$ allows output events to "percolate" past the following input events. This makes sequences like the following possible:

$$\alpha_1\alpha_1\beta_1\beta_1\alpha_1\beta_1\alpha_1\alpha_1\alpha_1\beta_1\beta_1\beta_1$$
$$\alpha_1\alpha_1\alpha_1\alpha_1\beta_1\alpha_1\beta_1\beta_1\alpha_1\beta_1\alpha_1\beta_1\beta_1\beta_1.$$

The grammar with the delay production guarantees sentences in which the number of output events is always equal to the number of input events. With this restriction on the event sequences, we have removed the possibility of *spontaneous events*. Such events are generated by the component itself, without any stimulus from the external environment (via the input channel). Since a single input event always occurs before each single output event, spontaneous output events are not possible in this particular component.

The prerequisite relation $prereq = \{\langle \alpha_1, \beta_1 \rangle\}$ indicates that event α_1 is a prerequisite for event β_1. In other words, it is possible[6] that the occurrence of an event α_1 causes the occurrence of an event β_1. As noted in Section 2.1, specifying the causal relationship between events in this way will become important when we wish to describe properties such as confidentiality in Sections 4 and 5.

2.2.2 SWITCH COMPONENT

Given a single input event, the Switch Component generates a corresponding output event on either of two output lines. The choice as to which output event is generated is dependent on whether an even or an odd number of toggle events have occurred.

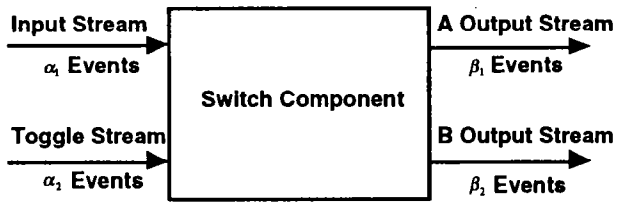

Figure 5: Switch Component

The component has two input streams and two output streams. One input stream is made up of α_1 events, the other is made up of α_2 events. Likewise, there are two output streams, with corresponding events β_1 and β_2. For convenience, we number the streams sequentially, and attach the stream number to the corresponding event

[5] We are using the standard notation for specifying elements of a formal grammar G, as described in [HU69].

[6] In this particular case, it is more than merely possible, it is guaranteed by description of the operation of the basic component.

terminal and non-terminal symbols. For example, α_1 and Y_1 are the terminal and non-terminal symbols for input stream 1.

A possible specification for the Switch Component is as follows:

$$G = \langle V_N, V_T, P, S \rangle$$
$$V_N = \{S, S_1, S_2 T_1, Y_1, Z_1, T_2, Z_2\}$$
$$V_T = \{\alpha_1, \alpha_2, \beta_1, \beta_2\}$$
$$P = \begin{Bmatrix} S \to S_1\alpha_2 S_2\alpha_2 S & S_1 \to T_1 S_1 & S_2 \to T_2 S_2 \\ S \to S_1\alpha_2 S_2 & S_1 \to T_1 & S_2 \to T_2 \\ S \to S_1\alpha_2 & T_1 \to Y_1 Z_1 & T_2 \to Y_1 Z_2 \\ S \to S_1 & Y_1 \to \alpha_1 & Z_2 \to \beta_2 \\ S \to \varepsilon & Z_1 \to \beta_1 & \\ & Z_1 Y_1 \to Y_1 Z_1 & Z_2 Y_1 \to Y_1 Z_2 \end{Bmatrix}$$
$$prereq = \{\langle \alpha_1, \beta_1 \rangle, \langle \alpha_1, \beta_2 \rangle\}$$

The component begins in a state in which it repeatedly accepts α_1 input events and generates β_1 output events. If a α_2 toggle input event is received, the component moves into another state in which each α_1 input event causes a β_2 output event. If a α_2 toggle input event is received, the component moves into back into the first state, where the cycle begins again.

The prerequisite relation $prereq = \{\langle \alpha_1, \beta_1 \rangle, \langle \alpha_1, \beta_2 \rangle\}$ specifies that an α_1 event can cause either a β_1 event or a β_2 event (depending on the state of the component).

2.2.3 LIMITED BUFFER

All the previous examples have allowed the input events to a particular component to queue up indefinitely. This is not a realistic representation of many practical systems, since usually the number of possible pending operations is limited. In this section, we will consider the Basic Component from Section 2.2.1 that has been modified so that it has an input buffer of size $ibs < \infty$. This means that only *ibs* "unprocessed" input events are allowed to queue up at any given time, before the corresponding output events are generated.

