Component Specification Using Event Classes

Mark Bickford

Cornell University Computer Science
and ATC-NY
33 Thornwood Dr. Suite 500, Ithaca, NY, USA
markb@cs.cornell.edu

Abstract. Working in a higher-order, abstract *logic of events*, we define *event classes*, a generalization of interfaces, and *propagation rules* that specify information flow between event classes. We propose a general definition of a *component* as a scheme, parameterized by a set of input classes, that defines a set of output classes and propagation rules. The specification of a component is a relation between its input classes and defined output classes that follows from its propagation rules and definitions.

We define a subset of *programmable* event classes that can be compiled and executed and a language, called $E^{\#}$, for specifying components. Components specified in $E^{\#}$ preserve programmability–if the component's input classes are programmable then its output classes and propagation rules are programmable.

Thus a component specified in $E^{\#}$ is a higher-order object: given programs for its input classes, it produces a distributed program for propagating information and programs for its output classes. These programs can be passed as inputs to other components so that components can be composed.

1 Introduction

We may view all computing as information processing: a computation receives information, initially or through an on-going interaction with its environment, and produces some information, on exit or interactively. From this viewpoint, specification of a computational process will naturally focus on *information associated with events*. A specification must define how the process recognizes its input events and reads the associated input information and must also define how the environment will recognize the output events produced by the process and read the information associated with them. In addition, the specification defines the relation between the input and output events and their associated information.

The specification of the input (output) events and associated information is usually called the *interface* specification, while the relation between the interfaces might be called the *functional* specification. The interface specification and functional specification are often written separately but, when this is done, important properties of the process may become difficult or impossible to specify.

For example, a separate functional specification may express the relation between the input information and the output information but be unable to express relations that depend on the temporal ordering of the input and output events. Conversely, an interface specification may be unable to express that the information coded by an input event depends on the state of the process, which, in turn, is a function of the prior history of input and output events.

In this paper we define the concept of an *event class* as a generalization of an interface. An event class is a set of events together with a function that assigns information to each event in the class. The concept of event class combines aspects of both interface and functional specification: determining which events are in the class could be an interface issue, while the function that assigns information to the event can depend on state (a function of the past history of events) in ways that would be part of the functional specification.

Using event classes we factor a specification into components where each component is parameterized by a set of input classes. A component defines auxiliary classes and output classes in terms of the input parameters and also imposes information flow constraints called *propagation rules*. A component is a *program scheme* rather than a program: it describes how to *construct* programs for its output classes and information flow rules from programs for its input classes.

2 Constructive Methods

Our work is in the area of automated program construction referred to as the *proofs-as-programs paradigm*[2,6]. The idea is that certain logical statements can be regarded as specifications and constructive proofs of such statements can be transformed into programs that satisfy the specifications.

For example, a constructive proof of the statement

$$\forall x\colon T.\ \exists y\colon S.\ R(x, y)$$

can be transformed into a functional program F, of type $T \rightarrow S$ such that, $R(x, F(x))$ is true, for any $x \in T$. Constructive theorem proving systems such as NuPrl and Coq *extract* the *constructive content* of proofs and produce such functions automatically.

As another example, for any type Σ of symbols, Σ^* is the same as the type $\mathit{list}(\Sigma)$, and we can also define the type $\mathit{Reg}(\Sigma)$ of regular grammars over Σ and the relation $w \in \mathit{Lang}(G)$ for $G \in \mathit{Reg}(\Sigma)$ and $w \in \Sigma^*$. The constructive content of $w \in \mathit{Lang}(G)$ is a *parse tree* for w as a member $\mathit{Lang}(G)$. The constructive content of the following *decidability theorem*

$$\forall G\colon \mathit{Reg}(\Sigma).\ \forall w\colon \Sigma^*.\ w \in \mathit{Lang}(G)\ \vee\ w \notin \mathit{Lang}(G)$$

is a *parser generator*. Given a grammar G it produces a parser P which, when given word w, produces either a parse tree for w or determines that $w \notin \mathit{Lang}(G)$.

A parser generator is clearly a useful tool, but the efficiency of the parsers generated by this method depends on the proof of the decidability theorem. If we merely prove that it is possible to generate all possible parse trees of w and check each one for conformance with the grammar G, then the parsers extracted from this proof will use this infeasible algorithm. If, on the other hand, we develop the theory of finite state automata and prove the decidability theorem by showing that every regular grammar is recognized by an automaton, then the extracted parsers will be efficient.

Over the last several years, we have developed constructive methods that extend the proofs-as-programs paradigm to the generation of distributed systems. The programs we generate are not purely functional, they have state, they have multiple agents or threads, and agents send messages to one another. To extract such programs from proofs of specifications, the specifications must be logical properties of inherently distributed structures, the *event structures* discussed in section 3.

