Formal Aspects of Computing (1998) 3: 1–??
© 1998 BCS

Towards a formal treatment of implicit invocation using rely/guarantee reasoning

J. Dingel[1], D. Garlan[1], S. Jha[1], and D. Notkin[2]

[1]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA
[2]Department of Computer Science, University of Washington, Seattle, WA 98195, USA

Keywords: Implicit invocation; rely/guarantee; assumption/commitment

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.

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 methods

---

Correspondence and offprint requests to: J. Dingel, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, `dingel@cs.cmu.edu`
Acknowledgement of sponsorship: 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.
Disclaimer: 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 or the U.S. Government.

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 methods. 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 method A can invoke another method B without A being required to know B's name. Methods such as B "register" interest in particular "events" that methods such as A "announce." When A announces such an event, the II mechanism is responsible for invoking method B, even though A doesn't know that B or any other methods are registered.

There are two fundamental benefits of using the II architectural style. First, II mechanisms facilitate building systems that relieve users from having to explicitly invoke related components. This is because complex interaction between methods need not be directly encoded in the methods themselves. Methods can be built largely independently, but still cooperate in supporting a common goal. Second, II mechanisms reduce the cost of system evolution [SN92]. In particular, because methods are loosely coupled, it is possible to integrate new methods without affecting the methods that implicitly invoke the new methods.

Due to these features, II has been used in diverse settings such as programming environments and operating systems. 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, JAVABEANS [Jub98]), 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 method announce sufficient events to permit effective integration? If a new method is added to an existing system, will it break the existing system? Are there the right methods to produce the 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 state space to model the announcement of events. The specification language is enriched accordingly to express the presence of events. Moreover, location predicates capture the contents of the program counter, that is, where in each method control currently resides. The overall system behaviour can then be reasoned about compositionally by establishing invariants over the effects achieved by individual methods together with the state of pending events (i.e.,

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

those waiting to implicitly-invoke other computations) and the location predicates.

While the rely/guarantee paradigm proves to be robust and powerful enough to handle II in general, the paper also exposes some inherent limitations. Since our work on formal treatments of II is ongoing, we will discuss the advantages and disadvantages of our approach and how we expect these insights to shape our future work.

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 method A can invoke a method 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.[2]

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 postconditions for procedures and uses standard compositional techniques over the primitives to demonstrate that the axiomatic conditions hold. These conditions are in turn used as primitives to prove properties about the enclosing procedures.

[2] 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.

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:

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 method from the system in which its events are bound to other methods. This is because changing any binding requires reanalysis of the methods 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 methods 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 methods 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 define and specify method interfaces and mechanisms for reasoning about aggregate behaviour. This theory would allow us to determine:

- What is the desired interface of a method?
- Does a given method 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, Barrett et al. [BCTW96] have 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. Three concepts are crucial to the model: *events*, *methods* and *bindings*.

Events

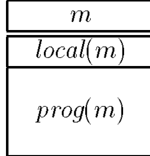
Formally, events are ground terms like *incr*, *modified* or *ins*(3), *del*(4). The main purpose of an event is to trigger other methods. Typically, the event thus communicates a certain state change that the rest of the system needs to know about. In other words, an event is announced if and only if 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 $sem(e)$.

Events will also be used to communicate data. To this end, we define *event templates* $e(x)$. Formally, an event template $e(x)$ is a term containing at most one variable x. Examples are *modified* and $ins(x)$. An event e is said to match an event template $e'(x)$ with substitution $[x = v]$ if the substitution unifies e and $e'(x)$, that is, $e = e'(v)$. For instance, event $ins(3)$ matches $ins(x)$ with $[x = 3]$. However, there is no substitution such that $ins(3)$ matches $del(y)$. Note that the definition of event templates and matching could easily be extended to an arbitrary number of variables.

Methods

A method m is a piece of (imperative) code, denoted by $prog(m)$ 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 program of m and changes to them are not visible to the outside. The general structure of a method m thus is:



The bindings of local variables $local(m) = \{x_1, \ldots, x_n\}$ are recorded in the program c itself and supersede the bindings of global variables with the same name. To this end, c is required to be of the form

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

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, sequential, imperative language augmented with primitives for announcing and consuming events, **announce**(e) and **consume**$(e(x))$ where e is an event and $e(x)$ is an event template. We restrict the occurrence of a **consume** statement to the very beginning of C, that is, C has to be either of the form C' or **consume**$(e(x)); C'$ for some event template $e(x)$ where

$$\begin{array}{lll} C' & ::= & x := expr \mid \\ & & C'_1; C'_2 \mid \\ & & \textbf{if } B \textbf{ then } C'_1 \textbf{ else } C'_2 \mid \\ & & \textbf{while } B \textbf{ do } C' \mid \\ & & \textbf{announce}(e). \end{array}$$

