Composing Specifications of Event Based Applications*

Pascal Fenkam, Harald Gall, and Mehdi Jazayeri

Technical University of Vienna, Distributed Systems Group
A-1040 Vienna, Argentinierstrasse 8/184-1
{p.fenkam,h.gall,m.jazayeri}@infosys.tuwien.ac.at

Abstract. The event based architectural style has been recognized as fostering the development of large-scale and complex systems by loosely coupling their components. It is therefore increasingly deployed in various environments such as middleware for mobile computing, message oriented middleware, integration frameworks, communication standards, and commercial toolkits. The development of applications based on this paradigm is, however, performed in such an ad-hoc manner that it is often difficult to reason about their correctness. This is partly due to the lack of suitable specification and verification techniques. In this paper, we review the existing theory of specifying and verifying such applications, argue that it cannot be applied for the development of large-scale and complex systems, and finally propose a novel approach (*LECAP*) for the construction of correct event based applications. Our approach is superior to the existing approaches in many respects: 1) we assume a while-parallel language with a synchronization construct, 2) neither a pending event infrastructure nor a consume statement are required, 3) a dynamic (instead of static) binding is assumed, 4) no restriction is made on the number of simultaneous executions of the same program 5) our approach is oriented towards top-down development of systems. The paper also presents two examples for illustrating the approach.

1 Introduction

The increasing complexity of (distributed) software systems has led to the investigation of new methods that can ease their development. Such methods include formal methods, object orientation, extreme programming, component based software engineering. These approaches are supported by emerging paradigms such as objects, encapsulation, polymorphism, concurrency, communication, shared variables, or mobile agents. Each of these paradigms comes with its set of features that challenge existing specification and verification techniques.

* This work was supported by the European Commission in the Framework of the IST Program, Key Action II on New Methods of Work and eCommerce. Project number: IST-1999-11400 MOTION (MObile Teamwork Infrastructure for Organizations Networking).

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 67–86, 2003.

The event based architectural style is one such paradigm. Essentially, an event based (EB) system consists of allowing some components *called subscribers* to express their interests in some kind of information, while allowing other components *called publishers* to publish this information. The EB system is responsible for matching publications to subscriptions and forwarding them to interested subscribers. An application that includes such subscribers and publishers is called an *event based application*. The importance of the EB paradigm is witnessed by the increasing number of domains and tools in which it is exploited. Examples of such domains/tools are programming environments (e.g. Smalltalk), operating systems (e.g. AppleEvents [3]), communication middleware (e.g. Corba [14], Siena [7], JEDI [9], Elvin [29]), integration frameworks (e.g. OLE [5], JavaBeans [26], FIELD [25], SunSoft [28], Polylith [24], ISIS [4], Yeast [19]), and message oriented middleware (e.g. see [21]).

While considerable effort has gone into techniques for composition of software based on procedure invocation [10,15], shared data [8,22], and message passing [6,20], there is no established method for developing correct applications based on the EB paradigm. As a result, applications based on this paradigm are often developed in an ad-hoc and informal manner. Although it is not expected that all developers carry out formal techniques throughout the whole design and implementation process, they may use the intuition that has been built up during the development of the supporting techniques [12]. Further, formal techniques can be applied in relaxed versions as lightweight formal methods [2,6].

This paper proposes a novel formal approach for building correct applications using the EB paradigm. This approach is called *LECAP*: **L**ogic of **E**vent **C**onsumption **A**nd **P**ublication. This logic is compositional; hence, intrinsically oriented towards construction of complex systems. LECAP is based on Jones's rely/guarantee [17,27,32] program derivation technique which is extended in two respects. First, we extend the specification of a program to include announcement-conditions (ann-conditions). These are conditions that specify which events a program is allowed to announce. Next, we provide a rule for composing separately developed specifications into one large specification.

Let us assume that we want to build a software system that satisfies the requirements $\phi_1, \ldots, \phi_n$. Our methodology consists of four steps:

1. Identify components necessary for constructing the system.
2. Develop the formal specifications $S_1, \ldots, S_n$ of these components and verify some local properties of these specifications. These formal specifications consist in pre-conditions, post-conditions, rely-conditions, guarantee-conditions (guar-condition), wait-conditions, and ann-conditions.
3. Compute the formal specification S of the whole system using the specifications $S_1, \ldots, S_n$ and our composition rule. The specification S is computed such that any application which refines it satisfies the requirements $\phi_1, \ldots, \phi_n$.
4. Separately refine the specifications $S_1, \ldots, S_n$ to some implementations $I_1, \ldots, I_n$.

It is important to stress that the development of $I_1, \ldots, I_n$ can be performed by different teams that know nothing about each other. Each of them simply receives some specification S_i and is required to deliver some code that satisfies this specification. In other words, $I_1, \ldots, I_n$ might be off-the-shelf components that satisfy the requirements $S_1, \ldots, S_n$. Indeed, this is one of the expected benefits of the loose coupling of components.

The contribution of the paper is following:

1. We provide a formal definition of a language that supports development of event based applications. This language is called *the LECAP programming language*. The presentation of this language is important since it provides some added value compared to the specification provided in [12,11]. In particular, the LECAP programming language is a parallel programming language with synchronization constructs while that of Dingel et al. [12,11] is a sequential while-language. Further, the LECAP programming language doesn't require constructs such as the "consume" statement introduced in [12,11].
2. We provide a formal specification and semantics of an event based system. We believe that this specification is simpler compared to [12,11]: no pending event infrastructure is required. The capability of delaying events is obtained naturally from the programming language through the synchronization construct.
3. A method for the specification of event based applications is provided. Besides the pre-, post-, rely-, guar-, and wait-conditions, we introduce the ann-conditions that specify the kind of events a component is allowed to announce.
4. We present a rule for composing specifications (of components) into specification of systems.
5. We present two examples that illustrate the application of the technique.

The remainder of the paper is organized as follows. The next section (Sect. 2) presents related approaches. Section 3 provides the formal definition of the LECAP programming language. Section 4 presents a logic of specifying event based programs and a rule for composing these specifications. Section 5 presents some discussions. Section 6 presents two examples that illustrate our approach and Section 7 concludes the paper.

2 Related Work

Although the event based paradigm is at the heart of countless software systems, not much work has been presented on building correct applications using this paradigm. There are three main research areas that are related to our work.

The first related area concerns event broadcasting. A significant amount of work has been done on this topic that led to a number of theories such as calculi of broadcasting systems. Examples of such calculi are the $b\pi$-calculus [13] and the CBS [23] (calculus of broadcasting systems). The issue in such works is

how to achieve fault tolerance through replication. All the components in such a system are interested in all events. The requirements are thus different from that of event based systems where each component specifies the kind of events it is interested in.

The second related area of research concerns construction of parallel programs. Jones's rely/guarantee [17] (extended e.g. by Stolen [27] and Xu [32]) and the work of Owicki/Gries [22] are among the approaches that have influenced this area. Our work is strongly based on these two works. We extend the concept of rely/guarantee specification technique through ann-conditions. We also borrow auxiliary variables from Owicki/Gries and Stolen [27] to formulate ann-conditions.

The third area of work is about verifying the correctness of event based applications. The only work we are aware of is by Dingel et al. [12,11]. A method for reasoning about event based applications is proposed. This approach, which we call Dingel's approach is also based on Jones's rely/guarantee paradigm. To illustrate the contribution of the paper, we give a summary of Dingel's approach and some of its shortcomings.