The concept of *event classes* has arisen, in the context of this work, as a useful abstraction with which to write specifications. Event classes combine some of the features of the examples just mentioned. Recognizing when an event is in an event class is analogous to deciding whether $w \in Lang(G)$–with primitive *kinds* of events (e.g. receipt of a message with a particular header) corresponding to the basic symbols Σ, and the history of prior events corresponding to the word w. Computing the value of an event in the class is like computing a functional program. These two aspects are woven together. The events are recognized not merely on the basis of the primitive kinds of prior events, but on the values computed for them. The value computed for an event can be a function of some state variables, where state variables are expressed abstractly as the most recent value of some event class.

Event classes are useful because we can combine them using *combinators* that encode programming patterns. We prove general abstract properties of these combinators, and use them to derive further properties of specifications that use the combinators.

Equally important is the fact that we can define a subset of event classes that we call *programmable*. A programmable class can be implemented by an efficient program, one that uses a finite number of threads, has a finite set of state variables, and handles a finite set of primitive kinds of events. Such a program is analogous to the finite state automata that recognize regular languages.

We will describe a language, $E^{\#}$, for defining components in which all classes are programmable. $E^{\#}$ achieves this property by limiting the definition of event classes to a certain general form that preserves programmability. Because of this, correct-by-construction programs can be automatically generated from specifications written in $E^{\#}$.

It is probably obvious, from this, that $E^{\#}$ can specify only a limited range of distributed systems. One major limitation is that $E^{\#}$ can express only the temporal ordering between events, not explicit time bounds between events. The reason is that generating correct-by-construction programs for specifications with explicit time bounds would require an algorithm to decide whether the time bounds are feasible and produce a schedule when they are. It is a goal for future work to find a restricted class of programs and specifications for which the scheduling problem is decidable, so that the constructive approach can be applied.

Another current limitation of $E^{\#}$ is that messages can only be sent on links with known endpoints. More flexible message passing methods can be added to a future version of $E^{\#}$ provided that we can give precise logical specifications for them and correctly implement the specifications with efficient programs.

Even with these limitations, $E^{\#}$ is expressive enough to specify a variety of distributed algorithms, such as leader election and consensus protocols, or authentication protocols. The concepts, event class and programmable event class, may also be useful for other component specification models, even if the goal is not program synthesis.

All of our concepts and definitions are expressed in a logical language that we call the *logic of events*. We begin with a brief overview of this language.

3 Logic of Events

An *event language* $\mathcal{L}$ is any language extending $\langle E, loc, <\rangle$, where loc is a function on E, and $<$ is a relation on E. For an event $e \in E$, $loc(e)$ is the *location* of the event and $e_1 < e_2$ means event e_1 causally precedes event e_2. Locations represent agents, processes, or threads. An *event structure*[1] is a model for $\mathcal{L}$ that satisfies the axioms

- $\leq$ is a locally-finite partial order (every e has finitely many predecessors)
- *Local order*, defined by $e_1 <_{loc} e_2 \equiv e_1 < e_2 \wedge loc(e_1) = loc(e_2)$, is a total ordering on any set of events whose members all have the same location.

There is an analogy between event structures and space-time: events correspond to points, locations to spatial coordinates, and $<$ and $<_{loc}$ to temporal ordering. We have also added an operator, *time*(e), to the event language that provides a temporal coordinate, but, since the methods described here construct programs only for asynchronous systems, our specifications will use only the temporal ordering relations.

To make a connection between abstract events and programs we must make clear what attributes of an event a program can recognize. Some events are receipts of messages while others represent internal events such as a timer expiring or a random number being generated. We therefore add two operators, *kind*(e), and *val*(e) to the event language. When an event e occurs, a program that has access to *kind*(e), and *val*(e) can pass *val*(e) to an appropriate handler, based on *kind*(e).

We use a model of message passing in which messages are sent reliably on named, fifo, *links* from one location, the *source* of the link, to another location, the *destination* of the link. Each message has an associated *tag*, which can be thought of as a header, and is included in the kind. Thus, one kind of event is **rcv**(l, tg). An event e of this kind is the receipt of a message with tag tg on link l. In this case, *val*(e) is the remaining data in the received message.

Any other event e is called an internal event and has a kind of the form **internal**(nm), where nm is a token. In this case, *val*(e) may have whatever meaning is appropriate for the internal action. In particular, it provides a convenient way to represent nondeterministic choice: the value of an internal event can be chosen nondeterministically.

4 Using the Logic of Events

We use the logic of events within a higher-order logic so that we can define and quantify over functions from events to events, and functions from events to other types. In particular, we use the constructive type theory of NuPrl, but our methods are compatible with any higher-order logic, such as the logics (classical or constructive) used formal systems like Isabelle/HOL, PVS, and Coq.

[1] The event structures defined here differ from those defined by Winskel [7]. See section 12 for a comparison.

For example, we will need the concept of an *event proposition*, a proposition about events. In a classical higher-order logic, event propositions are just sets of events, so they have type[2] $E \to \mathbb{B}$. In constructive type theory, a proposition about events is a member of the type[3] $E \to \mathbb{P}$. In fact, since the type, E, of events, depends on the event structure we are talking about, this definition, and all the definitions mentioned in this paper, include another parameter, es, of type *ES*, the type of event structures. Our notation consistently hides the parameter es.

