Towards a formal treatment of implicit invocation

J. Dingel D. Garlan S. Jha D. Notkin

July 29, 1997
CMU-CS-97-153

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Implicit invocation [SN92,GN91] has become an important architectural style for large-scale system design and evolution. This paper addresses the lack of specification and verification formalisms for such systems. A formal computational model for implicit invocation is presented. We develop a verification framework for implicit invocation that is based on Jones' rely/guarantee reasoning for concurrent systems [Jon83,Stø91]. The application of the framework is illustrated with several examples. The merits and limitations of the rely/guarantee paradigm in the context of implicit invocation systems are also discussed.

Effort sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0299, and the the National Science Foundation under Grant No. CCR-9633532. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.
The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, the National Science Foundation or the U.S. Government.

Keywords: Implicit invocation, rely-guarantee, assumption-commitment

1 Introduction

A critical issue for large-scale systems design and evolution is the choice of an architectural style that permits the integration of separately-developed components into larger systems. Familiar styles include those based on remote procedure call [BN84], shared variables, asynchronous message passing, etc.

One key factor determining the effectiveness of an architectural style is the ability to reason effectively about properties of a system from properties of its components. As a result, considerable effort has gone into techniques for composition based on procedure invocation [Dij76,Hoa69], shared data [CM88,OG76], and message passing [Hoa85,Mil80,ISO87]. Even though practitioners rarely carry out formal reasoning throughout the full design and implementation process, they can both use the techniques as needed and also apply intuition that has been built up during development of the supporting techniques.

One increasingly important architectural style for system composition is implicit invocation (II) [SN92,GN91][1]. At its heart, II is based on the idea that a component A can invoke another component B without A being required to know B's name. Components such as B "register" interest in particular "events" that components such as A "announce." When A announces such an event, the II mechanism is responsible for invoking component B, even though A doesn't know that B or any other components are registered.[2]

There are a number of benefits of using the II architectural style, and it has been used in diverse settings such as programming environments and operating systems and others. Mechanisms to support II are found in commercial toolkits (e.g., Softbench [Ger89], ToolTalk, DecFuse), communication standards (e.g., Corba), integration frameworks (e.g., OLE), and programming environments (e.g., Smalltalk).

However, there is currently no established basis for reasoning about II systems. In particular it is difficult to answer questions like: What will be the effect of announcing a given event? Have enough event bindings been declared to achieve desired system behaviour? Does a given component announce sufficient events to permit effective integration? If a new component is added to an existing system, will it break the existing system? Are there the right components to produce desired overall system behaviour?

In this paper we describe one approach to providing such a basis for reasoning about systems designed using the II architectural style. The basic ideas are based on extending Jones' rely/guarantee approach to events. Specifically, we augment the assertion language to allow us to express the conditions under which a component will announce events. The overall system behaviour can then be reasoned about by establishing invariants over the effects achieved by individual components together with the state of pending events (i.e., those waiting to implicitly-invoke other computations). In order to reason with these invariants

[1] In other contexts "implicit invocation" is referred to by other names, such as "publish-subscribe" and "event multicast".

[2] In this paper, as we will see, a "component" is just a procedure or method.

we are also led to impose several constraints on the form of system computations to guarantee the atomicity of certain state changes. As we will discuss, the need for these additional constraints illustrates some of the limitations of an approach based on rely/guarantee, and suggests future extensions of the techniques described in the paper.

1.1 II Systems: utility and challenges

As sketched above, the central notion underlying II systems is that the "invokes" relation is decoupled from the "names" (or "knows-about") relation. That is, a component A can invoke a component B without knowing B's name. One of the simplest examples of II is when an operating system allows user code to register a callback procedure. For example, user code might register a procedure that is invoked when a particular signal is raised by the kernel. This allows the user code added control without compromising the kernel.

A somewhat more complicated example arises in broadcast message-based programming environments (such as those derived from Reiss' Field [Rei90] system). A collection of tools, such as a compiler, a debugger, an editor, a program visualization tool, etc., execute together. Rather than calling one another directly, at appropriate times they each announce potentially interesting activities. For example, the editor might announce, "procedure f was saved", while the debugger might announce, "the breakpoint in file x.c at line 173 was reached." Other tools might decide to listen for particular kinds of announcements. For example, the editor might listen for "breakpoint" announcements, so that it can move the cursor to the appropriate file and line. A centralized message server is used to deliver announcements to the tools that have registered interest.

By having tools announce potentially interesting events, and by having tools register interest, the conventional link between "invokes" and "names" is broken. In the example above, for instance, the debugger "invokes" the editor by announcing a breakpoint event, but the debugger is unaware of this. Indeed, some editors might not listen for this event, or multiple tools (even multiple editors) might listen for it. So, not only is implicit invocation used, but the invocation relation becomes one-to-many as opposed to the conventional one-to-one in conventional direct procedure invocation approaches.[3]

The conventional approach to reasoning about software systems depends on the link between invokes and names. Specifically, it is hierarchical and thus will not apply directly to II systems. In the hierarchical approach there are a set of primitives—often language constructs—that are associated with specific semantics (weakest preconditions, for example). Then one defines pre- and post-conditions for procedures and uses standard compositional techniques over the primitives to demonstrate that the axiomatic conditions hold. These conditions

[3] Logically, there is no reason that conventional procedure invocation need be one-to-one. But it happens at most rarely, and the one-to-many is a natural extension of implicit invocation. Note, however, that the operating system callback case is a situation in which it is implicit but also one-to-one.

are in turn used as primitives to prove properties about the enclosing procedures. And so on, until one can prove a property (often correctness) at the top-level of the program.

If one changes one of the primitives or procedures, a bounded amount of reasoning needs to be reapplied: basically, proofs from that point to the root of the tree need to be redone.

At the heart of these hierarchical reasoning approaches is the notion that the invocation relation is known statically. This is what allows reasoning about a procedure to be done in terms of the primitives and preconditions of procedures in which the given procedure is written. This static invocation relationship is not the fundamental composition structure used in II, so this reasoning approach is not necessarily appropriate for II systems.