A possible specification for the Limited Buffer Component is as follows:

$$G = \langle V_N, V_T, P, S \rangle$$
$$V_N = \{S, S_1, S_2, T_1, Y_1, Z_1, T_2, Y_2, Z_2\}$$
$$V_T = \{\alpha_1, \beta_1\}$$
$$P = \begin{Bmatrix} & S_1 \to T_1 & S_2 \to T_2 \\ & S_1 \to T_1 T_1 & S_2 \to T_2 T_2 \\ & \vdots & \vdots \\ S \to S_1 S_2 S & \overbrace{\quad}^{ibs \text{ times}} & \overbrace{\quad}^{ibs \text{ times}} \\ S \to S_1 & S_1 \to \overbrace{T_1 T_1 \ldots T_1} & S_2 \to \overbrace{T_2 T_2 \ldots T_2} \\ S \to \varepsilon & T_1 \to Y_1 Z_1 & T_2 \to Y_2 Z_2 \\ & Y_1 \to \alpha_1 & Y_2 \to \alpha_1 \\ & Z_1 \to \beta_1 & Z_2 \to \beta_1 \\ & Z_1 Y_1 \to Y_1 Z_1 & Z_2 Y_2 \to Y_2 Z_2 \end{Bmatrix}$$
$$prereq = \{\langle \alpha_1, \beta_1 \rangle\}$$

Here, we have defined a component that effectively has two separate states, each of which operates in an identical manner. The component has a single block. The productions are defined so that up to *ibs* input-output event pairs may occur before the transition is made to the other state.

The delay productions $Z_1 Y_1 \to Y_1 Z_1$ and $Z_2 Y_2 \to Y_2 Z_2$ allow outputs to be delayed, but only within the events of each state. Thus, an output event is never more than *ibs* steps away from the input event that caused it. It can be shown that as *ibs* grows large, the language generated by the limited buffer specification approaches that for the unlimited case described in Section 2.2.1.

3 DECIDABILITY ISSUES

An important consideration in the development of a grammatical representation is the decidability of a number of important issues. It is known that some classes of languages or grammars have most relevant issues decidable, such as the nature of the language that results from language union and intersection. Another useful attribute is the ability to construct a deterministic parser to verify the validity of some event sequence under consideration. Turnbull showed that these issues are decidable for his DRP grammars. Practical and efficient parsers can be constructed to recognize sentences in DRP languages.

In [Nes93], we demonstrated that Turnbull's D2SM was capable of recognizing sentences in the language generated by an arbitrary component grammar G.

Turnbull showed that most questions that are decidable for $LR(k)$ are also decidable for DRP. Thus, it is always possible to construct a practical recognizer that will be able to verify whether an event sequence is valid for the system being modelled.

This ability has important practical implications for the model. The construction of an automated specification and verification tool based on the model would require the ability to quickly and easily verify whether an event sequence was valid for a modelled system. By using grammars from the DRP class, developing an algorithm to do this is straightforward.

4 PROPERTIES

In the most general sense, a *property* is a (desired) quality (of a system) that is relevant to the modelling exercise. In our case, a property places constraints on the way that a component operates. The property may only allow certain types of behaviours to occur. Using our representation, we shall concentrate on the notion of a property as a restriction on the possible event sequences corresponding to a component.

We extend our model by including a set of *event labels* L. Each event terminal and non-terminal symbol in a component grammar is labelled with an element of L. We indicate this label as a left-superscript on the event terminal or non-terminal symbol. For example, an event labelled "Secret" occurring on input stream 1 of a component would have terminal symbol ${}^{\text{Secret}}\alpha_1$. In general, we allow events with different labels to occur on a single input or output stream. For example, the set of events allowed on input stream i can be specified as $\left\{ {}^{s}\alpha_i \middle| s \in L \right\}$.

Given an arbitrary component grammar, a property will restrict the possible combinations of labels that can occur in each transaction for the component. The restrictions will be expressed as relations whose domain is the set of labels. In turn, this will limit the possible label combinations that are possible in valid sentences of the component language.