As an example of the expressiveness of event logic we define the operators of linear temporal logic. If P and Q are event propositions, then the following are also event propositions:

$$\begin{aligned}
P \wedge Q &=_{def} \lambda e.\, P(e) \wedge Q(e)\\
P \Rightarrow Q &=_{def} \lambda e.\, P(e) \Rightarrow Q(e)\\
\neg P &=_{def} \lambda e.\, \neg P(e)\\
\Box P &=_{def} \lambda e.\, \forall e'\colon E.\, e \le_{loc} e' \Rightarrow P(e')\\
\Diamond P &=_{def} \lambda e.\, \exists e'\colon E.\, e \le_{loc} e' \wedge P(e')\\
P\,\mathbf{U}\,Q &=_{def} \lambda e.\, \exists e'\colon E.\, e \le_{loc} e' \wedge Q(e') \wedge\\
&\qquad\qquad \forall e''\colon E.\, (e \le_{loc} e'' <_{loc} e') \Rightarrow P(e'')
\end{aligned}$$

We easily prove identities such as:

$$\begin{aligned}
\Box\Box P &\equiv \Box P\\
\Diamond\Diamond P &\equiv \Diamond P\\
\Diamond P &\equiv \mathrm{TR}\,\mathbf{U}\,P
\end{aligned}$$

where

$$\begin{aligned}
P \equiv Q &=_{def} \forall e\colon E.\, P(e) \Leftrightarrow Q(e)\\
\mathrm{TR} &=_{def} \lambda e.\, \mathrm{True}
\end{aligned}$$

The logic of events is more general than linear temporal logic for several reasons. The set of locations can be arbitrary, so there are many disjoint sets of linearly ordered events, but these events may be related by the causal order. Higher-order logic over a model that includes events allows us to define functions from events to events, and express useful properties of such functions.

For example, suppose that $(*) \quad \forall e\colon E.\, P(e) \Rightarrow \exists e'\colon E.\, Q(e') \wedge R(e,e')$ This means that there is a function $f\colon \{e\colon E \mid P(e)\} \to \{e'\colon E \mid Q(e')\}$ that satisfies $\forall e\colon E.\, P(e) \Rightarrow R(e, f(e))$. This property merely states that f is a Skolem function for $(*)$, but we can express additional properties of a function like f that do not follow from f being a Skolem function. For example, we can state that f is *local order preserving*:

$$\forall e_1, e_2\colon E.\, e_1 <_{loc} e_2 \Rightarrow f(e_1) <_{loc} f(e_2)$$

[2] $\mathbb{B}$ is the type of Boolean values $\{T, F\}$.

[3] In constructive logic, the propositions $\mathbb{P}$ are identified with the universe of types. A nonempty type is "true" and members of the type are witnesses to its truth.

5 Event Classes

An *event class* (of type T) is simply a partial function from events to values of type T. If X is an event class[4], then we write $E(X)$ for the set of events in the domain of X and say that events in $E(X)$ are *in* the class X. For an event $e \in E(X)$, $X(e)$ is the value (of type T) assigned to e by class X.

Thus, class X partitions the events and for those events e that are in $E(X)$ assigns a value of type T.

Notation: We write[5] $X(e) = \omega$ to mean $e \notin E(X)$.

Notation: If $t(X, \ldots, Y)$ is an expression mentioning event classes $X, \ldots, Y$ define $[t(X, \ldots, Y)]$ to be

$$\lambda e.\ e \in E(X) \wedge\ \ldots\ \wedge e \in E(Y)\ \wedge\ t(X(e), \ldots, Y(e))$$

So, for example, $[X = 5](e)$ means $e \in E(X)\ \wedge\ X(e) = 5$. Note that an event e may be in any number of different event classes.

The class $f(X)$: If f is a function from $T \rightarrow S$, we may define a class $Y = f(X)$ by stipulating that

$$E(Y) \subseteq E(X)\ \wedge\ \forall e\colon E.\ \forall v.\ [X = v](e)\ \Rightarrow\ [Y = f(v)](e)$$

This says that an event is in class Y if and only if it is in class X, and its Y-value is computed from its X-value by applying function f.

The class $(X|p)$: Similarly, if p is a predicate $T \rightarrow \mathbb{B}$, we may define a class $Y = (X|p)$ by stipulating that

$$\forall e\colon E.\ \forall v.\ [X = v\ \wedge\ p(v)](e)\ \Leftrightarrow\ [Y = v](e)$$

This says that an event is in class Y if and only if it is in class X and its X-value passes the test p.