To see why, consider an approach that attempts to reduce reasoning about II systems to standard hierarchical reasoning using pre- and post-conditions. In the case of a sequential II system (one in which each event-triggered procedure is executed to completion), one would be tempted to substitute:

$$\textbf{announce}(e)$$

with the corresponding procedure calls of the procedures bound to e. One can then apply standard pre-post reasoning techniques to the system.

However there are two fundamental problems with this. First, it violates the intended goal of decoupling the reasoning about a given component from the system in which its events are bound to other components. This is because changing any binding requires reanalysis of the components that announce the events in the changed bindings. Second, the technique is not tractable. Since the procedures bound to an event can be invoked in any order, it is necessary to consider all n! sequences of procedure invocations where n is the number of procedures.

In fact, the loosely coupled nature of the components in II systems cause them to be formally much more like a concurrent system than a sequential one (even when there is a single thread of control). Since the procedures associated with an event can be invoked in any order by the underlying II mechanism, there is inherent non-determinism in II systems, similar to that of concurrently executing processes. This suggests that it should be possible to apply techniques for reasoning about concurrent systems to II systems. In particular, it should be possible to enhance the interface specifications of II components so that they make explicit the role that they play in a system and environmental conditions under which they expect to function.

Thus, the central challenge in reasoning about II is to find ways to specify component interfaces and together with tractable composition mechanisms for reasoning about aggregate behaviour. This theory would allow us to determine:

- Does a given component satisfy its interface?
- Is a given composition well-formed (complete and consistent)?
- Is the aggregate behaviour of a system as desired?

1.2 Related Work

There are two general areas of related work. The first is research on implicit invocation systems. Most of the work on such systems has centered around developing practical mechanisms for exploiting the paradigm in real systems, such as programming environments like Field and Softbench [Rei90,Ger89]. Our work is inspired by the practical success of this work, and hopes to make engineering efforts based on it more effective by providing more principled basis for reasoning about II systems.

Within the general area of II research several researchers have attempted to provide precise characterizations of implicit invocation systems. An early survey of applications of the technique appeared in [GKN88] in which the authors illustrated how and why the ideas of II systems are pervasive in software systems. More recently [BCTW96] produced a taxonomic survey of II mechanisms, together with a generic object model for comparison of them. While this line of research has led to improved understanding of the design space for II-based systems, unlike our work, it does not attempt to provide a formal basis for reasoning about them.

Closer to our line of research, several researchers have attempted to provide a formal characterization of certain aspects of II systems. Two of this paper's authors produced an early characterization of II systems in Z [GN91]. More recently, researchers in software architecture have looked at some of the formal properties of II architectural styles [AAG95]. This research was primarily focused on taxonomic issues, and does not provide an explicit computational model that permits compositional reasoning about the behaviour of such systems.

Other researchers have looked at formal issues of event-multicast and process groups as a mechanism for achieving fault tolerance through replication [BJ89]. This work differs from that on implicit invocation in that multiple recipients of an event typically perform the *same* computations. This leads to very different requirements for underlying theory, since the main issue is how to add and remove replicated servers correctly to a running system.

The second closely related area of research is the area is formal models of concurrency. As we have said, this paper draws heavily on that work, and especially that of Jones and Stølen [Jon83,Stø91]. In our work we attempt where possible to apply existing research to this new domain, and to understand the strengths and limitations of established techniques.

In the remainder of this paper we describe a formalization of implicit invocation systems that is a first step towards this goal. The next section introduces a formal model for II systems. Section 3 describes the specification language. Section 4 demonstrates how II systems can be verified using rely/guarantee reasoning. Section 5 concludes and outlines further work.

2 A formal model of implicit invocation

We describe a computational model for II systems. A syntax and an operational semantics are given. Two concepts are crucial to the model: *methods* and *events*.

Methods A method m is a piece of (imperative) code, denoted by $code(m)$ or just c, also called program, that uses local and global variables. We assume there exists a set V of global variables that can be read and written by the entire system. Each method has its own set of local variables. The local variables $local(m)$ of a method m can only be read or written by the code of m and changes to them are not visible to the outside. The bindings of local variables $local(m) = \{x_1, \ldots, x_n\}$ are recorded in the code c itself and supersede the bindings of global variables with the same name. To this end, c is required to be of the form

$$c \ ::= \ \textbf{local}\ [x_1 = v_1, \ldots, x_n = v_n]\ \textbf{in}\ C$$

for some $n \geq 0$ where the values of the local variables are given by the declaration list $[x_1 = v_1, \ldots, x_n = v_n]$. We assume that all of the x_1 through x_n are distinct.

C is a program of a simple imperative language augmented with primitives for announcing and consuming events and an atomic section construct:

$$\begin{aligned} C \ ::= \ & x := expr \mid C_1; C_2 \mid \\ & \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2 \mid \textbf{while}\ B\ \textbf{do}\ C \mid \\ & \textbf{announce}(e) \mid \textbf{consume}(e) \mid \langle C \rangle \end{aligned}$$

The formal semantics of these statements will be given in the next section. The structure of a method m is illustrated in Figure 1.

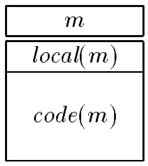

Fig. 1. Structure of a method m

Events The main purpose of an event is to trigger other methods. Typically, the event thus communicates a certain state change that the rest of the system needs to know about. In other words, an event is announced if and only if a certain state predicate is met. *Events are thus a carrier of semantics.* The state predicate whose truth is communicated through an event e is called the semantics of e, written $sem(e)$.

An event-method-binding EM, or *binding* for short, associates each event e with a set of methods that are to be triggered when that event is announced. Formally, EM is a possibly empty set of event-method pairs (e, m). Note that an event need not be bound to any methods and that several methods can be bound to the same event. Let $EM(e)$ denote the set of methods that e is bound to in EM, that is, $EM(e) = \{m \mid (e, m) \in EM\}$. An event e is considered to be *external* with respect to a set of methods M, if none of the methods in M issue e. (Note, however, methods still can be bound to external events.) Events that are not external are called *internal*.