Let $S = (M, V, EM, Ex)$ denotes a system consisting of a set of methods M, a set of global variables V, a binding of methods to events EM, and a set of external events Ex. Further, a specification is a 4-tuple (P, R, G, Q), where P, R, G, Q denote the pre-, rely-, guar-, and post-conditions. To show that the system S satisfies some partial correctness property T, 4 steps are required:

1. Define the pre-, rely, and post-conditions of the system: P, R, Q.
2. For each method $m \in M$, define the guarantee conditions G_m and $G_{M\setminus\{m\}}$ such that
   (m, V, EM, Ex) and $(M \setminus \{m\}, V, EM, Ex)$ satisfy $(P, R \vee G_{M\setminus\{m\}}, G_m, Q)$
3. Conclude using rely/guarantee soundness that (M, V, EM, Ex) satisfies $(P, R, \bigvee_{m\in M} G_m, Q)$
4. Show that any system that satisfies $(P, R, \bigvee_{m\in M} G_m, Q)$ also satisfies T.

This approach has a number of shortcomings.

1. It assumes a programming language with a *consume* construct. Each method must start with this statement that specifies which events the method is interested in. Dingel et al. use the consume construct to model invocation of methods by the EB system and to trace changes in the pending event infrastructure. They, however, recognize that this construct "introduces an unnecessary dependency between the event-method binding and the program of a method.[12,11]" Further, no real programming language or event based system needs such a construct.
2. The underlying specification technique is based on a pending event infrastructure. The primary intent of an EB system is not to queue events, but to dispatch them to subscribers. Queuing events results from the fact that an EB system might not be able to forward events at the speed at which they are received. Hence, we suggest that, although it may be important to take it into consideration at the implementation level, a mechanism for queuing

events should only influence the abstract model in a such a way that it does not complicate the reasoning too much.

3. Dingel et. [12,11] assume in their work that when a program is running it cannot be triggered anymore. No mechanism is however given for achieving this. On the other hand there are applications where such a limitation is not acceptable (e.g. reservation systems).
4. The approach doesn't take the definition of new subscriptions into consideration. A static binding EM is assumed. In this sense, the approach seems to miss a fundamental aspect of the event based paradigm which is (because of loose coupling) to ease the integration of new components.
5. Dingel's approach is intended for a-posteriori verification of systems instead of stepwise construction of systems: components of the completed programs are verified in isolation and then put together where general properties are proved. Jones [17] argues that such approaches are unacceptable as program development methods: erroneous design decisions taken in early steps are propagated until the system is implemented and proven.
   Although Dingel et al. [12,11] do not claim to propose a method for the stepwise construction of systems, the fact that their approach is based on Jones's rely/guarantee method for the construction of interfering programs may lead one to expect that it can also be used for such a purpose. To see why this is difficult, let us consider the following development method naively derived from the above reasoning technique:
   To construct a system S that satisfies some partial correctness property T, 6 steps must be followed:
   a) Define the pre-, rely, and post-conditions P, R, Q of the system.
   b) Identify the set of methods M of the system.
   c) For each method $m \in M$ (not yet implemented), define the guarantee conditions G_m and $G_{M\setminus\{m\}}$ such that (m, V, EM, Ex) and $(M \setminus \{m\}, V, EM, Ex)$ satisfy $(P, R \vee G_{M\setminus\{m\}}, G_m, Q)$
   d) Conclude that (M, V, EM, Ex) satisfies $(P, R, \bigvee_{m\in M} G_m, Q)$
   e) Show that any system that satisfies $(P, R, \bigvee_{m\in M} G_m, Q)$ also satisfies T.
   f) Now, refine each method m to some implementation.
   This approach, however, doesn't work since there is nothing that relates the specifications $(P, R \vee G_{M\setminus\{m\}}, G_m, Q)$ of the different methods to each other. This relation should be provided by the event based system. The methods should communicate with each other through the event based system by announcing and consuming events. This notion of announcement and consumption of events is, however, absent from the specification, hence the insufficiency of the specification and the inadequacy of the approach.

We propose an approach that overcomes these shortcomings.

3 The LECAP Programming Language

This section introduces the LECAP programming language. This language is used for the development of while-parallel programs with shared variables. The

particularly of LECAP programs is that they may communicate through an event based system. We define the syntax of the language and its operational semantics. We also give a definition of the concept of event based systems.

3.1 Syntax

A LECAP program is a while-program augmented with parallel, synchronization, and event publication constructs. Its syntax can be defined as follows:

$P ::= x := e \mid P_1; P_2 \mid$ **if** b **then** P_1 **else** P_2 **fi** $\mid$ **while** b **do** P **od** $\mid$ **await** b **then** P **end** $\mid P_1 \| P_2 \mid$ **announce(e)** $\mid$ **skip**.

Three constructs in this language need some explanations: the parallel construct, the await construct, and the announce construct. The first models nonderministic interleaving of the atomic actions of P_1 and P_2. Synchronization and mutual exclusion are achieved using the await construct. The announce construct allows announcement of events. It is intended for the notification of the EB system which in turn triggers some other programs. The execution of announce is limited to sending the event to the EB system. The sequential composition $P_1; P_2$ can thus be defined in the usual way as a program that first behaves as P_1 and follows as P_2 if P_1 terminates. To simplify the deduction rules, it is required that variables used in the boolean test cannot be accessed by programs running in parallel. This constraint can be removed as discussed in [27]. We say that a program z_0 is a subprogram of another program z iff z can be written in one the following forms:

- $z_1; z_0; z_2$;
- **if** b **then** z_1 **else** z_2 **fi**, with z_0 a subprogram of z_1 or z_2;
- **while** b **do** z_1 **od**, with z_0 a subprogram of z_1;
- $z_1 \| z_2$, with z_0 a subprogram of z_1 or z_2;
- **await** b **do** z_1 **od** , with z_0 a subprogram of z_1.

3.2 Event Based System

Although there are various paradigms that make up an event based system in practice, not all of them are needed at the abstract level. We construct an abstract model based on a set of programs, a set of events, a binding, and a set of variables. An event is a piece of data that may be published by a program. Subscriptions are templates for allowing categorization of events. The set of programs is the set of handlers of events. Such programs are triggered when an event is announced that matches one of their subscriptions. The programs in an event based systems may not only communicate (by announcing and consuming events), but they may also share some variables.

Definition 1. *An event based system is a 4-tuple* $(\mathcal{E}, \mathcal{M}, \vartheta, \mathcal{B})$ *composed of a set of events* $\mathcal{E}$*, a set of programs* $\mathcal{M}$*, a set of global variables* ϑ *shared among programs in* $\mathcal{M}$*, and a binding* $\mathcal{B}$ *which maps each program to its set of subscriptions.*

The behavior of an EB system consists of providing facilities for announcing and receiving events. Programs announce events that are dispatched to some other programs. The purpose of an event is thus to trigger some programs. The process of determining which programs are interested in an event is called matching. A matching is performed between an event and a subscription. This is a query that describes the interest of a program in receiving some events. Such subscriptions are typically assertions that characterize events. They can be viewed as total functions defined on the set of events and returning true or false. An example of subscription is: λx : *x starts with 'John'*. An example of event that matches this subscription is: *'John is leaving the office'*.

Definition 2. *Assuming an EB system* $(\mathcal{E}, \mathcal{M}, \mathcal{B})$*, a subscription s is a total function from* $\mathcal{E}$ *to* $\{True, False\}$.