The class $(X)'$: Class $Y = (X)'$ (of great importance to us) is defined by the property that, for any event e, the value $(X)'(e)$ is the X-value of the most recent X-event occurring prior to e, if there is one, and otherwise, e is not in class $(X)'$. Formally, for all events e, and values v:

$$\begin{aligned}[Y = v](e) \Leftrightarrow \exists e'\colon E.\ e' <_{loc} e\ \wedge\ [X = v](e') \wedge{} \\ \forall e''\colon E.\ e' <_{loc} e'' <_{loc} e \\ \Rightarrow e'' \notin E(X)\end{aligned}$$

[4] In classical logic, the domain $E(X)$ and the values $X(e), e \in E(X)$ completely characterize the event class X. In constructive logic, we will need an additional property: the domain $E(X)$ is *decidable*–there is an effective procedure to decide whether or not an event e is in $E(X)$. Thus, in constructive type theory an event class is represented by a function of type $E \rightarrow (T + \mathit{Unit})$.

[5] We use ω rather than $\bot$ because $X(e) = \bot$ would indicate a computation compatible with all possible results, rather than a computation that has *decided* that e is not in the domain of X.

6 Programmable Event Classes

An event class X is *programmable* if there is a program, possibly a distributed program, that recognizes the events in X and computes their values. Since we assume that programs can directly access only the kind and val of events, a program for class X must decide whether event e is in class X, and, if so, compute $X(e)$ as a function of the history

$$\{\langle kind(e'), val(e')\rangle | e' \ \leq_{loc} \ e\}$$

An efficient program will not store the entire history, it will instead accumulate a function of the history. Also, the program for a class should depend on the history of only a fixed, finite set of kinds of events. Without this restriction, a program's behavior could change when composed with another program that introduced other kinds of events. Finally, the events in a programmable class should occur at only finitely many locations, so that a distributed program for the class will require only a finite set of agents.

We therefore define a *class program* (of type T) to be

1. a finite list of locations, $locs$, and for each $i \in locs$:
2. an initial state, $s_0(i)$
3. a finite list of event kinds, $ks(i)$
4. a function $update_i(s, k, v)$ for updating the state s when an event with kind $k \in ks(i)$ and val v occurs at location i
5. a function $test_i(s, k, v)$ that returns a member of the disjoint union $(T + \text{Unit})$

Type $T + S$ is the disjoint union of types T and S. Its members are values of the form $inl(t)$, where $t \in T$, or the form $inr(s)$, where $s \in S$. Thus the type $(T + \text{Unit})$ represents values from T (as $inl(t)$) and ω (as $inr()$).

The run of a class program initializes, at each of its locations, i, a state variable s_i to $s_0(i)$, and sets the state variable $s_i := update_i(s_i, k, v)$ whenever an event of kind $k \in ks(i)$ and val v occurs at location i.

The value returned by test function, $test_i$, is a member of a disjoint union– either $inl(t)$, for some $t \in T$, or else $inr()$. Event e with $loc(e) = i$ is in the class computed by the program if and only if $i \in locs$, and $kind(e) \in ks(i)$, and $test_i(s_i, kind(e), val(e))$ returns $inl(t)$. In this case, t is the value assigned to e by the class program.

Definition. An event class is *programmable* if it is the class computed by a class program.

Examples

- The class VAL(k) where $[\text{VAL}(k) = v](e) \ \Leftrightarrow \ (kind(e) = k \ \wedge \ val(e) = v)$, is programmable.
- If X is programmable then the classes $f(X)$ and $X|p$ are programmable.
- The class $(X)'$ is not programmable since events of any kind may occur after an X-event, so there is no finite list of kinds for all $(X)'$-events.

7 Closure Properties of Event Classes

The event classes $f(X)$ and $X|p$ are simple examples of a class defined as a function of other classes. In fact, event classes are closed under a very general definition scheme. We illustrate this with a few more examples, before discussing the general case.

The class $(X+Y)$*:* Suppose that X is a class of type T, and Y is a class of type S. We can define a class $X+Y$, of type $T+S$, so that

$$\begin{aligned} [X=t](e) &\Rightarrow [(X+Y)=inl(t)](e) \\ e \notin E(X) \ \wedge \ [Y=s](e) &\Rightarrow [(X+Y)=inr(s)](e) \\ E(X+Y) &\subseteq E(X) \cup E(Y) \end{aligned}$$

For this, it suffices to define[6]

$$\begin{aligned} X+Y &= \lambda e.\ F(X(e),Y(e)) \\ \text{where: } F(\omega,\omega) &= \omega \\ F(t,-) &= inl(t) \\ F(\omega,s) &= inr(s) \end{aligned}$$

The clause, $F(\omega,\omega)=\omega$ implies that an event in $X+Y$ must be in at least one of the classes X or Y.

The class $(X;Y)$*:* Suppose that X is a class of type T, and Y is a class of type S. The class $X;Y$, of type $T \times S$, includes only Y-events, e, that have been preceded by an X-event. The value, $(X;Y)(e)$, of such an event is the pair $\langle t,s\rangle$ where $Y(e)=s$ and t is the value of the most recent X-event before e.

Recalling our definition of the class $(X)'$, we see that

$$\begin{aligned} X;Y &= \lambda e.\ G(Y(e),(X)'(e)) \\ \text{where: } G(\omega,-) &= \omega \\ G(s,\omega) &= \omega \\ G(s,t) &= \langle t,s\rangle \end{aligned}$$

The clause, $G(\omega,-)=\omega$ implies that an event in $X;Y$ must be in class Y. The clause, $G(s,\omega)=\omega$ implies that an event in $X;Y$ must also be in class $(X)'$.