Definition 1. A system $\mathcal{S} = (M, V, EM, Ex)$ is a collection of methods M together with a set of global variables V, a binding EM and a set of events Ex that is external to M. □

2.1 Operational semantics

The essential operational behaviour of an II system is that when methods execute they may announce events. When an event is announced the set of event-method pairs (as determined by EM) is added to an "active event" data structure ae. Concurrent with method executions, event-method pairs are removed from ae, causing the invocation of the associated method. Let l be a list of (e, m) pairs. We assume that ae supports two update operations $store(ae, l)$ and $remove(ae, (e, m))$, two predicates $empty(ae)$ and $(e, m) \in ae$, and an operation $\#(ae, e)$ that counts the number of occurrences of e in ae, that is, $\#(ae, e) = |\{(e, m) \mid (e, m) \in ae\}|$. In this model, we leave unspecified (i.e., non-deterministic) how precisely the events are stored and retrieved. In other words, our model does not contain any build-in assumptions about, for example, the policy that decides which event-method pair will be selected from the active event data structure or how duplicate occurrences of events or concurrent updates should be handled. In practice, systems institute specific policies to achieve certain kinds of ordering relationships. In Section 3.1 we will see an example where it is necessary to pick a particular dispatch policy. However, this paper does not attempt to classify which policies are needed by which applications.

To achieve compositionality the semantics of a collection of methods will be given subject to the behaviour of the environment the methods are executing in. The semantics defines transitions between configurations. We first introduce the components of a configuration. Methods can either be waiting for events or executing. To distinguish between these states each method m_i is associated with a boolean flag a_i. If $a_i = true$, then c_i, the code of method m_i, is currently being executed, and we say that method m_i is *active*. $a_i = false$ indicates that code c_i is currently not being executed. In this case, method m_i is called *idle*.

A *state* s, is a mapping from global variables to values, $s : V \to Val$. Whenever $(e, m) \in ae$ then event e is currently *active* and still needs to be delivered to method m.

Definition 2. Let c_i be programs, a_i be boolean flags, s a state, and ae the active events data structure. A *configuration* is a 3-tuple

$$\langle \langle (c_i, a_i) \rangle_{i=1}^{n}, s, ae \rangle$$

where $\langle (c_i, a_i) \rangle_{i=1}^{n} = \langle (c_1, a_1), \ldots, (c_n, , a_n) \rangle$. If the precise number of methods is irrelevant, we will abbreviate this by $\langle (c_i, a_i) \rangle_i$.

A *transition* is of the form $\langle\langle(c_i, s_i, a_i)\rangle_i, s, ae\rangle \xrightarrow{l} \langle\langle(c'_i, s'_i, a'_i)\rangle_i, s', ae'\rangle$ where the label l is one of $\{env, pro\}$. If $l = pro$ then we have a *program transition. Environment transitions* have $l = env$. Intuitively, environment transitions model transitions made by other methods in the system. $\square$

Method semantics Before the operational semantics of the overall system can be defined, we need to give a semantics for the code of a method. This semantics is a family of local transition relations $\rightsquigarrow_{(EM,m)}$ that is parameterized with the current binding *EM*, and the method m which is currently executing. These local transition relations link local configurations of the form $\prec c, s, ae \succ$. A local transition

$$\prec c, s, ae \succ \rightsquigarrow_{(EM,m)} \prec c', s', ae' \succ$$

means that code c transformed state s and the active events data structure ae to s' and ae' respectively assuming that c is executed under the binding *EM*, and that c is the code of method m. The remainder of the code is c'. A local transition

$$\prec c, s, ae \succ \rightsquigarrow_{(EM,m)} \prec s', ae' \succ$$

additionally expresses that c terminates in one step.

The imperative constructs have the standard semantics. Assignments, for example, are defined as follows:

$$\frac{}{\prec x := e, s, ae \succ \rightsquigarrow_{(EM,m)} \prec [s|x = n], ae \succ} \quad \text{if } e \text{ evaluates to } n \text{ in } s$$

The atomic section construct hides intermediate states:

$$\frac{\prec C, s, ae \succ \rightsquigarrow^*_{(EM,m)} \prec s', ae' \succ}{\prec \langle C \rangle, s, ae \succ \rightsquigarrow_{(EM,m)} \prec s', ae' \succ}$$

where $\rightsquigarrow^*_{(EM,m)}$ denotes the reflexive and transitive closure of $\rightsquigarrow_{(EM,m)}$.

The event primitives **announce** and **consume** behave as follows:

$$\frac{}{\prec \mathbf{announce}(e), s, ae \succ \rightsquigarrow_{(EM,m)} \prec s, store(ae, [(e, m_1), \ldots, (e, m_n)]) \succ}$$

where $EM(e) = \{m_1, \ldots, m_n\}$. That is, **announce**$(e)$ causes (e, m') to be announced only if e is bound to m' in the current binding *EM*. Note that if an announced event has no methods bound to it by *EM*, no pairs are added to the active event data structure — that is, ae is unchanged.

Once a **consume**(e) is executed by method m, the event e is considered delivered to m and the pair (e, m) is removed from the active events data structure.

$$\frac{}{\prec \mathbf{consume}(e), s, ae \succ \rightsquigarrow_{(EM,m)} \prec s, remove(ae, (e, m)) \succ}$$

We want a local variable declaration to hide the changes of the declared variables. We adopt the standard operational treatment of local variables. If, from a state in which the local variables carry their local values, C has a transition to $\prec C', s', ae' \succ$, then $\prec$**local** lb **in** $C, s, ae \succ$ has a transition that leaves the values of the local variables unchanged and stores the new values of the local variables in the updated declaration list dl'.

$$\frac{\prec C, [s|x_1 = v_1|\ldots|x_n = v_n], ae \succ \ \rightsquigarrow_{(EM,m)} \ \langle C', s', ae' \rangle}{\prec \mathbf{local}\ dl\ \mathbf{in}\ C, s, ae \succ \ \rightsquigarrow_{(EM,m)} \ \prec \mathbf{local}\ dl'\ \mathbf{in}\ C', s'', ae' \succ}$$

where $dl = [x_1 = v_1, \ldots, x_n = v_n]$ and $dl' = [x_1 = s'(x_1), \ldots, x_n = s'(x_n)]$ and $s'' = [s'|x_1 = s(x_1)|\ldots|x_n = s(x_n)]$. Termination of the body of a declaration induces termination of the declaration.

$$\frac{\prec C, [s|x_1 = v_1|\ldots|x_n = v_n], ae \succ \ \rightsquigarrow_{(EM,m)} \ \prec s', ae' \succ}{\prec \mathbf{local}\ dl\ \mathbf{in}\ C, s, ae \succ \ \rightsquigarrow_{(EM,m)} \prec s'', ae' \succ}$$

where $dl = [x_1 = v_1, \ldots, x_n = v_n]$ and $s'' = [s'|x_1 = s(x_1)|\ldots|x_n = s(x_n)]$.