$\mathcal{B}(z)$ denotes the set of subscriptions of a program z and determines the set of events z is interested in.

Definition 3. *Given an event e, we define* $subscribers(e) = \{z \in \mathcal{M} \mid \exists s \in \mathcal{B}(z), s(e) = true\}$ *as the set of programs that are interested in the event e.*

Among the set of events $\mathcal{E}$, there may be some external events. These are events that are announced by programs not in $\mathcal{M}$. We denote the set of external events as $\mathcal{E}_x$ and the set of programs that subscribed to some of these events as $\mathcal{M}_x$. Formally, $\mathcal{M}_x = \bigcup_{e \in \mathcal{E}_x} subscribers(e)$.

3.3 Operational Semantics

We give the operational semantics of the LECAP programming language in the style of [1]. A state maps all programming variables to values and a configuration is a pair $\langle p, s \rangle$ where p is a program and s is a state. The semantics of the LECAP programming language is given relative to an EB system $(\mathcal{E}, \mathcal{M}, \mathcal{B})$. In the sequel z denotes a program that is different from the empty program ϵ.

An environment transition $\xrightarrow{v}$ is the least binary relation on configurations such that:

- $\langle z,\ s_1 \rangle \xrightarrow{v} \langle z,\ s_2 \rangle$, environment transitions are only allowed to modify the state of EB systems.

A program transition $\xrightarrow{i}$ is the least binary relation on configurations such that one of the following holds:

- $\langle skip,\ s \rangle \xrightarrow{i} \langle \epsilon,\ s \rangle$,
- $\langle u := r,\ s \rangle \xrightarrow{i} \langle \epsilon,\ s[u/r\,] \rangle$, where $s[u/r\,]$ denotes the state obtained from s by mapping the variable u to the value r and leaving all other state variables unchanged,

- $\langle announce(e); z,\ s\rangle \xrightarrow{i} \langle \| subscribers(e) \cup \{z\},\ s\rangle$. The effect of announcing an event is to trigger the set of programs that subscribed to this event and execute them in parallel with the remainder of the announcing program. Programs triggered by an event are part of the running program and their transitions are therefore internal transitions.

- $\langle z_1; z_2,\ s_1\rangle \xrightarrow{i} \langle z_2,\ s_2\rangle$ if $\langle z_1,\ s_1\rangle \xrightarrow{i} \langle \epsilon,\ s_2\rangle$ and **announce(e)** is not a subprogram of z_1,

- $\langle z_1; z_2,\ s_1\rangle \xrightarrow{i} \langle z_3; z_2,\ s_2\rangle$ if $\langle z_1,\ s_1\rangle \xrightarrow{i} \langle z_3,\ s_2\rangle$, $z_3 \neq \epsilon$ and **announce(e)** is not a subprogram of z_1,

- $\langle$if b then z_1 else $z_2,\ s\rangle \xrightarrow{i} \langle z_1,\ s\rangle$ if $s \models b$,

- $\langle$if b then z_1 else $z_2,\ s\rangle \xrightarrow{i} \langle z_2,\ s\rangle$ if $s \models \neg b$,

- $\langle$while b do z od, $s\rangle \xrightarrow{i} \langle z;$ while b do z od, $s\rangle$ if $s \models b$,

- $\langle$while b do z od, $s\rangle \xrightarrow{i} \langle \epsilon,\ s\rangle$ if $s \models \neg b$,

- $\langle \{z_1 \parallel z_2\},\ s_1\rangle \xrightarrow{i} \langle z_2,\ s_2\rangle$ if $\langle z_1,\ s_1\rangle \xrightarrow{i} \langle \epsilon,\ s_2\rangle$,

- $\langle \{z_1 \parallel z_2\},\ s_1\rangle \xrightarrow{i} \langle z_1,\ s_2\rangle$ if $\langle z_2,\ s_1\rangle \xrightarrow{i} \langle \epsilon,\ s_2\rangle$,

- $\langle \{z_1 \parallel z_2\},\ s_1\rangle \xrightarrow{i} \langle \{z_3 \parallel z_2\},\ s_2\rangle$ if $\langle z_1,\ s_1\rangle \xrightarrow{i} \langle z_3,\ s_2\rangle$, $z_3 \neq \epsilon$.

- $\langle \{z_1 \parallel z_2\},\ s_1\rangle \xrightarrow{i} \langle \{z_1 \parallel z_3\},\ s_2\rangle$ if $\langle z_2,\ s_1\rangle \xrightarrow{i} \langle z_3,\ s_2\rangle$, $z_3 \neq \epsilon$.

- $\langle$ await b do z_1 od , $s_1\rangle \xrightarrow{i} \langle \epsilon,\ s_n\rangle$ if $s_1 \models b$, and there exists a list of configurations $\langle z_2, s_2\rangle$,...,$\langle z_{n-1}, s_{n-1}\rangle$, such that $\langle z_{n-1}, s_{n-1}\rangle \xrightarrow{i} \langle \epsilon,\ s_n\rangle$ and for all $1 < k < n, \langle z_{k-1}, s_{k-1}\rangle \xrightarrow{i} \langle z_k,\ s_k\rangle$,

The meaning of an await statement is not very clear when its body does not terminate [31]. When it however terminates the final state is required to satisfy the post-condition. Given that we are not interested (in this work) in non-terminating programs we can stipulate that any computation of an await-statement has a finite length.

Besides the state of the system that programs may read and update, they also have local variables that are hidden such that environment transitions are not allowed to access them. We do not model this concept in our work since it has no impact on our rules.

Definition 4. *A configuration c_1 is disabled if there is no configuration c_2 such that $c_1 \xrightarrow{i} c_2$.*

Definition 5. *A computation is a possibly infinite sequence of environment and program transitions:* $\langle z_1, s_1\rangle \xrightarrow{l_1} \ldots \xrightarrow{l_{k-1}} \langle z_k, s_k\rangle \xrightarrow{l_k} \ldots$ *such that the final configuration is disabled if the sequence is finite. A computation is blocked if it is finite and the program of the last computation is different from* ϵ*. A computation terminates iff it is finite and the program of the last configuration is* ϵ*.*

The above operational semantics doesn't explicitly discuss the case of events announced by the environment (including external events). The programs triggered by these events are part of the environment and their transitions are environment transitions.

Given a computation σ, then $Z(\sigma)$, $S(\sigma)$ and $L(\sigma)$ are the projections to sequences of programs, states and transition labels, while $Z(\sigma_k)$, $S(\sigma_k)$ and $L(\sigma_k)$ and σ_k denote respectively the k'th program, the k'th state, the k'th transition label and the k'th configuration. The number of configurations in σ is denoted $len(\sigma)$. If σ is infinite, then $len(\sigma) = \infty$.

In the sequel, $cp[z]$ denotes the set of computations σ such that $Z(\sigma_1) = z$. These computations are called computations of z.

4 Specification Language

We show how rely- and guar- conditions can be extended and used for the specification of LECAP programs. These programs are parallel programs that might use synchronization constructs and are based on the event paradigm. Specifically, we show that the quintessence of the logic of specified programs proposed by Stolen [27] can be reused.

In the style of VDM [18], hooked variables are used to denote an earlier state. For any variable v of type $\mathcal{T}$, there exists a corresponding hooked variable $\overset{\square}{v}$ of type $\mathcal{T}$. Hooked variables cannot appear in programs.