The class **accum**(h,b,X)*:* When an event e in class X occurs, we can assign it not only its X-value, v, but a "running total" of v and the values of all prior X-events. The running total is defined by the parameters h and b: b is the total when no X-events have occurred, and $h(t,v)$ is the new total when an X-event of value v occurs when the previous total is t. The class that contains the same events as class X but assigns them this running total, we name **accum**(h,b,X). It can be defined by:

$$\begin{aligned} \mathbf{accum}(h,b,X) &= \lambda e.\ H(X(e),(\mathbf{accum}(h,b,X))'(e)) \\ \text{where: } H(\omega,-) &= \omega \\ H(v,\omega) &= h(b,v) \\ H(v,t) &= h(t,v) \end{aligned}$$

[6] If classes X and Y have an event e in common, then the definition of $(X+Y)(e)$ is asymmetric–it treats e only as an X-event. A slightly more complicated definition of the class would have type $T+S+(T\times S)$ and handle this case symmetrically.

This definition is recursive since it defines **accum**(h, b, X) as a function of itself. The recursion is well-founded because the occurrence of **accum**(h, b, X) on the righthand side of the definition is inside of a "prime". The value of **accum**(h, b, X) at event e is defined in terms of the prior values of **accum**(h, b, X). The ordering $<_{loc}$ is well-founded, so the recursion is well-founded and defines a unique event class.

Example: Request/Ack Suppose we want to design a component that enforces the requirement

> After a request is made, no request is made until after an acknowledgment is received.

One design that implements this requirement is to remember the last relevant event and when "a request is requested" make the request if the last event was an ack and ignore the request otherwise.

We can parameterize our component design with two input classes W (for *wants* to request) and A (for *acknowledge*). The relevant events are in class $W + A$, and remembering the last relevant event corresponds to the class $(W + A)'$. If we define

$$\begin{aligned} \mathit{Req} &= \lambda e.\ H(W(e), (W + A)'(e)) \\ \text{where: } H(\omega, -) &= \omega \\ H(x, \mathit{inl}(x')) &= \omega \\ H(x, \text{otherwise}) &= x \end{aligned}$$

Then the clause $H(\omega, -) = \omega$ implies that a *Req*-event must be a W-event, and the second clause, $H(x, \mathit{inl}(x')) = \omega$ implies that a W-event is not a *Req*-event when the last relevant event was a W-event. Otherwise, either there was no prior relevant event or the prior event was an A-event, so the W-event is also a *Req*-event (with the same value).

It is now easy to show that, *no matter what the classes W and A are*, the class *Req* satisfies

$$\begin{aligned} &\forall e\colon E(\mathit{Req}).\ \forall e'\colon E(\mathit{Req}).\ (e \ <_{loc}\ e') \Rightarrow \\ &\qquad \exists e''\colon E(A).\ (e \ <_{loc}\ e'' \ <_{loc}\ e') \end{aligned}$$

This is a formal statement of the original requirement.

This example illustrates (part of) our general concept of a component: in terms of a set of input classes, it defines a set of output classes and a guaranteed relation between the input classes and defined output classes. The need for something more can be seen from the fact that merely defining the class of events that are "allowed to make a request" does not force any requests to actually be made (and our specification was trivially satisfied when there are no requests). To state that events in a class cause some action to be taken, we will use *propagation rules* (see section 9).

General class combinator: The classes $X + Y$, $X; Y$, **accum**(h, b, X), and *Req* share a common definition form under which the set of event classes is closed. The general form, which we call a *class combinator*, is a function

$$H(U_1, \ldots, U_m; P_1, \ldots, P_n; S)$$

of $m > 0$ *unprimed* class arguments, $n \geq 0$ *primed* class arguments, and one, optional, *self* argument, that is *strict in its unprimed arguments*:

$$H(\omega, \ldots, \omega; -, \ldots, -, -) = \omega$$

Theorem 1. *If H is a class combinator, then for any event classes*

$$U_1, \ldots, U_m; P_1, \ldots, P_n$$

there is a unique event class C such that for every event e,

$$C(e) = H(U_1(e), \ldots, U_m(e); (P_1)'(e), \ldots, (P_n)'(e); (C)'(e))$$

Proof. Let[7] $C =_{rec} \lambda e.\ H(U_1(e), \ldots, U_m(e), (P_1)'(e), \ldots, (P_n)'(e), (C)'(e))$
To show that C is an event class, we must prove that, for every event e, $C(e)$ is defined– either ω or some value x. We prove this by induction on $<_{loc}$. Assuming that $C(e')$ is defined for all $e' \ <_{loc}\ e$ we see that $(C)'(e)$ is defined and hence $C(e)$ is defined. The uniqueness of C is easy to prove. □

Notation: Let $C_H(U_1, \ldots, U_m, P_1, \ldots, P_n)$ be the class defined by combinator H and classes $U_1, \ldots, U_m; P_1, \ldots, P_n$.