In general, we have a transaction with a number of input and output events. A *component transaction property* is expressed as a transitive relation between the labels associated with the input events and the labels associated with the output events that make up a particular complete transaction. Formally, the mathematical specification of a property, based on transitive relation ρ, for a component is:

$$\forall s_1, s_2 \in L, w \in \bigcup_{i\in[1,m]} \left\{ {}^{s_1}\alpha_i \right\}, y \in \bigcup_{j\in[1,n]} \left\{ {}^{s_2}\beta_j \right\},$$
$$\langle w, y \rangle \in prereq \Rightarrow s_1 \rho s_2$$

This specification implies that if an input event α is a prerequisite for an output event β, then the labels s_1 and s_2 of events α and β must be such that $s_1 \rho s_2$. If the above holds for a component specification, then we can conclude that the component in question satisfies the desired property.

We have proven that all such component transaction properties are preserved under composition [Nes93].

5 PREREQUISITE CONFIDENTIALITY

In this section we introduce a new confidentiality definition that provides a number of significant improvements over previous definitions. The definition is based upon the grammatical component specification technique described in Section 2, and is an example of a component transaction property, as described in Section 4.

One of the most-studied properties in the area of formal modelling of systems is confidentiality [GM82, GM84, Sut86, McC87, McC88, Lin91]. Originally termed (computer) security, confidentiality has now been classified as one of several ideas involved in an overall notion of computer security.[Par91,Hin93] Confidentiality ensures that information is accessed for observation only by those duly authorized to have such access. Attempts at unauthorized access must be detected and prevented. This simple definition has led to a variety of mathematical formulations of confidentiality, all with varying abilities to capture the behaviour of real systems. Many intuitively reasonable confidentiality properties are not preserved by implementation or by composition.[LBTS92]

We will develop a notion of confidentiality that is based upon Sutherland's deducibility[Sut86]. It will be shown that our new confidentiality definition provides a number of significant improvements over previous definitions.

Informally, our definition of confidentiality, called *prerequisite confidentiality*, is as follows:

> Given an execution sequence with events of varying confidentiality levels, knowledge of low level events in the sequence provides no information about when or if higher level events occur.

We can easily express this confidentiality notion within the component framework by requiring that high-level input events to a component never result in lower level output events being generated. If an input event α is a prerequisite for output event β, then our knowledge that β has occurred might allow us to deduce that event α has (previously) occurred. Therefore, the confidentiality label attached to β must dominate the label of event α, in order to prevent our knowledge of β's occurrence giving us information about higher-level events. In other words, if it is possible that the occurrence of an input event result in the occurrence of an output event, then the required relationship between confidentiality levels of the two events must hold.

Consider a set of partially ordered confidentiality labels L_C with the dominates relation $\rhd_C$. Then a component specification has the prerequisite confidentiality property if and only if

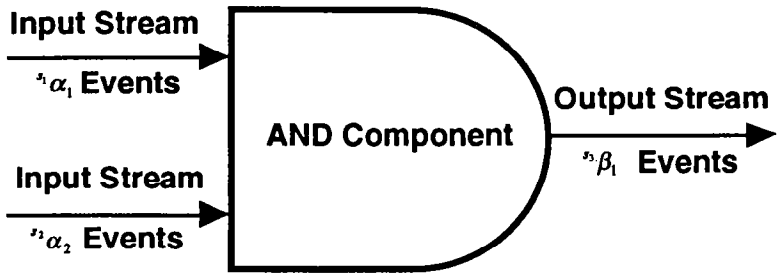

Figure 6: AND Component with Prerequisite Property

$$\forall s_1, s_2 \in L_C, w \in \bigcup_{i\in[1,m]}\{^{s_1}\alpha_i\}, y \in \bigcup_{j\in[1,n]}\{^{s_2}\beta_j\},$$
$$\langle w, y\rangle \in prereq \Rightarrow s_2 \rhd_C s_1$$

This requires that the confidentiality level of the output events dominate the confidentiality level of any prerequisite input events.

As an example, Figure 6 shows an AND Component, with the prerequisite confidentiality property imposed. Each of the input and output events has an attached confidentiality label.