An assertion is a boolean formula on states. Let P be an assertion and s_1 and s_2 be two states. We write $(s_1, s_2) \models P$ if P is true when each hooked variable v in P is assigned the value $s_1(v)$ and each unhooked variable is assigned the value $s_2(v)$. If P has no hooked variable, it may be thought of as the set of all states, such that $s \models P$. We write $\models P$ if P is valid in the actual structure.

4.1 Specification

If $(\mathcal{L}, \mathcal{M}, \mathcal{V}, \mathcal{B})$ is an EB system, a specification is of the form $(\mathcal{L}, \mathcal{M}, \mathcal{V}, \mathcal{B}) :: (P, R, W, G, E, A)$, where the *pre-condition* P and the *wait-condition* W are unary assertions while the *rely-condition* R, the *guar-condition* G, and the *post-condition* E are binary assertions. The ann-condition A is a set of assertions defined on computations. It characterizes the kind of events a program may announce.

Definition 6. *Given a computation* σ*, an assertion* Q*, and an event* e*,* Q *conditions the announcement of* e *in* σ *(or* σ *satisfies* $Q \prec e$*) iff for any state* s

found along σ that satisfies Q there is a program transition in the remainder of σ of one of the following forms:

- $\langle announce(e); z,\ s\rangle \xrightarrow{i} \langle \| subscribers(e)\ \cup \{z\},\ s\rangle$.
- $\langle announce(e) \| z,\ s\rangle \xrightarrow{i} \langle \| subscribers(e)\ \cup \{z\},\ s\rangle$.
- $\langle z \| announce(e),\ s\rangle \xrightarrow{i} \langle \| subscribers(e)\ \cup \{z\},\ s\rangle$.

There should be exactly one such configuration.

Definition 7. *An ann-condition is a set of formula of the form $Q \prec e$. A computation satisfies an ann-condition $A = \{Q_i \prec e_i\}_i$ iff it satisfies each $Q_i \prec e_i \in A$*

The restriction on the number announcements of an event e along a computation can be surmounted by labelling events.

If X is a set of variables and s_1, s_2 are two states, then $s_1 \overset{X}{=} s_2$ signifies that for all variables x in X, $s_1(x) = s_2(x)$ while $s_1 \overset{X}{\neq} s_2$ means that there exists x in X such that $s_1(x) \neq s_2(x)$.

Definition 8. *Given an EB system $(\mathcal{L}, \mathcal{M}, \vartheta, \mathcal{B})$, a pre-condition P, and a rely-condition R, then $ext[\vartheta, P, R]$ denotes the set of computations σ such that:*

- $S(\sigma_1) \models P$,
- *for all $1 \leq j < len(\sigma)$, if $L(\sigma_j) = e$ and $S(\sigma_j) \overset{\square}{\neq} S(\sigma_{j+1})$ then $(S(\sigma_j), S(\sigma_{j+1})) \models R$.*

The previous definition characterizes computations that satisfy the pre-condition and are subject to environment transitions. Informally, 1) the initial state must satisfy the pre-condition, and 2) any environment transition which changes the global state must satisfy the rely-condition.
A specification also characterizes commitments of the implementations:

Definition 9. *Assuming an event based system $(\mathcal{L}, \mathcal{M}, \vartheta, \mathcal{B})$, a guar-condition G, and a post-condition E, a wait-condition W, and an ann-condition A then $int[\vartheta, G, E, W, A]$ denotes the set of computations σ such that:*

- *for all $1 \leq j < len(\sigma)$, if $L(\sigma_j) = i$ and $S(\sigma_j) \overset{\square}{\neq} S(\sigma_{j+1})$ then $(S(\sigma_j), S(\sigma_{j+1})) \models G$,*
- *if $Z(\sigma_{len(\square)}) = \epsilon$ then $(S(\sigma_1), S(\sigma_{len(\square)})) \models E$,*
- *if $Z(\sigma_{len(\square)}) \neq \epsilon$ then $Z(\sigma_{len(\square)}) \models W$*
- *$len(\sigma) \neq \infty$*
- *σ satisfies the ann-condition A.*

The above definitions implicitly take into consideration the case of a program z_e triggered by an event e announced by z. The triggered program z_e is in fact part of the running program which becomes $z_e \| z_1$ where z_1 is the remainder of z (after the the announcement). However, in the parallel composition $z_e \| z_1$, z_e and z_1 are part of the environment of each other. They are therefore required to satisfy the rely-condition of each other. An interference free composition must hence require that they coexist.

4.2 Judgments

Definition 10. *Given an event based system* $(\mathcal{L}, \mathcal{M}, \vartheta, \mathcal{B})$*, a judgment is a pair consisting of a program* $z \in \mathcal{M}$ *and a specification* $(\mathcal{L}, \mathcal{M}, \vartheta, \mathcal{B}) :: (\mathcal{P}, \mathcal{R}, \mathcal{G}, \mathcal{E}, \mathcal{W}, \mathcal{A})$*. Such a judgment is denoted* $z \models (\mathcal{L}, \mathcal{M}, \vartheta, \mathcal{B}) :: (P, R, G, E, W, A)$*.*

Definition 11. *Let us assume an event based system* $(\mathcal{L}, \mathcal{M}, \vartheta, \mathcal{B})$*; A judgment* $z \models (\mathcal{L}, \mathcal{M}, \vartheta, \mathcal{B}) :: (P, R, G, E, W, A)$ *is valid iff* $cp[z] \cap ext[\vartheta, P, R] \subseteq int[\vartheta, G, E, W, A]$ *.*

We extend the concept of judgment to the whole event based system and say that $\models (\mathcal{L}, \mathcal{M}, \vartheta, \mathcal{B}) :: (P, R, G, E, W, A)$ is valid iff the judgment $z \models (\mathcal{L}, \mathcal{M}, \vartheta, \mathcal{B}) :: (P, R, G, E, W, A)$ is valid for any program $z \in \mathcal{M}$.

4.3 Composition

This section aims at formulating the rule for the composition of LECAP specifications. A LECAP program that satisfies its specification is called a correct program (w.r.t. its specification).

In the remainder, if $\mathcal{B}$ is a binding, z a program, and e an event, $\mathcal{B}\dagger\{z \mapsto e\}$ represents the binding obtained from $\mathcal{B}$ by subscribing the program z to the event e. By extension of this notation, if $\mathcal{S}$ is an EB system, $\mathcal{S}\dagger\{z \mapsto e\}$ represents $\mathcal{S}$ with its binding $\mathcal{B}$ replaced by $\mathcal{B}\dagger\{z \mapsto e\}$. $I_{\square}$ denotes the assertion $\bigwedge_{x \in \square} x = \overset{\square}{x}$. $I_{\square}$ is thus an assertion that states that the value of no variable in ϑ changed. If A, B, and C are some binary assertions, $B \mid C$ denotes the assertion characterizing the relational composition of B and C i.e. $(s_1, s_3) \models B \mid C$ iff there exists s_2 such that $(s_1, s_2) \models B$ and $(s_2, s_3) \models C$. B^* denotes the transitive closure of B. A^B denotes an assertion that characterizes any state that can be reached from a state A by a finite number of B steps. $\mathcal{S}$ represents the event based system $(\mathcal{L}, \mathcal{M} \cup \{z_1, z_2\}, \vartheta, \mathcal{B})$ and $\mathcal{S}_0$ represents the event based system $(\mathcal{L}, \mathcal{M}, \vartheta, \{\})$ with an empty binding. This means that no program in $\mathcal{S}_0$ is subscribed to any event. Announcing an event has an effect neither on the state of the system nor on running programs. In this case, the ann-condition A in a specification such as (P, R, G, E, W, A) is not relevant (denoted $\perp$). The set of deduction rules proposed by Stolen [27] is thus applicable. We illustrate the consequence, the parallel and the await rules below. They are further used for the construction of our composition rule.