In fact, Theorem 1, is an instance of a more general theorem that allows class definitions with mutual recursion.

8 Closure Properties of Programmable Event Classes

The programmable event classes are also closed under the class combinators.

Theorem 2. *If H is a class combinator, and event classes $U_1, \ldots, U_m; P_1, \ldots, P_n$ are programmable, then the event class $C_H(U_1, \ldots, U_m, P_1, \ldots, P_n)$ is programmable.*

Proof. We give a sketch of the proof. We proved the full version of the theorem, constructively, in NuPrl.

Each of the classes U_i and P_j has a program; let these be

$$Pgs = \ldots, Pgm(U_i), \ldots, Pgm(P_j), \ldots$$

We describe the program for $C_H(U_1, \ldots, U_m, P_1, \ldots, P_n)$, first for the case where the combinator H does not mention the optional "self" argument. The program has:

1. the list of locations, $locs$ formed from the union of the locations in Pgs. And has, for each $i \in locs$:

[7] In NuPrl, recursive definitions of the form $f \ =_{rec}\ \lambda x.\ F(f, x)$ are made using the *Y-combinator*, for which $Y(F) = F(Y(F))$. We prove that functions defined by recursion are total by an induction on some well-founded ordering (in this case, $<_{loc}$). Other logical systems, such as Coq, and PVS do not include the Y-combinator or general recursion, but do have strong enough induction/definition principle to carry out our argument, or will support a simple fixed point argument in domain theory.

2. the initial state, $\langle s^1_0(i), s^2_0(i), \ldots, \omega, \ldots, \omega\rangle$ consisting of the tuple of initial states of all programs in Pgs that include location i and extra state components, pr_j, with initial state ω, for each of the primed arguments P_j that contain location i. The extra state pr_j will remember the most recent value of class P_j.
3. the list of kinds, $ks(i)$ is the union of the kinds from each program in Pgs that includes location i.
4. the update function $update_i(s, k, v)$ updates each component of the state just as the corresponding program in Pgs would (if k is not one of the kinds for that program, the corresponding component is unchanged). The extra state component pr_j corresponding to argument P_j is updated as follows: If k is one of the kinds for program $Pgm(P_j)$ and s_j is the component of the state for P_j, and t_{ij} is the test function from program $Pgm(P_j)$ for location i, then if $t_{ij}(s_j, k, v) = inl(x)$ for some x, pr_j is set to x. Otherwise p_j is unchanged.
5. the test function $test_i(s, k, v)$ is

$$H(t_{i1}(s_1, k, v), \ldots, t_{im}(s_m, k, v), pr_1, \ldots, pr_n)$$

If the combinator H does mention the optional "self" argument, then each location will include one more state component, *self*, initially ω. This component is updated to x when the test function of the program returns $inl(x)$, and the test function itself is

$$H(t_{i1}(s_1, k, v), \ldots, t_{im}(s_m, k, v), pr_1, \ldots, pr_n, \mathit{self}) \qquad \square$$

Our constructive proof of theorem 2 provides the basis for a verified compiler for the language $E^{\#}$ described below.

9 Propagation Rules

Starting with basic programmable classes like VAL(k) and applying $(X + Y)$, $(X; Y)$, **accum**(h, b, X), and other definable class combinators, we can define complex programmable event classes. The program that implements such a class accumulates state information needed for the test function to decide whether an event is in the class and, if so, compute its value. But how is this information to be used by components of a distributed system?

Distributed systems must propagate information from one location to another, and this is not done by the programs for event classes. So, we need one more ingredient to complete our language for specifying components of distributed systems: *propagation rules*.

The *forward propagation rule* $X \xrightarrow{R} Y$ states that the information associated with an event in class X should propagate to related information associated with an event in class Y. Expressed in the logic of events, part of the formal meaning of this rule is:

$$\begin{aligned} &\forall e\colon E.\, \forall v.\, [X = v](e) \;\Rightarrow \\ &\qquad\qquad \exists e'\colon E.\, e \le e' \;\wedge\; [R(Y, v)](e') \end{aligned}$$

This says that for every event e in class X, with X-value v, there will be a Y-event e', *causally after or equal to* e, with a value related to v by relation R.

Recall that such a $(\forall e.\exists e')$-property can also be expressed using a Skolem function, f, and that we can also assert that f is *local order preserving*. We want the meaning of $X \xrightarrow{R} Y$ to include the order-preserving property, so, accordingly, our full definition of the propagation rule is:

$$\begin{aligned}&\exists f\colon E \to E.\, (\forall e\colon E.\, e \le f(e)) \;\wedge\\ &(\forall e_1, e_2\colon E.\, e_1 \;<_{loc}\; e_2 \;\Rightarrow\; f(e_1) \;<_{loc}\; f(e_2)) \;\wedge\\ &(\forall e\colon E.\, \forall v.\, [X = v](e) \;\Rightarrow\; [R(Y, v)](f(e)))\end{aligned}$$

When the relation $R(v, v')$ is $v = v'$, then we omit R and write simply $X \to Y$; in this case the last clause of the definition is

$$\forall e\colon E.\, \forall v.\, [X = v](e) \;\Rightarrow\; [Y = v](f(e))$$

This says that the value of every X-event will occur later as the value of a Y-event.