System semantics We are now ready to define the global transition relation that describes the behaviour of the entire system.

Definition 3. For each binding EM the transition relation $\longrightarrow_{EM}$ is the smallest relation satisfying

- environment transitions:

$$\langle\langle(c_i, a_i)\rangle_i, s, ae\rangle \xrightarrow{env}_{EM} \langle\langle(c_i, a_i)\rangle_i, s', ae'\rangle$$

  for all c_i, a_i, s, ae, s', ae' and
- program transitions:

$$\langle\langle(c_1, a_1), \ldots, (c_i, a_i), \ldots, (c_n, a_n)\rangle, s, ae\rangle \xrightarrow{pro}_{EM}$$
$$\langle\langle(c_1, a_1), \ldots, (c_i', a_i'), \ldots, (c_n, a_n)\rangle, s', ae'\rangle$$

  whenever
  1. $a_i = true$ and $\prec c_i, s, ae \succ \ \rightsquigarrow_{(EM,m_i)} \ \prec c_i', s', ae' \succ$ and $a_i' = a_i$, or
  2. $a_i = true$ and $\prec c_i, s, ae \succ \ \rightsquigarrow_{(EM,m_i)} \ \prec s', ae' \succ$ and $c_i' = code(m_i)$ and $a_i' = false$, or
  3. $(e, m_i) \in ae$ and $a_i = false$ and $a_i' = true$ and $c' = c$ and $ae' = ae$. □

The intuition behind the above definition is the following: The environment has access to the global state and the active events data structure and can change these arbitrarily in an environment transition. A program transition can arise in three different situations:

1. If a method is active and its code is not yet terminated, then it continues to be active and execute its code.

2. If a method is active and its code terminates, then it is set to idle, and the code is restored.
3. If event e is active and bound to method m_i that is not currently active, then m_i can be activated.

Note that an event cannot trigger a method that is already active. In other words, at most one "incarnation" of each method is active at any time. Once a method has been activated, its code will be fully executed before it gets deactivated. Also note that this formulation can readily be extended to handle, for instance, changes to the *EM* binding at runtime, or the use of more specific method activation strategies.

When reasoning about an II system it is typically the case that one wants to assert the establishment of some predicate once the system has reached a quiescent state. To facilitate that we identify a *disabled configuration* as one that can make no transitions.

Definition 4. A configuration $\langle\langle(c_i, s_i, a_i)\rangle_i, s, ae\rangle$ is *disabled* under *EM* if there are no c'_i, s'_i, a'_i, s' and ae' such that

$$\langle\langle(c_i, s_i, a_i)\rangle_i, s, ae\rangle \xrightarrow{pro}_{EM} \langle\langle(c'_i, s'_i, a'_i)\rangle_i, s', ae'\rangle.$$

Definition 5. A *computation* under some binding *EM* is a possibly infinite sequence of program and environment transitions

$$\langle\langle(c_{1_i}, s_{1_i}, a_{1_i})\rangle_i, s_1, ae_1\rangle \xrightarrow{l_1}_{EM} \ldots \xrightarrow{l_{j-1}} \langle\langle(c_{j_i}, s_{j_i}, a_{j_i})\rangle_i, s_j, ae_j\rangle \xrightarrow{l_j}_{EM} \ldots$$

such that the final configuration is disabled under *EM* if the sequence is finite. A finite computation is also said to be *terminating*. □

Given a computation σ, then $C(\sigma)$, $S(\sigma)$, $AE(\sigma)$ and $L(\sigma)$ are the obvious projection functions to sequences of programs, states, active events and transition labels. $\sigma[i]$, $C(\sigma, i)$, $A(\sigma, i)$, $S(\sigma, i)$, $AE(\sigma, i)$ and $L(\sigma, i)$ denote, respectively, the ith configuration $\langle\langle(c_{j_i}, s_{j_i}, a_{j_i})\rangle_j, s, ae\rangle$, the ith vector of programs $\langle c_i\rangle_i$, the ith vector of flags $\langle a_i\rangle_i$, the ith state s_i, the ith active events data structure ae_i, and the ith label l_i of σ. Let $S{\times}AE$ be the product of the two projection functions S and AE, that is, $S{\times}AE(\sigma, i) = (S(\sigma, i), AE(\sigma, i))$.

Given a system $\mathcal{S}$, σ is a computation of $\mathcal{S}$ if it starts out with a set of inactive methods.

Definition 6. Given a system $\mathcal{S} = (M, V, EM, Ex)$ with $M = \{m_1, \ldots, m_n\}$, the set of all computations of $\mathcal{S}$, $comp(\mathcal{S})$, is given by all computations σ under *EM* with $C(\sigma, 1) = \langle code(m_i)\rangle_{i=1}^n$ and $A(\sigma, 1) = \langle \mathit{false}\rangle_{i=1}^n$. □

3 Specification language

Rely/guarantee reasoning [Jon83,Stø91] has successfully been applied to concurrent systems. We now show this approach can be extended to our computational model of II systems.

Predicates States are described by *state predicates*. As usual, these are formulas consisting of constants, variables, function and predicate symbols and the standard boolean connectives. Unprimed variables will be used to refer to an *earlier* system state. Note that this is not necessarily the previous state. Thus, for each variable x, there is a primed variable x'. Primed variables cannot appear in programs. Let A be a state predicate. We write $(s_1, s_2) \models A$ if A is true when each unprimed variable x in A is assigned the value $s_1(x)$ and each primed variable x' in A is assigned the value $s_2(x)$. A state predicate A can thus be interpreted as the set of pairs of states (s_1, s_2) such that $(s_1, s_2) \models A$. In this case, A is called a *binary* state predicate. If, however, A does not contain any primed variables, then A may also be thought of as the set of states s such that $s \models A$. A is called a *unary* state predicate in this case.