Consequence rule:

$$\frac{\begin{array}{l} P_2 \Rightarrow P_1 \\ R_2 \Rightarrow R_1 \\ W_1 \Rightarrow W_2 \\ G_1 \Rightarrow G_2 \\ E_1 \Rightarrow E_2 \\ z \models \mathcal{S}_0 :: (P_1 \text{‹} R_1 \text{‹} G_1 \text{‹} E_1 \text{‹} W_1 \text{‹} A_1) \end{array}}{z \models \mathcal{S}_0 :: (P_2 \text{‹} R_2 \text{‹} G_2 \text{‹} E_2 \text{‹} W_2 \text{‹} \perp)}$$

Parallel rule:

$$\frac{\begin{array}{l} \neg(W_1 \wedge W_2) \wedge \neg(W_2 \wedge E_1) \wedge \neg(W_1 \wedge E_2) \\ G_1 \Rightarrow R_2 \\ G_2 \Rightarrow R_1 \\ z_1 \models \mathcal{S}_0 :: (P \text{‹} R_1 \text{‹} W \vee W_1 \text{‹} G_1 \text{‹} E_1 \text{‹} A_1) \\ z_2 \models \mathcal{S}_0 :: (P \text{‹} R_2 \text{‹} W \vee W_2 \text{‹} G_2 \text{‹} E_2 \text{‹} A_2) \end{array}}{\{z_1 \| z_2\} \models \mathcal{S}_0 :: (P \text{‹} R_1 \wedge R_2 \text{‹} W \text{‹} G_1 \vee G_2 \text{‹} E_1 \wedge E_2 \text{‹} \perp)}$$

The consequence rule allows refinement of specifications by strengthening the assumptions and weake-ning the commitments.

The parallel rule ensures parallelization of programs. The program $z_1 \| z_2$ can be derived from z_1 and z_2 if they satisfy the above specifications. An important requirement for z_1 and z_2 to coexist is that the guar-condition of one implies the rely-condition of the other. Further, not both processes should be in a waiting status as well as if the execution of one of them is completed, the other should not be waiting.

Sequential Rule

$$\frac{\begin{array}{l}\square \\ P_1 \wedge E_1 \Rightarrow P_2 \\ z_1 \models \mathcal{S}_0 :: (P_1 \, ^\frown R \, ^\frown W \, ^\frown G \, ^\frown E_1 \, ^\frown A_1) \\ z_2 \models \mathcal{S}_0 :: (P_2 \, ^\frown R \, ^\frown W \, ^\frown G \, ^\frown E_2 \, ^\frown A_2)\end{array}}{z_1; z_2 \models \mathcal{S}_0 :: (P_1 \, ^\frown R \, ^\frown W \, ^\frown G \, ^\frown E_1 | E_2 \, ^\frown \bot)}$$

Await Rule:

$$\frac{z \models \mathcal{S}_0 :: (P^R \wedge b \, ^\frown \text{ false,false,true },(G \vee I_{\square}) \wedge E \, ^\frown A)}{\text{await } b \text{ do } z \text{ od} \models \mathcal{S}_0 :: (P \, ^\frown R \, ^\frown P^R \wedge \neg b \, ^\frown G \, ^\frown R^* | E | R^* \, ^\frown \bot)}$$

The sequential rule is quite similar to that of sequential programming. It essentially requires that the post-condition of the first program implies the pre-condition of the second. The resulting specification is that of a program whose computations start in a state satisfying the first pre-condition, ends in a state satisfying the second post-condition, and is such that it contains a state satisfying the first post-condition.

The intent of the await-rule is to allow programs to synchronize on resources. Assume we want to construct a synchronizing program that satisfies $(P, R, P^R \wedge \neg b, G, R^* | E | R^*)$. This program needs to block when b is false. Hence, it executes when b is true. However, if the await-program executes when b is true, the await-body needs to have b as conjunct in its pre-condition. Further, if the await-program has P as pre-condition, the await-body needs to have P^R as conjunct in its pre-condition. The reason for this is that while the program is waiting for b to be true, some environment transitions may be performed that modify the state of the system in a way that satisfies the rely condition. The pre-condition of the await-body is thus $P^R \wedge b$. The rely- and wait-conditions of the wait-body results from the fact that we want the program to be executed in an atomic step (without any interference). Since the program is executed in an atomic step, if we want the await-program to guarantee G, the post-condition of its unique step (the await-body) needs to either leave all variables unchanged or satisfy G. Of course, the post-condition of the await-body also needs to satisfy E.

We now consider the above rules; still with an empty binding, but taking ann-conditions into consideration. The computations of rely- guar-, pre-, post- and wait-conditions remain the same as above. In the following, Q_1, Q_2 and Q designate some assertions, while e_i are events. $Fr(Q)$ denotes the set of free variables in the assertion Q.

Parallel Rule

$$\frac{\begin{array}{l}Fr(Q_i) \cap Fr(Q_j) = Fr(Q_i) \cap \square = Fr(Q_j) \cap \square = \emptyset \\ e_i \neq e_j \, ^\frown \quad i \in [1 \, ^\frown n] \, ^\frown \quad j \in [n+1 \, ^\frown m] \\ \neg(W_1 \wedge W_2) \wedge \neg(W_2 \wedge E_1) \wedge \neg(W_1 \wedge E_2) \\ G_2 \Rightarrow R_1 \\ G_1 \Rightarrow R_2 \\ z_1 \models \mathcal{S} :: (P \, ^\frown R_1 \, ^\frown W \vee W_1 \, ^\frown G_1 \, ^\frown E_1 \, ^\frown \{Q_i \prec e_i\}_1^n) \\ z_2 \models \mathcal{S} :: (P \, ^\frown R_2 \, ^\frown W \vee W_2 \, ^\frown G_2 \, ^\frown E_2 \, ^\frown \{Q_j \prec e_j\}_{n+1}^m)\end{array}}{\{z_1 \| z_2\} \models \mathcal{S} :: (P \, ^\frown R_1 \wedge R_2 \, ^\frown W \, ^\frown G_1 \vee G_2 \, ^\frown E_1 \wedge E_2 \, ^\frown \{Q_i \prec e_i\}_1^m)}$$

Sequential Rule

$$\frac{\begin{array}{l} Fr(Q_i) \cap Fr(Q_j) = Fr(Q_i) \cap \square = Fr(Q_j) \cap \square = \emptyset \\ e_i \neq e_j{}^{\frown} \quad i \in [1{}^{\frown} n]{}^{\frown} \quad j \in [n+1{}^{\frown} m] \\ P_1 \wedge E_1 \Rightarrow P_2 \\ z_1 \models \mathcal{S} :: (P_1{}^{\frown} R{}^{\frown} W{}^{\frown} G{}^{\frown} E_1{}^{\frown} \{Q_i \prec e_i\}_{n+1}^{m}) \\ z_2 \models \mathcal{S} :: (P_2{}^{\frown} R{}^{\frown} W{}^{\frown} G{}^{\frown} E_2{}^{\frown} \{Q_i \prec e_i\}_{1}^{n}) \end{array}}{z_1; z_2 \models \mathcal{S} :: (P_1{}^{\frown} R{}^{\frown} W{}^{\frown} G{}^{\frown} E_1 | E_2{}^{\frown} \{Q_i \prec e_i\}_1^m)}$$