Propagation rule for simple send: When $Y = \text{VAL}(\mathbf{rcv}(l, tg))$ then $X \to Y$ means that the X-value of every event in X must be sent (in order) on link l with tag tg. If all of the events in class X occur at the source[8] of l, then this is easily accomplished by adding "send statements" to the program for class X, so that, when the test function returns $inl(x)$, the program sends message $\langle tg, x\rangle$ on link l.

10 The $E^{\#}$ Language

We have built a prototype executable specification language, called $E^{\#}$, based on the concepts of event classes and propagation rules. It provides an ML-like syntax for defining basic items: locations, links, and kinds, and for declaring the types of basic kinds of events. For example, this $E^{\#}$ fragment:

```
let l1 = link(a,b,1);; let l2 = link(b,a,1);;
let r = rcv(l1,req);;  let a = rcv(l2,ack);;
r,a: int
```

defines locations, `a` and `b`, links `l1` and `l2`, and kinds `r` and `a`. Link `l1` goes from `a` to `b` (and it is link number 1 of possibly more links from `a` to `b`). Kind `r` is the kind of a receipt on link `l1` of a message with tag `req`. Because of the declaration `r: int`, the message received will be an integer (the message types are closed under union, product, and list, and include base types of strings, booleans, and integers).

$E^{\#}$ has a syntax for defining event classes and propagation rules. For example:

```
let +(X,Y) = f(X,Y) where f(x,none) = inl(x)
                          f(x,y) = inl(x)
                          f(none,y) = inr(y) end;;
```

[8] If events in class X can occur at locations other than the source of link l, then to implement $X \to \text{VAL}(\mathbf{rcv}(l, tg))$ requires a distributed program that sends the values of X-events to a "proxy" at the source of link l that forwards the values it receives on link l with tag tg.

```
let accum(h,b,X) = g(X, self')
                   where g(a,none) = h(b,a)
                         g(a,t) = h(t,a) end;;
let ;(X,Y) = f(Y, X') where f(b, none) = none
                            f(b,a) = pair(a,b) end ;;
```

defines the combinators $(X + Y)$, **accum**(h, b, X), and $(X; Y)$ defined in section 7. Note that the ascii syntax for ω is `none`, note also that the "strictness clause" like `f(none,none) = none` is supplied automatically and does not need to be written.

A "module" `combinators.esh`, containing general-purpose combinators can be created and then imported into other modules. Our Request/Ackknowledge example could be written:

```
import combinators.esh
let Req(W,A) = h(W,(W+A)')
    where h(x,d) = if isl(d) then none else x  end;;
```

Here the input class parameters `W` and `A` have been made into arguments of a combinator. We plan to extend $E^{\#}$ to declare type and class parameters so that we could write:

```
import combinators.esh
T : type
class W, A : T
let Req = h(W,(W+A)')
      where h(x,d) = if isl(d) then none else x end;;
```

If this were a module `req_comp.esh` then we would use it as follows:

```
class W : int
let l1 = link(a,b,1);; let l2 = link(b,a,1);;
let r = rcv(l1,req);;  let a = rcv(l2,ack);;
r,a: int
let A = val(a);;
let R = val(r);;
import req_comp.esh with (int for T, A for A, W for W)
Req => R
```

This $E^{\#}$ program includes definitions of `A` and `R` as basic classes VAL(**rcv**$(\texttt{l2}, \texttt{ack})$) VAL(**rcv**$(\texttt{l1}, \texttt{req})$). It also includes a propagation rule `Req => R`. The current prototype supports only the "simple send" propagation rules, so the class on the righthand side must be a "basic receive class".

11 Compilation of $E^{\#}$

The $E^{\#}$ compiler parses the input module and then generates two things, a logical specification and a program. The logical specification is a proposition in the logic of events. It states a property that is true in any event structure that is consistent with (i.e. could result from a run of) the program. Our prototype places the generated logical specification into the NuPrl library so that we can use it to derive other properties of the program.

The generated program is a simple *distributed state machine* (DSM). This is similar to the class programs described earlier except that they also contain "send statements". More precisely, a DSM is:

1. a finite list of locations, $locs$, and for each $i \in locs$:
2. a list of state variables with types and initial values.
3. a finite list of event kinds, $ks(i)$
4. handlers for each of these kinds, k: the handler for k is a list of expressions, in the state variables and the variable val, that define the next states for all the state variables when evaluated with val set to the value of the event of kind k. In addition, the handler includes a list of pairs of a link and an expression in the state variables and val. The expressions evaluate to a list of messages (a header tag and a value) that should be sent on the given link.