In certain situations we also want to express how the active events data structure will be changed in the course of a transition. To this end we introduce *event predicates*. The variable ae is reserved to denote the active events data structure. Given an event e, an event predicate is a boolean combination of the atomic predicates $active(e)$, e++ and e--. Let ae and ae' be two active events data structures. We say $active(e)$ is true in (ae, ae') if there is a method m such that $(e, m) \in ae'$, that is, $(ae, ae') \models active(e)$ iff $(e, m) \in ae'$ for some m. e++ expresses that e has just been announced. e-- says that e has just been consumed. e++ (or e--) is true in (ae, ae') if the number of occurrences of e in ae' is one greater (or smaller) than the number of occurrences of e in ae. Formally, $(ae, ae') \models e$++ iff $\#(ae, e) = \#(ae', e) + 1$ and $(ae, ae') \models e$-- iff $\#(ae, e) = \#(ae', e) - 1$. A *state-event predicate* is the boolean combination of state and event predicates and is thus interpreted over 4-tuples $((s, ae), (s', ae'))$ in the obvious fashion.

Specifications A specification is of the form $\varphi = (P, R, G, Q)$, where the *pre-condition* P is a unary event-state predicate, and the *rely-condition* R, the *guarantee-condition* G, the *input/output-condition* Q are binary event-state predicates.

Let $len(\sigma)$ be the number of configurations in σ. Given a set of variables X and two states s_1, s_2, then $s_1 =_X s_2$ denotes that for all variables $x \in X$, $s_1(x) = s_2(x)$ while $s_1 \neq_X s_2$ denotes that there exists a variable $x \in X$, such that $s_1(x) \neq s_2(x)$.

Definition 7. Let V be the set of global program variables. Given a binding EM, a pre-condition P, a rely-condition R, then $env(V, P, R)$ denotes the set of all computations σ under EM, such that

- $S{\times}AE(\sigma, 1) \models P$,
- for all $1 \leq i < len(\sigma)$, whenever $L(\sigma, i) = env$ and $S(\sigma, i) \neq_V S(\sigma, i+1)$, then $(S{\times}AE(\sigma, i), S{\times}AE(\sigma, i+1)) \models R$. That is, all environment transitions that change the value of at least one variable satisfy the rely R. □

Definition 8. Let V be the set of global program variables. Given a binding EM, a guarantee-condition G, a input/output-condition Q, then $prog(V, G, Q)$ denotes the set of all computations σ under EM, such that

- σ is finite,
- for all $1 \leq i < len(\sigma)$, whenever $L(\sigma, i) = pro$ and $S(\sigma, i) \neq_V S(\sigma, i+1)$, then $(S{\times}AE(\sigma, i), S{\times}AE(\sigma, i{+}1)) \models G$. That is, all program transitions that change the value of at least one variable satisfy the guarantee G.
- $(S{\times}AE(\sigma, 1), S{\times}AE(\sigma, len(\sigma))) \models Q$. □

Judgements A judgement is a pair consisting of a system $\mathcal{S} = (M, V, EM, Ex)$, and a specification $\varphi = (P, R, G, Q)$, written $\mathcal{S} \models \varphi$. A judgement is true, if all computations σ of M under EM are such that whenever σ terminates and satisfies the relies (on initial state and environment transitions), then it will also satisfy the guarantees (on the program transitions and the final state).

Definition 9. Let $\mathcal{S} = (M, V, EM, Ex)$ be a system. The judgement

$$\mathcal{S} \models (P, R, G, Q)$$

is true iff

$$comp(\mathcal{S}) \cap env(V, P, R) \subseteq prog(V, G, Q).$$

□

We now define executions. These are finite computations that start and end with an empty active events data structure and restrict top-level environment interference to the announcement of external events while the state is left unchanged.

Definition 10. Let $\mathcal{S} = (M, V, EM, Ex)$ be a system. The set of executions of $\mathcal{S}$, $exec(\mathcal{S})$, is given by

$$exec(\mathcal{S}) = comp(\mathcal{S}) \cap env(V, empty(ae), R_{Ex}) \cap prog(V, true, true)$$

where R_{Ex} is

$$\begin{array}{l}(\bigwedge_{x \in V} x' = x) \wedge \\ (ae' = store(ae, [(e_1, m_1), \ldots, (e_n, m_n)])) \wedge \\ (\forall 1 \leq i \leq n . e_i \in Ex \wedge (e_i, m_i) \in EM)\end{array}$$

and thus restricts the top-level environment to the announcement of external events. For state-event predicates P and Q, a partial correctness triple

$$\{P\}\ \mathcal{S}\ \{Q\}$$

is true iff every execution of $\mathcal{S}$ that starts in a state satisfying P terminates in a state such that Q holds. □

When considering executions the system is thus regarded not as a closed system but one that is still subject to interference by the top-level environment. However, this interference is limited to the announcement of external events.

3.1 Example: sets and counters

A common use of II systems is to provide loose coupling between parts of a system that are individually responsible for updating separate portions of the state. The *EM* binding is used to provide establish relationships between the different parts of the system state: specifically, when one part of the system changes its part of the state, events trigger corresponding updates to other parts of the state.

As a simple example, consider a system in which the state consists of a set and a counter. The set has methods to insert and delete elements. The counter has increment and decrement methods. The *EM* binding is then used to establish a system "invariant" that the value of the counter be the size of the set. Formally, consider a system $\mathcal{S}$ with methods

$$M = \{\mathit{insert}(x), \mathit{increment}, \mathit{delete}(x), \mathit{decrement}\},$$

global variables $V = \{C, S\}$, external events $Ex = \{\mathit{ins}(n), \mathit{del}(n) \mid n \in \mathbb{N}\}$, internal events $\{\mathit{incr}, \mathit{decr}\}$, and binding *EM* with

event	*method*
$\mathit{ins}(n)$	$\mathit{insert}(x)$
$\mathit{del}(n)$	$\mathit{delete}(x)$
incr	$\mathit{increment}$
decr	$\mathit{decrement}$

The idea is that an element n can be inserted into or deleted from the set S using the method $\mathit{insert}(x)$ or $\mathit{delete}(x)$. Analogously, the counter C can be incremented or decremented using *increment* or *decrement*. In this case *EM* provides the necessary binding between events announced by the methods that change the state of the set, so that the state of the counter can be updated. The methods have the following structure.