Define the event label set $L_C = \{TS, S, U\}$ to correspond to confidentiality levels "Top Secret", "Secret", and "Unclassified", with $TS \rhd_C S \rhd_C U$. The specification is as follows:

$$G = \langle V_N, V_T, P, S\rangle$$
$$V_N = \{S, T\} \cup \bigcup_{s\in L_C}\{^sY_1, ^sY_2, ^sZ_1\}$$
$$V_T = \bigcup_{s\in L_C}\{^s\alpha_1, ^s\alpha_2, ^s\beta_1\}$$
$$P = \{S \to TS\} \cup \bigcup_{s\in L_C}\left\{\begin{matrix}^sY_1 \to ^s\alpha_1\\ ^sZ_1 \to ^s\beta_1\end{matrix}\right\}$$
$$\cup \bigcup_{s_1,s_2\in L_C}\left\{\begin{matrix}^{s_1}Y_1{}^{s_2}Y_2 \to {}^{s_2}Y_2{}^{s_1}Y_1\\ ^{s_1}Z_1{}^{s_2}Y_1 \to {}^{s_2}Y_1{}^{s_1}Z_1\end{matrix}\right\}$$
$$\cup \bigcup_{\substack{s_1,s_2,s_3\in L_C\\ s_3\rhd_C s_1\\ s_3\rhd_C s_2}}\{T \to {}^{s_1}Y_1{}^{s_2}Y_2{}^{s_3}Z_1\}$$
$$prereq = \bigcup_{\substack{s_1,s_2,s_3\in L_C\\ s_3\rhd_C s_1\\ s_3\rhd_C s_2}}\{\langle {}^{s_1}\alpha_1, {}^{s_3}\beta_1\rangle, \langle {}^{s_2}\alpha_2, {}^{s_3}\beta_1\rangle\}$$

The prerequisite relation is defined so that the labels attached to output events always dominate those of the two input events that were prerequisites. The transaction production $T \to {}^{s_1}Y_1{}^{s_2}Y_2{}^{s_3}Z_1$ is defined in an analogous manner.

Prerequisite confidentiality, as defined in this section, provides a simple, intuitive notion of confidentiality. The definition is grounded in the relationship between particular groups of input and output events. By enforcing the confidentiality constraint at the place where the causal relationships between events is most visible, we greatly simplify the understanding and application of the property. In addition, the grammatical expression of prerequisite confidentiality allows us to easily construct a mechanical recognizer for event sequences that satisfy the property. This ability is not available with other models or confidentiality definitions.

Since the dominates operator used in the definition of prerequisite confidentiality is clearly a transitive relation, prerequisite confidentiality is an example of a component transaction property. Therefore, prerequisite confidentiality is preserved under composition.

6 COMPARISON WITH OTHER DEFINITIONS

In this section we will compare prerequisite confidentiality to several previous definitions of confidentiality.

6.1 SUTHERLAND'S DEDUCIBILITY

In 1986, David Sutherland took a different approach to solving the security modelling problem.[Sut86] He made the claim that his model was "a generalization of the Goguen-Meseguer Model." The basis of the model is a standard state machine representation. Each distinct execution of the system being modelled is an element of the set of *possible worlds*. A piece of information about the system is represented by an *information function* whose domain is the set of possible worlds. For example, one information function might represent all of the information that is available for observation by a particular user.

Sutherland then defines inference in terms of these concepts:

> Given a set of possible worlds Ω and two [information] functions f_1 and f_2 with domain Ω, we say that information flows from f_1 to f_2 if and only if there exists some possible world ω and some element z in the range of f_2 such that z is achieved by f_2 [i.e., $f_2 = z$] in some possible

world, but in every possible world ω' such that $f_1(\omega') = f_1(\omega), f_2(\omega') \neq z$.

In other words, information flows from f_1 to f_2 if knowing the value of f_2 rules out (or eliminates from consideration) even a single possible value of f_2.

Using these definitions, he formalized the notion of security. The binary relation $legal\_to\_get \subseteq I \times I$ (where I is the set of information functions) defines those entities that are entitled to obtain certain pieces of information, as required by the security policy. The security requirement was then stated as follows:

> For all f_1 and f_2 in I, if information flows from f_2 to f_1 then $legal\_to\_get(f_1, f_2)$.

Sutherland then instantiated the model using a state machine framework as its basis. One of the key information functions used is $view(l)$, that, given a possible world, returns the subsequence of the execution sequence consisting of those inputs (to the state machine) whose security levels are $\leq l$. Conversely, the $hidden\_from(l)$ function returns the subsequence of inputs whose levels are not $\leq l$. Security is then specified by making it illegal for information to flow from $hidden\_from(l)$ to $view(l)$ for any l. This essentially requires that every possible observation must be compatible with every conceivable secret. This prevents the observer from deducing the secret based on his or her observations.