Verified compiler. The core of our prototype $E^{\#}$ compiler is correct-by-construction because we use the constructive proof of Theorem 2 to assemble the programs for all event classes in the $E^{\#}$ program being compiled. The NuPrl system can *extract* an executable witness term from a proof. The extract of Theorem 2 is a function that takes programs for input classes and produces a program for the class defined by the combinator. We have programs for the basic classes of the form VAL(k) (these come from the extract of a very simple theorem). Thus the $E^{\#}$ compiler simply applies these extracts repeatedly to compute the program for each class.

Compilation of the propagation rules is currently implemented with "hand written" ML code. For the "simple send" propagation rules this is fairly straightforward, but we plan to prove a theorem about the "programmability" of propagation rules and implement this part of the compiler with a "correct-by-construction" witness term, as well.

12 Related Work

We developed the logic of events and the theory of event classes out of our needs for abstraction in the formal verification and/or synthesis of distributed systems. Because of our somewhat confined world view, it is likely that there are many connections between our ideas and existing software engineering methods that we are unaware of. Here are some connections that we are aware of:

de Alfaro and Henzinger [3] use *interface automata* to specify assumptions about the temporal ordering of input events and guarantees on the temporal ordering of output events. They use a game theoretic algorithm to compute a new set of assumptions on the environment when components are composed. The interface assumptions expressed by interface automata can clearly be expressed in the logic of events, and moreover by programmable event classes. We expect that the interface automata composition rule can be made into a combinator on event classes, and this is a topic for further research.

The AsmL specification language [1] developed at Microsoft Research is based on Gurevitch's Abstract State Machine model. The distributed version [5] of this model, Distributed ASM's has a semantics similar to our event structures. In particular, actions are partially ordered with the actions of a single agent linearly ordered. AsmL specifications focus on evolving states while our methods focus on events and associated information and we make state a derived quantity.

Our use of the name *event structure* is misleading to readers familiar with Winskel's event structures[7]. Our event structures are formal model of what might be called *extended message sequence charts* because the events and their locations and ordering form a message sequence chart, while the kinds and values are an extension. Thus our event structures simply model the observed behavior (or a *run*) of a distributed system. The causal ordering between events is merely the ordering in which the events in the run occurred. In contrast, Winskel's event structures are a generalization of Petri-nets, an abstract model of the processes themselves. The ordering relation is an *enabling relation* and the additional *conflict relation* specifies events that can not occur in the same run. For example, a Winskel event structure might have events that have time, location, and value, and the conflict relation contains pairs of events with the same time and location but with different values. This would say that events with the same time and location but different values cannot occur in the same run. We would prove the same result by showing that, in any event structure (in our model, this is a single run) events with the same location and time must be equal, in which case they must have the same value.

The conflict relation in Winskel event structures can be used to model the choice of *fresh* values, such as *nonces* used in authentication protocols. Two events that choose the same nonce can be in conflict, and this models the assumption that all nonces are fresh. In our model, we state such assumptions as logical constraints on the event structures.

13 Ongoing Work

We are currently building "back end" translators from the DSM intermediate language, produced by the $E^{\#}$ compiler, into target languages such as Java, C#, and F#. The Java version should be ready by June, 2009.

We are using the $E^{\#}$ tools (collectively called "Elan"–the Event Logic Assistant) to specify and generate consensus algorithms. We have a complete proof of a simple consensus algorithm[9] that tolerates t crash failures and uses $3t + 1$ voters, and we have specified the algorithm in $E^{\#}$ and generated the intermediate DSM for it. Within another year or two we plan to have correct-by-construction versions of real-world consensus algorithms such as Paxos (Lamport), and the randomized consensus algorithm of Ben-Or.

Another focus of our work is to have the ability to generate *diverse* code from $E^{\#}$ specifications. Having back ends for several languages is one factor for diversification, but we can also use different representations for the state in our generated programs (e.g. arrays, linked lists, etc. for abstract lists, hash tables, balanced trees, etc for sets).

[9] The FLP[4] theorem states that no safe consensus algorithm is guaranteed to terminate. We prove the safety of the consensus algorithm and a weaker liveness property, *non-blocking*, "from any reachable state it is *possible* to reach termination".

Acknowledgments

We would like to thank: David Guaspari for editing and reorganizing several drafts of this report; Bob Constable for enthusiastic technical and financial support for this work and collaboration.

References

1. Barnett, M., Schulte, W.: The abcs of specification: Asml, behavior, and components. Informatica 25, 517–526 (2001)
2. Bates, J., Constable, R.: Proofs as programs. ACM Transactions on Programming Languages and Systems 7, 113–136 (1985)
3. de Alfaro, L., Henzinger, T.: Interface automata. In: Proceedings of the, ACM Press, New York (2001)
4. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. In: PODS 1983: Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems, pp. 1–7. ACM, New York (1983)
5. Gurevitch, Y., Rosenzweig, D.: Partially Ordered Runs: A Case Study. Microsoft Research MSR-TR-99-88 (1999)
6. Martin-Lof, P.: Constructive mathematics and computer programming. In: Mathematical Logic and Programming Languages, pp. 167–184. Prentice-Hall, London (1985)
7. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)