If Q_1 (resp. Q_2) conditions the announcement of e_1 (resp. e_2) in z_1 (resp. z_2), the parallel composition of z_1 and z_2 is such that Q_1 (resp. Q_2) conditions the announcement of e_2 (resp. e_1) in $z_1 \| z_2$. There are two reasons why this is true:

- no event is announced by both programs. If an event was announced by both programs, the parallel composition would yield a program that announces the same event twice. This is not compatible with the definition of $Q \prec e$.
- the set of variables in any assertion in the ann-condition of z_1 is disjoint from the global state and from the set of variables of any assertion in the ann-condition of z_2 (and vice-versa). To understand the necessity of this restriction, assume a program z_1 that satisfies $x > 2 \prec e_1$ and some of its computations σ_1 have a configurations such that its state satisfies $x > 2$ holds. By the definition of announcement conditions there is an announcement transition in σ_1. On the other hand there may be a program z_2 such that when composing in parallel with z_1 there is no state any more that satisfies $x > 2$. The resulting program $z_1 \| z_2$ announces a program although the condition $x > 2$ is not satisfied. This doesn't happen when the assertions are based on auxiliary variables that appear only in one program.

The enhancement of the await-rule is trivial: if the announcement of an event is conditioned by Q in z, this event remains conditioned by Q when we embed z in an await construct.

Await rule:

$$\frac{z \models \mathcal{S}_0 :: (P^R \wedge b{}^{\frown} \text{ false,false,true },(G \vee I_{\square}) \wedge E{}^{\frown} \{Q_i \prec e_i\}_1^m)}{\text{await } b \text{ do } z \text{ od} \models \mathcal{S}_0 :: (P{}^{\frown} R{}^{\frown} P^R \wedge \neg b{}^{\frown} G{}^{\frown} R^* | E | R^{*}{}^{\frown} \{Q_i \prec e_i\}_1^m)}$$

Starting from an empty binding, we now need to successively add subscriptions to the EB system. Before investigating the composition rule, we give one more definition. Let us denote the set of events that a program possibly announces as *events*(z); let also the set $\gamma(z) =$*subscribers(events*(z)*)* be the set of programs subscribed on the events that the program z announces. $\gamma^*(z)$ denotes the transitive closure of γ defined as $\gamma(z) \cup \bigcup_{s \in \square(z)} \gamma^*(s)$

Definition 12. *The binding $\mathcal{B}$ of an EB system $(\mathcal{L}, \mathcal{M}, \wp, \mathcal{B})$ is well founded iff for any program z, $z \notin \gamma^*(z)$. A well founded binding will be denoted wf $\mathcal{B}$.*

The intent of the above definition is to avoid infinite loops in EB systems. The simplest such case is when a program subscribes to the events that itself announces. This restriction seems to be strong: a program may subscribe on events it

announces without producing infinite loops. We doubt on the necessity of such configurations and exclude them as may complicate the composition rule.

Computing the specification of a system out of those of its components consists of successively adding new subscriptions to the EB system. The process starts with a system with an empty binding. After adding a new subscription, the composition rule is applied and new specifications are derived. The algorithm is following:

```
To subscribe program z2 to  event e do:
  add  the entry (z2 , e) to the the binding;
    for each program z that announces e, do:
        apply the composition rule
        if necessary, update any specification that depends on z
```

The composition rule can now be given. For each subscription that is performed, it is required that the binding remains well founded. The claim of this rule is that if the programs z and z_2 are specified as shown in the premises while z_2 is not interested in e_1, subscribing z_2 to e_1 results in a program that behaves like z first and eventually (from a state satisfying I^{R_1}), behaves like $z_2 \| z$. This is a natural consequence of the semantics of event announcement.

Composition rule:

$$\frac{\begin{array}{l} Fr(Q_i) \cap Fr(Q_j) = Fr(Q_i) \cap \square = Fr(Q_j) \cap \square = \emptyset \\ \neg(W_1 \wedge W_2) \wedge \neg(W_2 \wedge I|E_1) \wedge \neg(W_1 \wedge E_2) \\ e_i \neq e_j \text{ ᶜ} \quad i \in [1 \text{ ᶜ} n] \text{ ᶜ} \; j \in [n+1 \text{ ᶜ} m] \\ \text{wf } \mathcal{B}\dagger\{z_2 \mapsto e_1\} \\ \square \\ P \wedge I|R^* \Rightarrow P_2 \wedge Q_1 \\ z \notin \text{subscribers}(e_1) \\ y \models \mathcal{S} :: (P \text{ ᶜ} R_1 \text{ ᶜ} W_1 \text{ ᶜ} G_1 \text{ ᶜ} I|E_1 \text{ ᶜ} \{Q_i \prec e_i\}_1^n) \\ z_2 \models \mathcal{S} :: (P_2 \text{ ᶜ} R_2 \text{ ᶜ} W_2 \text{ ᶜ} G_2 \text{ ᶜ} E_2 \text{ ᶜ} \{Q_i \prec e_i\}_{n+1}^m) \end{array}}{z \models \mathcal{S}\dagger\{z_2 \mapsto e_1\} :: (P \text{ ᶜ} R_1 \wedge R_2 \text{ ᶜ} W_1 \text{ ᶜ} G_1 \vee G_2 \text{ ᶜ} I|E_1 \wedge P_2|E_2 \text{ ᶜ} \{Q_i \prec e_i\}_1^m)}$$

4.4 Cause of Events

Before subscribing a program to an event, one needs to know the meaning of this event. The cause of an event is an assertion that characterizes the announcement of an event in the whole system. An event is announced in the system iff this assertion is true. Let $(\mathcal{M}, \mathcal{L}, \vartheta, \mathcal{B})$ be an EB system and A_z be the ann-condition of a program z in $\mathcal{M}$. We further denote the union of all ann-conditions as $\mathcal{A} =$. The following formula gives a formal definition:

$$cause(e) = \bigvee_{Q \in S} Q, \text{where } S = \{Q, Q \prec e' \in \bigcup_{z \in \mathcal{M}} A_z\}.$$

The formula says that the cause of an event is the disjunction of the assertions Q that condition the announcement of events with the same semantics as e. The formula is not applicable to events that may be announced by the environment since the environment may be non-deterministic.

5 Discussion

An important issue is the tractability of our logic. Let us consider a system in which z, and y announce the events e_1, and e_2 respectively while y is subscribed to e_1. If we further subscribe z_3 to e_2, y needs to be re-computed. Worst, all specifications that depend on y (in our case only z) need also be updated. In real project, the chain may be long and the composition can become painful. Fortunately, the difference between the different programs are syntactically clear. For instance after subscribing z_3 to y only a conjunct of the form $Q^P|T$ needs to be added to specifications that depend on y. Updating the specifications is thus a copy-paste process that can be easily mechanized. We believe that CASE tools for composing specifications can help solve this problem.

Another factor that influences the tractability of our approach is the order in which subscriptions are performed. It is obvious that if z_3 is subscribed to e_2 before y is subscribed to e_1, the changes are less significant. Thus, an initial step in composing the specification of a system out of those of its components is to find a suitable sequence of application of the composition rule.