m :	$\mathit{insert}(x)$	$\mathit{delete}(x)$
$\mathit{local}(m)$:	$\emptyset$	$\emptyset$
$\mathit{code}(m)$:	**local** [] **in** **consume**$(\mathit{ins}(x))$; **if** $x \notin S$ **then** $\langle S := S \cup \{x\}$; **announce**$(\mathit{incr})\rangle$	**local** [] **in** **consume**$(\mathit{del}(x))$; **if** $x \in S$ **then** $\langle S := S \backslash \{x\}$; **announce**$(\mathit{decr})\rangle$

In the methods above x is used as a formal parameter that is instantiated upon method invocation.

m :	*increment*	*decrement*
$local(m)$:	$\emptyset$	$\emptyset$
$code(m)$:	**local** [] **in** $\langle C := C + 1;$ **consume**$(incr)\rangle$	**local** [] **in** $\langle C := C - 1;$ **consume**$(decr)\rangle$

Given the external event $ins(n)$, the formal parameter x of method $insert(x)$ is replaced by n and the method is invoked. Similarly for external events $del(n)$. If necessary, the set S is updated by inserting or deleting the element n and the corresponding event is announced. This in turn triggers either *increment* or *decrement*.

The above methods communicate by exchanging the events *incr* and *decr*. These events have the following semantics.

event e	$sem(e)$
incr	$\exists x.x \notin S \wedge S' = S \cup \{x\}$
decr	$\exists x.x \in S \wedge S' = S \backslash \{x\}$

Given a set of events E, the *characteristic formula* of E expresses that all events in E get announced if and only if their semantics is met. Formally, cf_E is $\bigwedge_{e \in E}(e\text{++} \leftrightarrow sem(e))$. Making cf_E part of the guarantee condition, thus allows us to show that a given method respects the semantics of its events.

When run in an initial state in which $x \notin S$, $insert(x)$ announces the event *incr* precisely when its semantics is met. Similarly for $delete(x)$ and initial states in which $x \in S$. If these preconditions are not met, both methods will not cause any state change (the next state relation is restricted to stuttering through the guarantee *false*). Unrestricted environment interference prevents us from being able to make any non-trivial assertions about the final state. Formally,

$$(insert(x), V, EM, Ex) \models (x \notin S, true, cf_{\{incr,decr\}}, true)$$

$$(insert(x), V, EM, Ex) \models (x \in S, true, false, true)$$

$$(delete(x), V, EM, Ex) \models (x \in S, true, cf_{\{incr,decr\}}, true)$$

$$(delete(x), V, EM, Ex) \models (x \notin S, true, false, true).$$

We assume that events are commutative, that is, the order in which they are announced is irrelevant. In this case an implementation of *ae* as a *multiset* would therefore be correct.

Suppose we wanted to extend our system by the external event *init*, the internal event *res*, the methods *initialize* and *reset* and the bindings

event	*method*
init	*initialize*
res	*reset*

where

m :	*initialize*	*reset*
$local(m)$:	$\emptyset$	$\emptyset$
$code(m)$:	**local** [] **in** **consume**(*init*); $\langle S := \emptyset$; **announce**(*res*)$\rangle$	**local** [] **in** $\langle C := 0$; **consume**(*res*)$\rangle$

The external event *init* causes method *initialize* to be invoked, which empties the set and announces the *res* event. This in turn triggers the *reset* method which sets the counter to 0. Note that in this extended system events are not commutative anymore. We need to keep track of the order in which events are announced and thus require a more refined computational model. More precisely, the active events data structure *ae* must thus be kept in a *queue* rather than a multiset.

4 Formal reasoning

Assume that we want to reason about the system $\mathcal{S} = (M, V, EM, Ex)$ and show that it satisfies some partial correctness triple $\{P_{\mathcal{S}}\}\ \mathcal{S}\ \{Q_{\mathcal{S}}\}$. This section shows how this can be accomplished.

1. We start with some local reasoning on the method level.
   (a) First, we choose appropriate predicates P, R, and Q describing the initial state, the relies on the top-level environment, and the final state respectively.
   (b) For each method $m \in M$ and the corresponding "rest of the system" $M\backslash\{m\}$ we identify guarantees G_m and $G_{M\backslash\{m\}}$ such that
      i. whenever m is executed from an initial state satisfying P and in an environment satisfying $R \vee G_{M\backslash\{m\}}$ and terminates, then m will change the state according to G_m and the final state will be such that Q holds. Formally,

      $$(m, V, EM, Ex) \models (P, R \vee G_{M\backslash\{m\}}, G_m, Q)$$

      for all $m \in M$, and
      ii. whenever $M\backslash\{m\}$, the rest of the system, is run from an initial state satisfying P and in an environment satisfying $R \vee G_m$ and terminates, then $M\backslash\{m\}$ will change the state according to $G_{M\backslash\{m\}}$ and the final state will be such that Q holds. Formally,

      $$(M\backslash\{m\}, V, EM, Ex) \models (P, R \vee G_m, G_{M\backslash\{m\}}, Q)$$

      for all $m \in M$.

Intuitively, the above shows that both the method m and the rest of the system $M \backslash \{m\}$ stick to their guarantees if the other one does.

(c) Now it is safe to conclude that whenever the entire system is executed in an initial state satisfying P and in an environment satisfying R and terminates, then it will change the state according to $\bigvee_{m \in M} G_m$ and the final state will be such that Q is met. That is,

$$(M, V, EM, Ex) \models (P, R, \bigvee_{m \in M} G_m, Q).$$

The soundness of this step is implied by the rely/guarantee reasoning method put forward by Jones and others [Jon83,Stø91].