Deducibility is often thought to be excessively optimistic. If a high-level input is reproduced exactly as a low-level output, but with a, say, 1 in 10^9 chance that the output is not an accurate copy of the input, the system is deemed to be secure.

Prerequisite confidentiality improves upon that proposed by Sutherland by more realistically describing nondeterministic behaviour. If it is possible for an input event to be a prerequisite for an output event, then prerequisite confidentiality requires that the desired event label relationship hold. Thus, confidentiality violations, no matter how unlikely, are made visible by the model.

6.2 MCCULLOUGH'S HOOK-UP SECURITY AND RESTRICTIVENESS

The first significant work in the area of security composition was done by Daryl McCullough in 1987.[McC87]

McCullough's definition of *hook-up security* is based upon execution traces. Given a trace τ, modify it by adding or deleting some high-level inputs to result in a sequence σ, which is not necessarily a valid trace. It may be possible to construct a valid trace τ'' from σ by adding or deleting high-level outputs after the last input in, or immediately following, the modified input sequence. With this, hook-up security can then be defined as:

> A system is *hook-up secure* if, for every τ and for every σ that can be formed from it, at least one τ'' exists.

Intuitively, this requires that no high-level input can have an effect on low-level events and that the "fixing" of high-level outputs must wait until after any immediately following inputs.

The following theorem is then presented:

> If a system is hook-up secure, then it is deducibility secure, and it has the non-interference property. If systems A and B are both hook-up secure, then any composite system formed by identifying outputs of A with inputs of B with the same security level, and vice-versa, is hook-up secure.

The validity of this result has been questioned in the literature.[LBL+88] In a later paper McCullough redefined his property as *restrictiveness*, and improved its abilities to handle nondeterministic systems.[McC88]

It is not clear how one might describe hook-up security using the grammatical method, since the definition is not based upon a relationship between causally related input and output events, as is the case with prerequisite confidentiality. In fact, it may be impossible to construct an appropriate recognizer that determines whether an event system satisfies the hook-up security property.

6.3 SECURITY AND MACHINE COMPOSITION

In 1988, Johnson and Thayer presented a new security property that is composable, but which is demonstrably weaker (less restrictive) than McCullough's hook-up security.[JT88] Event traces are used as the basis for Johnson and Thayer's model. Informally, a *perturbation* of a trace is a sequence formed by inserting, modifying, or deleting high-level inputs in the trace. A *correction* of a sequence is a trace obtained by inserting, modifying, or deleting high-level non-inputs (i.e., outputs or internal events) in the sequence. A system is *correctable* if every perturbation has a correction that is a valid trace. In general, correctability is not preserved by composition.

They then introduce a restricted form of correctability, denoted *n-forward correctability*, that is a composable property. The additional restrictions are as follows:

- It must be possible to correct every perturbation by modifying the sequence only after the perturbed part (if at all) to form a valid trace.

- If the perturbation happens to be immediately followed by n or fewer low-level inputs, it must be possible to delay any correction until after those inputs (and form a valid trace).

As in the case of hook-up security, it seems unlikely that an appropriate recognizer could be constructed to determine whether an event system satisfies the required *n-forward correctability* property.

6.4 LIN'S BEHAVIORAL SECURITY

In 1991, Ping Lin proposed a new security property, behavioral security, that is a form of noninterference.[Lin91] A system algebra, based upon many-sorted algebras, was introduced to allow a representation of the input/output behaviour of systems and the effects of composition. The security property was then expressed within the system algebra to prevent the disclosure of both high-level inputs and outputs to low-level observers. The low-level behaviour of the system had to remain the same, regardless of whether high-level activity was present or not. It was proven that behavioral security was composable.

Our event system explication of prerequisite confidentiality is analogous to the algebraic definition of behavioral security. Prerequisite confidentiality requires that high-level input events never be prerequisites for low-level output events. Thus, if high-level input activity is removed, the low-level activity remains unchanged.

The connection between prerequisite confidentiality and behavioral security is based upon the fact that both are simply-expressed notions of input/output causality that are preserved under composition. Further work would need to be done to determine whether a mechanical algorithm exists for recognizing components/event sequences that have the behavioral security property.

7 SUMMARY AND CONCLUSIONS

This paper began by introducing the notion of event systems that describe all possible valid event sequences for a component being modelled. It was then proposed that formal grammars be used to specify the structure of the valid event sequences of modelled components. The overall behaviour of the component is represented by a component grammar G and prerequisite relation *prereq*. Valid event sequences for a component correspond to sentences in the language generated from G.