This issue of tractability is not specific to our composition approach. Techniques of composition based on procedure invocation have comparable problems. The manifestation of such problems in practice is e.g. regression testing that tackles the issue of detecting which part of a system must be tested following the modification in another part of the system.

We argued at the beginning of this document that our approach does not require a pending event infrastructure. The question is thus, how to model such a requirement since it might be important in some cases to show e.g. that the result of an operation doesn't depend on the ordering of events. This can be done in our approach using the synchronization construct. Queuing or delaying an event until a condition Q is fulfilled means embedding the subscribed programs in an await construct conditioned by Q.

Our composition rule requires that bindings be well-founded. There are, however, cases where a program announces an event, and waits for another program to consume the event and send a result back. A typical such scenario is to simulate method invocation using the event based paradigm, which would make "caller" less strongly coupled to the "callee". To achieve this, we need an auxiliary program p and an auxiliary variable v. The purpose of p is simply to store the event it receives in v. Now, for achieving our method invocation, the caller first subscribes the auxiliary program p to events it would like to wait for. Next, the caller announces the event containing the parameters of the call and blocks (by means of the await construct). On the other side the callee is triggered by the event based system. After processing the event, the callee publishes an event to which the auxiliary program p is subscribed for which the caller is waiting. Once the auxiliary program is triggered it stores the received event in the related auxiliary variable such that the wait-condition of the caller now holds. The caller can continue its execution by reading the content of the auxiliary variable. We are working on generalizing this solution to make the various details transparent to the designers.

6 Examples

We consider an example similar to that of Dingel et al. [12,11]. The goal is to develop a system consisting in a buffer and a counter. Each time an element is added to the buffer, the counter must be incremented. Similarly, each time an element is removed from the buffer, the counter has to be decremented. We adopt two approaches for designing this system.

6.1 Example 1

We design a system with four programs. The first program (*add*) adds elements of type $\mathcal{T}$ to the buffer while the second program *incr* increments a counter *Count*. Similarly, *remove* removes elements from the buffer and *decr* decrements the counter. The global state is thus composed of *Buf* and *Count*.

Let us construct the event based system $\mathcal{S}_0 = (\mathcal{L}, \mathcal{M}, \square, \mathcal{B})$ first. We already have $\square = \{ Buf, Count\}$ and $\mathcal{M} = \{add, incr, remove, decr\}$. We deduce the empty binding $\mathcal{B} = \{add \mapsto \emptyset, incr \mapsto \emptyset, decr \mapsto \emptyset, remove \mapsto \emptyset\}$. We further extend the event based system with the set of auxiliary variables $\square_a = \{ Buf_{a1}, Buf_{a2}\}$ used in the ann-conditions. An event is a tuple consisting of an identification number and an element of type $\mathcal{T}$. The set of event is thus $\mathcal{E} = \{(id, elt) | id \in \mathbb{N} \wedge elt \in \mathcal{T}\}$. We access the identifier (resp. element) of an event *evt* using the notation *evt.id* (resp. *evt.elt*). A subscription is a total function defined on the set of events. An example of subscription (using the lambda notation) is: $s = \square e : e \triangleright id > 40$

The next step in the example is to give the formal specification of the programs *add* and *incr* denoted as S_a and S_i. In this example, we want each program to run alone à-la sequential programming. The rely-conditions of the programs are false while their wait-, and guar-conditions are true. If we assume *Buf* to be an unbounded buffer, then elements can always be inserted, hence the pre-condition is true. The post-condition can be defined as $Buf = e \triangleright elt + \overset{\square}{Buf}$. Each program that is started is done so with an event as input. We want the program *add* to announce the event $(1, e.elt)$ whenever an element is added in the buffer. The ann-condition is thus $\{ Buf_{a1} = \{e \triangleright elt\} \prec (1, e \triangleright elt)\}$. For this ann-condition to be satisfied when an element is added to the buffer the conjuncts $Buf_{a1} = \emptyset$ and $Buf_{a1} = \{e.elt\}$ must be added to the pre- and post-conditions respectively. S_a is thus expressed as: $S_a = \mathcal{S} :: (Buf_{a1} = \emptyset, false, false, false, Buf_{a1} = \{e \triangleright elt\} \wedge Buf = e \triangleright elt + \overset{\square}{Buf}, \{ Buf_{a1} = \{e \triangleright elt\} \prec (1, e \triangleright elt)\})$.
Similarly, we define the following specification of *incr*: $S_i = \mathcal{S}_0 :: (true, false, false, false, Count = \overset{\square}{Count} + 1, \emptyset)$. Using the consequence rule we can refine it by strengthening the pre-condition: $S_i = \mathcal{S}_0 :: (Buf = e \triangleright elt + \overset{\square}{Buf}, false, false, false, Count = \overset{\square}{Count} + 1, \emptyset)$.

We now subscribe *incr* to events with identifier equals to 1. This is done with the subscription query $\square x : x \triangleright id = 1$. Applying the composition rule we obtain the new definition of S_i: $S_a = \mathcal{S} :: (Buf_{a1} = \emptyset, false, false, false, Buf_{a1} = \{e \triangleright elt\} \wedge Buf = e \triangleright elt + \overset{\square}{Buf} \wedge Count = \overset{\square}{Count} + 1, \{ Buf_{a1} = \{e \triangleright elt\} \prec (1, e \triangleright elt)\})$.

A similar reasoning yields the following specifications S_r, and S_d of *remove* and *decr* respectively:

$S_r = \mathcal{S} :: (\ Buf_{a1} = \{e \triangleright elt\} \wedge e \triangleright elt \in Buf \triangleleft false \triangleleft false \triangleleft false \triangleleft Buf_{a2} = \emptyset \wedge Buf = \overset{\square}{Buf} - e \triangleright elt \triangleleft \{\ Buf_{a2} = \emptyset \prec (2 \triangleleft e \triangleright elt)\})$,
$S_r = \mathcal{S} :: (\ Count > 0 \triangleleft false \triangleleft false \triangleleft false \triangleleft Count = \overset{\square}{Count} - 1 \triangleleft \emptyset)$.
Subscribing *decr* to events with identifiers equal to 2 (subscription query $\square x : x \bowtie id = 2$) and applying the composition rule (strengthening the pre-condition of *decr* first) yields the following specification:
$S_r = \mathcal{S} :: (\ Buf_{a2} = \{e \triangleright elt\} \wedge e \triangleright elt \in Buf \wedge Count > 0 \triangleleft false \triangleleft false \triangleleft false \triangleleft Buf_{a2} = \emptyset \wedge Buf = \overset{\square}{Buf} - e \triangleright elt \wedge Count = \overset{\square}{Count} - 1 \triangleleft \{\ Buf_{a2} = \emptyset \prec (2 \triangleleft e \triangleright elt)\})$.

We now specify the kind of interactions we expect from the environment: 1) the environment may not publish events such that the identifier is equal to 1 or 2. This would lead to invalid incrementation/decrementation of the counter. 2) the environment may announce any other event. Various techniques can be applied for implementing these constraints on the environment. An example of such techniques is access control.

At this stage, the system is almost useless: the programs *add* and *remove* are not accessible to the environment. We subscribe them to the events with identifiers equal to 3 ($\square x : x \bowtie id = 3$) and to events with identifiers equal to 4 respectively. The environment can thus remove elements from or add elements to the buffer as it wills. However, since the programs *add* and *remove* do not allow interference, any event that is announced while one of these programs is running will simply be ignored.