2. Now we weaken the above judgement. By definition, every execution starts in a state with $empty(ae)$ and the interference allowed by the top-level environment is described by R_{Ex}. Moreover, we are only interested in initial states satisfying $P_{\mathcal{S}}$. Thus, we need to show $P_{\mathcal{S}} \wedge empty(ae) \Rightarrow P$ and $R_{Ex} \Rightarrow R$. In this case, we get

$$(M, V, EM, Ex) \models (P_{\mathcal{S}} \wedge ae = \emptyset, R_{Ex}, true, Q).$$

3. Due to the semantics of **announce**(e), ae cannot contain events that do not trigger anything. Thus, every disabled configuration must have $empty(ae)$. To obtain the desired partial correctness property, we therefore need to show $Q \wedge empty(ae) \Rightarrow Q_{\mathcal{S}}$. In this case, it is sound to conclude that the partial correctness property holds

$$\{P_{\mathcal{S}}\}\ \mathcal{S}\ \{Q_{\mathcal{S}}\}.$$

Following [Jon83,Stø91] a more general formulation step 1 would be possible. However, the present treatment is sufficient for our purposes.

4.1 Example: sets and counters

Let $\mathcal{S}$ be the system introduced in Section 3.1. By binding the *incr* and the *decr* events to *increment* and *decrement* respectively, we hope to have established a link between the size of the set S and the value of the counter C. More precisely, we want the triple

$$\{|S| = C\}\ \mathcal{S}\ \{|S| = C\}$$

to hold.

1. Let I_1 be given by

$$I_1 \;\equiv\; (|S| = C + \#incr - \#decr)$$

where $\#e$ abbreviates the number of occurrences of e in ae, that is, $\#(ae, e)$. To prove the partial correctness property above we adopt the outlined strategy in a somewhat degenerate but sufficient fashion. We show that I_1 is an invariant for each of the methods and thus also for the entire system.

More precisely, with respect to the above strategy we let $P = R = G_m = G_{M\backslash\{m\}} = Q = I_1$ for all $m \in M$. We can show that all methods preserve I_1.

$$(m, V, EM, Ex) \models (I_1, I_1, I_1, I_1)$$

for all $m \in \{insert(x), delete(x), increment, decrement\}$.

This part of the verification reveals an important point. I_1 expresses a relationship between the state variables and the active events data structure. For this invariant to be preserved by *every* transition, it is necessary that every method announces changes to the state variables that destroy that relationship by *simultaneously* announcing the corresponding event using the atomic section construct. If, for instance, a method first updates the state variables and then announces the event at some later stage, it is likely to be impossible to establish any non-trivial relationship between the state variables and the pending events for that method. We regard the need for an atomic region construct as a limitation of our framework that compromises practicality. Section 5 contains a more detailed discussion of this issue.

Next, it is easy to see that for each m the rest of the system $M\backslash\{m\}$ also preserves the invariant.

$$(M\backslash\{m\}, V, EM, Ex) \models (I_1, I_1, I_1, I_1)$$

for all $m \in \{insert(x), delete(x), increment, decrement\}$. Thus, I_1 is an invariant for all of $\mathcal{S}$.

$$\mathcal{S} \models (I_1, I_1, I_1, I_1).$$

2. We weaken the specification (I_1, I_1, I_1, I_1) to $(C = |S| \wedge empty(ae), R_{Ex}, true, I_1)$. Note that $C = |S| \wedge empty(ae) \Rightarrow I_1$ and $R_{Ex} \Rightarrow I_1$. Thus,

$$\mathcal{S} \models (C = |S| \wedge empty(ae), R_{Ex}, true, I_1).$$

3. We show

$$\{C = |S|\}\ \mathcal{S}\ \{C = |S|\}$$

by arguing that $I_1 \wedge empty(ae)$ implies $C = |S|$.

Note that the above reasoning could easily be extended to handle the example system augmented with the methods *initialize* and *reset* under the appropriate binding.

4.2 Example: a filesystem

We now consider an example inspired by the common application of implicit invocation to software development environments, such as Field [Rei90].

Previously, a state was a mapping from variables to values. We now consider a slightly different scenario, in which the state is given by the contents and the attributes of a file system $\mathcal{F}$. Suppose F is a set of source files. We assume that the files in F correspond to an executable file *target* and that **make**(F, *target*) creates a new executable with respect to the current contents of F.

In the following, f will range over files in $\mathcal{F}$. The system $\mathcal{F}$ contains the methods $M = \{edit(f), compile\}$, the internal event *modified*, the external events $update(f)$, and the binding EM with

event	*method*
$update(f)$	$edit(f)$
modified	*compile*

The semantics of the *modified* event is

$$sem(modified) \;\equiv\; \neg fresh(target, F)$$

where $fresh(f, F)$ denotes that the last modification date of f is more recent than that of all files in F, that is, for all $f' \in F$,

$$date\_last\_modified(f) \geq date\_last\_modified(f').$$

We assume that the methods are of the following form:

m :	$edit(f)$	*compile*
$local(m)$:	*buf*	$\emptyset$

$code(m)$:	**local** [$buf = \emptyset$] **in**
	  consume($update(f)$)
	  read(f, *buf*);
	  ...
	  $\langle$**if** $dirty(buf, f)$ **then**
	    save(buf, f);
	    if $f \in F$ **then**
	      announce(*modified*)$\rangle$
		local [] **in**
		  $\langle$**make**(F, *target*);
		  consume(*modified*)$\rangle$

An external $update(f)$ event causes the file f to be edited. The $edit(f)$ method copies the contents of f into a local buffer *buf* and if, at the end of the edit session, the buffer differs from the contents of f, then f is updated with *buf*. If f also is a source file relevant to *target* the *modified* event is announced. The *modified* event triggers the *compile* method which updates the executable. Note that the $update(f)$ and the *modified* event are not commutative, that is, the order in which events are announced does matter. Again, this means that *ae* must be kept as a *queue* rather than a multiset.