A good representational technique should allow the modelling of a wide range of practical situations. Through the general nature of our technique, it should be clear that we can easily model virtually any component that reacts to input events by generating output events. Most real systems can be represented in this way.

This modelling methodology was then applied to a new definition of confidentiality, called prerequisite confidentiality. By comparison with several existing definitions, it was demonstrated that this definition is superior. It provides a simple, intuitive notion of confidentiality that is based upon causally-related events. Because it is based upon a grammatical specification, it is always possible to construct a recognizer that will identify those event sequences that satisfy the property.

This ability has important practical implications for the model. The construction of an automated specification and verification tool based on the model would require the ability to quickly and easily verify whether an event sequence was valid for a modelled system or if it satisfied prerequisite confidentiality. By using grammars from Turnbull's DRP class, developing an algorithm to do this is straightforward. This is not the case with most other confidentiality definitions.

8 ACKNOWLEDGEMENTS

This work was based upon doctoral research by J.P. Nestor in the Department of Electrical and Computer Engineering at the University of Toronto, under the supervision of Professors E.S. Lee and P.I.P. Boulton. The work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC). The authors would also like to thank the reviewers for their valuable comments and suggestions.

9 REFERENCES

[Cho56] Chomsky, N. Three Models for the Description of Language. *IRE Transactions on Information Theory*, 2(3):113–124.

[Cho59] Chomsky, N. On Certain Formal Properties of Grammars. *Information and Control*, 2(2):137-167.

[GM82] Goguen, Joseph A., and José Meseguer. Security Policies and Security Models. *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, April 1982, pp. 11–20.

[GM84] Goguen, Joseph A., and José Meseguer. Unwinding and Inference Control. *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, April 1984, pp. 75–86.

[Hin93] Hinton, Heather M. An Environment-Based Approach to the Composition of Safe Systems. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, January 1993.

[Hoa78] Hoare, C.A.R. Communicating Sequential Processes. *Communications of the ACM*. 21(8):666–677, August 1978.

[HU69] Hopcroft, John E. and Jeffrey D. Ullman. *Formal Languages and their Relation to Automata*. Series in Computer Science and Information Processing, Addison-Wesley, 1969.

[HU79] Hopcroft, John E. and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science, Addison-Wesley, 1979.

[Knu69] Knuth, D.E. On the Translation of Languages from Left to Right. *Information and Control 8*, 1965, pp. 607–639.

[Lin91] Lin, Ping. The Composability of Behaviorally Secure Systems, Ph.D. Thesis, Department of Electrical Engineering, University of Toronto, 1991.

[LBL+88] Lee, E.S., P.I.P. Boulton, D.M. Lewis, M. Stumm, and B.W. Thomson. A Trusted Network Architecture. Technical Report CSRI-228, Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada, 1988.

[LBTS92] Lee, E.S., P.I.P. Boulton, B.W. Thomson, and R.E. Soper. Composable Trusted Systems. Technical Report CSRI-272, Computer Systems Research Institute, University of Toronto, Toronto, Ontario, May 1992.

[McC87] McCullough, Daryl. Specifications for Multi-Level Security and a Hook-Up Property. *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, 1987, pp. 161–166.

[McC88] McCullough, Daryl. Noninterference and the Composability of Security Properties. *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, 1988, pp. 177–186.

[Nes93] Nestor, John P. The Composition of Property-Preserving Event Systems. Ph.D. Thesis. Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, November, 1993. Also available as Technical Report CSRI-290, Computer Systems Research Institute, University of Toronto, Toronto, Ontario, November, 1993.

[Par91] Parker, Donn B. Restating the Foundation of Information Security. *Proceedings of the Second Annual North American System Security Symposium*, Toronto, Ontario, October 1991.

[Sut86] Sutherland, David. A Model of Information. *Proceedings of the 9th National Computer Security Conference*, 1986, pp. 175–183.

[Tur74] Turnbull, Christopher J.M. Deterministic Left to Right Parsing. Ph.D. Thesis, Department of Electrical Engineering, University of Toronto, 1974.

[TL79] Turnbull, Christopher J.M. and E.S. Lee. Generalized Deterministic Left to Right Parsing. Acta Informatica 12, 1979, pp. 187-207.