Various properties of the system can be proved based on this specification. For instance, one can verify that each computation of the system conserves the property $\#\ Buf = Count$. This is, if the number of elements in the buffer is equals to the value of the counter before a computation, this will also be the case when the computation is completed. After proving some desirable properties of the system, the different components can be implemented such that they are correct w.r.t. their specifications.

6.2 Example 2

In the previous example, if an event arrives while a program is running, the event will simply be discarded. We extend the previous example such that if an event arrives while a program is running, the triggered program has to wait until it is save to run.

We consider the same system $\mathcal{S}_0$ as in the previous example. We enrich the global state with a boolean variable called *sentinel*. We want *add* to be of the form: `await` *sentinel* `do` *addbody*. A specification of *add* is $S_a = (P_1 \triangleleft R_1 \triangleleft P_1^{R_1} \wedge sentinel \triangleleft G_1 \triangleleft R_1^* | E_1 | R_1^* \triangleleft A_1)$ with $P_1 \overset{def}{=} Buf_{a1} = \emptyset$, $R_1 \overset{def}{=} Count = \overset{\square}{Count}$, $G_1 \overset{def}{=} Buf = \overset{\square}{Buf} + e \triangleright elt$, $E_1 \overset{def}{=} G_1$, and $A_1 = \{\ Buf_{a1} = \{e \triangleright elt\} \prec (1 \triangleleft e \triangleright elt)\}$. The specification of the body related to S_a can be deduced using the await-rule.

On the other hand we customize the specification of *incr* to meet the requirements of the composition rule. We now have $S_i \overset{def}{=} (P_2 \triangleleft R_2 \triangleleft W_2 \triangleleft G_2 \triangleleft E_2 \triangleleft \emptyset)$ where $P_2 \overset{def}{=} true$, $R_2 \overset{def}{=} G_1$, $W_2 = false$, $G_2 \overset{def}{=} R_1$, $E_2 \overset{def}{=} Count = \overset{\square}{Count} + 1$. It can easily be verified that starting from $\mathcal{S}_0$, the premises of the composition rule are satisfied. We now subscribe *incr* to events with identifiers equal to 1. The composition

rule yields the following result: $S_a = (P_1 \frown R_1 \frown P_1^{R_1} \wedge sentinel \frown G_1 \vee G_2 \frown E_1 \wedge E_2 \frown \{\ Buf_{a1} = \{e \triangleright elt\} \prec (1 \frown e \triangleright elt)\})$.
This is indeed the specification of a program which adds an element in the buffer and increments the counter. The program blocks until *sentinel* becomes true. Further, executing programs will not be interrupted by the environment. Other programs will wait until *sentinel* becomes true and no other program is running. This means that announced events are not simply discarded as in the previous example.

7 Conclusion

"Concurrent programming is hard and shared variable programming is very hard [30]." Concurrent programming with synchronization, shared variable and event based communication is therefore "three times" harder. We presented some shortcomings of the existing approaches in this paper.

In addition, we proposed a logic (LECAP) that allows specifying applications based on these paradigms (concurrency, shared variables, events, synchronization). The logic also supports composition of these specifications into specifications of larger applications. LECAP is therefore intrinsically oriented towards construction of complex systems. The paper also gave the formal semantics of the LECAP programming language. Such a language is based on an event based system whose formal definition was presented. Jones's rely/guarantee approach for the construction of interfering programs was extended with announcement conditions. The paper finally presented two examples that illustrate the approach.

It is obvious that a software development method cannot be established based on two examples. We therefore need to develop more examples and case studies to further experiment our approach. Additionally, refinement and verification of LECAP specifications are tasks we have to tackle.

Acknowledgments. We would like to acknowledge the helpful comments of Clemens Kerer, Gerald Reif, and Joe Oberleitner. We have also gratefully appreciated the useful comments received from Ketil Stølen.

References

1. P. Aczel. An inference rule for parallel composition, University of Manchester, 1983.
2. Stern Agerholm and Peter Gorm Larsen. A Lightweight Approach to Formal Methods. In *Proceedings of the Interna-tional Workshop on Currents Trends in Applied Formal Methods*. Springer Verlag, October 1998.
3. Apple Computer. *Inside Macintosh*, volume 6. Addison Wesley, 1991.
4. K.P. Birman. The progress group approach to reliable distributed computing. *Communications of the ACM*, 12:37–53, December 1993.
5. K. Brockschmidt. *Inside OLE*. Microsoft Press, Redmond, 1995.
6. C.A.R Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

7. A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 3(19):332–383, August 2001.
8. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.
9. G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI Event-Based Infrastructure and its Application to the Development of the OPSS WFMS. *Transaction of Software Engineering (TSE)*, 27(9), September 2001.
10. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
11. J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasonning about Implicit Invocation. In *Proceedings of the 6th International Symposium on the Foundations of Software Engineering, FSE-6*. ACM, 1998.
12. J. Dingel, D. Garlan, S. Jha, and D. Notkin. Towards a formal treatment of implicit invocation using rely/guarantee reasoning. *Formal Aspects of Computing*, 10, 1998.
13. C. Ene and Traian Muntean. A broadcast-based calculus for communicating systems. Technical report, Laboratoire d'Informatique de Marseille, 2000.
14. Object Management Group. OMG Formal Documentation. Technical report, OMG, December 1999.
15. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.
16. D. Jackson and J. Wing. *Lightweight Formal Methods*. In IEEE Computer. Springer Verlag, April 1996.
17. C.B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and Systems*, 5(4), October 1983.
18. Cliff B. Jones. *Systematic software development using VDM*. Prentice-Hall International, 1990. 2nd edition.
19. B. Krishnamurthy and N.S. Barghouti. Provence: A process visualization and enactment environment. In *Proceedings of 4th European Software Engineering Conference*, pages 451–465, 1993.
20. R. Milner. *The Calculus of Communicating Systems*. Prentice Hall, 1993.
21. Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Client/Server Survival Guide*. Wiley Computer Publishing, second edition, 1996.
22. S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5), May 1976.
23. K. Prasad. A calculus of broadcasting systems. In *Proceedings of TAPSOFT'91*, volume 493, 1991.
24. J.M. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.
25. S.P. Reiss. Connecting tools using message passing in the field program development environment. *IEEE Software*, 19(5), July 1990.
26. Ed Roman and Scott W. Ambler and Tyler Jewell. *Mastering Enterprise JavaBeans*. John Wiley & Sons, second edition, 2002.
27. K. Stolen. A Method for the Development of Totally Correct Shared-State Parallel Programs. In *CONCUR'91*, pages 510–525. Springer Verlag, 1991.
28. SunSoft. *The ToolTalk Service: An Inter-operability Solution*. Prentice-Hall, 1993.
29. Peter Sutton, Rhys Arkins, and Bill Segall. Supporting disconnectedness-transparent information delivery for mobile and invisible computing. In *Proceedings of 2001 IEEE International symposium on Cluster Computing and the Grid (CCGrid'01)*, May 2001.

30. Jim Woodcock and Arthur Hughes. Unifying theories of parallel programming. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM 2002)*. Springer Verlag, 2002.
31. Q. Xu, W.-P. de Roever, and J. He. The rely guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9:149–174, 1997.
32. Q. Xu and J. He. A theory of state-based parallel programming by refinement: part 1. In *Proceedings of the 4th BCS-FACS Refinement Workshop*. Springer Verlag, 1991.