We would like to show that

$$\{fresh(target, F)\}\ \mathcal{F}\ \{fresh(target, F)\}.$$

To this end, we again first establish an invariant. However, in contrast to the previous example, we make use of the semantics of the *modified* event to prove the invariant. Let

$$\begin{aligned}
I_2 &\equiv \mathit{fresh}(\mathit{target}, F) \vee \mathit{sem}(\mathit{modified}) \\
I_2' &\equiv \mathit{fresh}(\mathit{target}, F) \vee \mathit{active}(\mathit{modified}) \vee \\
&\quad (F' = F \wedge \neg(\mathit{modified}\text{++}) \wedge \neg(\mathit{modified}\text{--})).
\end{aligned}$$

I_2 is a tautology and thus trivially an invariant. We can show that whenever the environment changes the state according to I_2', then $\mathit{edit}(f)$ will announce *modified* if and only if its semantics is met. Similarly for *compile*.

$$(\mathit{edit}(f), V, EM, Ex) \models (\mathit{true}, I_2', \mathit{cf}_{mod}, I_2')$$

$$(\mathit{compile}, V, EM, Ex) \models (\mathit{true}, I_2', \mathit{cf}_{mod}, I_2')$$

where $\mathit{cf}_{mod} \equiv \mathit{modified}\text{++} \leftrightarrow \mathit{sem}(\mathit{modified})$. Using the tautology I_2 it is easy to see that cf_{mod} implies I_2'. Consequently, the relies and guarantees fit together, and we can conclude

$$\mathcal{F} \models (\mathit{true}, I_2', \mathit{cf}_{mod}, I_2').$$

Since R_{Ex} implies I_2', this can be weakened to

$$\mathcal{F} \models (\mathit{empty}(ae) \wedge \mathit{fresh}(\mathit{target}, F), R_{Ex}, \mathit{true}, I_2')$$

which then implies the desired result

$$\{\mathit{fresh}(\mathit{target}, F)\}\ \mathcal{F}\ \{\mathit{fresh}(\mathit{target}, F)\}.$$

5 Conclusion and further work

We have presented a formal model of II. Using this model as a guideline, we developed a framework that supports formal reasoning about II systems. This framework was obtained as an extension of Jones' rely/guarantee reasoning, and thus naturally inherits many of its benefits and deficiencies like, for instance, the reconciliation of concurrency and compositionality and the lack of support for liveness properties. Several examples illustrated the use and applicability of the proposed framework. A potential abstraction mechanism is offered through the event semantics.

The problem of atomicity is inherent to concurrent systems with shared resources. It resurfaces in this work with the following interesting consequences. To allow for fine-grained parallelism we also chose a fine-grained operational semantics. On the specification level, however, we would like to be more abstract and not always be forced to reason about every transition. Unfortunately, the kind of rely/guarantee reasoning adopted here requires us to do exactly that: An assertion is only an invariant if it is preserved by *every* transition. As we have seen, invariants are crucial for the verification. To be able to prove non-trivial invariants, we thus had to enforce certain atomicity constraints by means of an atomic region construct.

This is undesirable for three reasons: First, it conflicts with our ideal of fine grained parallelism. Second, it compromises the practicality of the framework, since sometimes II systems are implemented without such a construct. Third, and most importantly, it seems to be, in some sense, an unnecessary restriction. Consider the set/counter example. Suppose we removed all critical region constructs. The invariant would obviously fail, whereas the partial correctness property would continue to hold. What is essential here is that every set update is eventually followed by the announcement of the appropriate event. The simultaneity in our framework enforced by the need for an invariant is just a special case of this. This reveals a fundamental mismatch between judgements that are true on the one hand and judgements that can be proven in our framework on the other hand.

Another artifact of our need for low level invariants is the explicit $\mathbf{consume}(e)$ statement. On the one hand, it allows us to pinpoint changes to the active events data structure to transitions that also update the state in a specific way. On the other hand, it compromises practicality and maintainability. II systems in general do not have an explicit $\mathbf{consume}(e)$ statement. Instead, system runtime mechanisms invoke the method bound to an event, automatically removing that event from active event set. Moreover, the explicit consumption of events introduces an unnecessary dependency between the event-method binding EM and the code of a method. In particular, changes to EM must be reflected by changes to the **consume** statements.

Further work The most important focus of further work will be the development of a verification framework that does not impose the restrictions mentioned above. Such a framework would allow, for example, the proof of the partial correctness triple of the counter example even when the $insert(x)$ method chooses to announce the $incr$ after the actual update of the set. The framework should also not depend on the explicit consumption of events. A formulation of rely/guarantee reasoning in which relies and guarantees can be given in terms of temporal logic formulas seems promising in this respect.

The event semantics plays only a peripheral role in this paper. However, we envision it as a powerful abstraction mechanism that forms the basis of a two stage process: First, it shown that events are announced precisely when they are supposed to. In other words, using local reasoning similar to the one described in this paper, we prove that all methods respect the event semantics. Second, this event semantics is then used to do global reasoning, that is, the behaviour of the overall system is reasoned about purely in terms of the events and their semantics. To structure the reasoning, it might then be helpful to organize the dependencies between events by means of a graph or even a Petri net.

Acknowledgement We thank Stephen Brookes for carefully reading drafts of this paper.

References

[AAG95] G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, October 1995.

[BCTW96] D.J. Barrett, L.A. Clarke, P.L. Tarr, and A.E. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, October 1996.

[BJ89] K. Birman and Th. Joseph. Exploiting replication in distributed systems. In Mullender and Sape, editors, *Distributed Systems*, pages 319 – 365. Addison Wesley, 1989.

[BN84] A. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):356–372, February 1984.

[CM88] K.M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison Wesley, 1988.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Ger89] C. Gerety. HP Softbench: A new generation of software development tools. Technical Report SESD-89-25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, November 1989.

[GKN88] D. Garlan, G.E. Kaiser, and D. Notkin. On the criteria to be used in composing tools into systems. Technical Report 88-08-09, Department of Computer Science, University of Washington, Seattle, WA, August 1988.

[GN91] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, pages 31–44, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag, LNCS 551.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), October 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[ISO87] ISO. Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour. Technical Report ISO/TC 97/SC 21, International Standards Organization, 1987.

[Jon83] C.B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and Systems*, 5(4):569–619, October 1983.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume Lecture Notes in Computer Science, volume 92. Springer-Verlag, 1980.

[OG76] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–284, May 1976.

[Rei90] S.P. Reiss. Connecting tools using message passing in the FIELD program development environment. *IEEE Software*, July 1990.

[SN92] K. Sullivan and D. Notkin. Reconciling environment integration and component independence. *ACM Transactions on Software Engineering and Methodology*, 1(3), July 1992.

[Stø91] K. Stølen. A method for the development of totally correct shared-state parallel programs. In *CONCUR '91*, pages 510–525. Springer Verlag, 1991.