Intuitively, a method m whose program begins with **consume**$(e(x))$ will only be executed if an event e' is announced such that e' matches $e(x)$ and e' is bound to m. Such an event is said to trigger m. We define

$$trig(m) = \{e \mid e \text{ triggers } m\}.$$

However, if m does not begin with a **consume** statement, then it does not need to be triggered. In that case,

$$trig(m) = \emptyset.$$

Note that the framework could easily be extended to allow for multiple **consume** statements at the beginning of a program. For the purposes of this paper however, the special case described above suffices.

The formal semantics of **announce** and **consume** statements will be given in the next section.

Bindings

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) such that

$$(e, m) \in EM \Rightarrow e \in trig(m) \qquad (1)$$

for all (e, m). An event can thus not be bound to a method that is not triggered by that event. 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 2.1. 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 a "pending event" data structure pe. Concurrent with method executions, event-method pairs are removed from pe, causing the invocation of the associated methods. Let l be a list of (e, m) pairs. We assume that pe supports two update operations $store(pe, l)$ and $remove(pe, (e, m))$, two predicates $empty(pe)$ and $(e, m) \in pe$, and an operation $\#(pe, e)$ that counts the number of occurrences of e in pe, that is, $\#(pe, e) = |\{(e, m) \mid (e, m) \in pe\}|$.

In this model, we leave unspecified how precisely the events are stored and retrieved. In other words, our model does not contain any built-in assumptions about, for example, the policy that decides which event-method pair will be selected from the pending 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. Note that the correctness of a system may depend on a specific policy. While we will identify the necessary policy for each of the examples in this paper, we will not attempt a general classification 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. Every variable $x \in V$ has a domain Dom_x associated with it. A *state* s is a total mapping from variables to values, $s : V \to \bigcup_{x \in V} Dom_x$. Whenever $(e, m) \in pe$ for some m, then event e is currently *pending* and still needs to be delivered to method m.

Definition 2.2. Let c_i be programs, s a state, and pe the pending event data structure. A *configuration* is a 3-tuple

$$\langle \langle c_i \rangle_{i=1}^n, s, pe \rangle$$

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

A *transition* is of the form $\langle \langle c_i \rangle_i, s, pe \rangle \xrightarrow{l} \langle \langle c_i' \rangle_i, s', pe' \rangle$ 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. □

Method semantics

Before the operational semantics of the overall system can be defined, we need to give a semantics for the program $prog(m)$ of a method m. This semantics is a family of local transition relations $\leadsto_{(EM,m)}$ 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 $\prec c, s, pe \succ$ and $\prec s, pe \succ$. A local transition

$$\prec c, s, pe \succ \ \leadsto_{(EM,m)} \ \prec c', s', pe' \succ$$

means that program c transformed state s and the pending events data structure pe to s' and pe' respectively assuming that c is executed under the binding EM, and that c is the program of method m. The remainder of the program is c'. If c is an atomic statement, that is, c terminates in one step, we have

$$\prec c, s, pe \succ \ \leadsto_{(EM,m)} \ \prec s', pe' \succ.$$

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

$$\frac{}{\prec x := e, s, pe \succ \ \leadsto_{(EM,m)} \ \prec [s|x = v], pe \succ} \qquad \text{if } e \text{ evaluates to } v \text{ in } s$$

where $[s|x = v]$ denotes the state like s except that x has value v. Sequential composition is characterized by

$$\frac{\prec c_1, s, pe \succ \ \leadsto_{(EM,m)} \ \prec c_1', s', pe' \succ}{\prec c_1; c_2, s, pe \succ \ \leadsto_{(EM,m)} \ \prec c_1'; c_2, s', pe' \succ}$$

and

$$\frac{\prec c_1, s, pe \succ \ \leadsto_{(EM,m)} \ \prec s', pe' \succ}{\prec c_1; c_2, s, pe \succ \ \leadsto_{(EM,m)} \ \prec c_2, s', pe' \succ}$$

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

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

where $EM(e) = \{m_1, \ldots, m_n\}$. That is, $\mathbf{announce}(e)$ 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 pending event data structure, that is, pe remains unchanged.

As we will see later, the methods execute asynchronously, that is, they are not required to move in lockstep. The only means of synchronization is the **consume** statement. The execution of $\mathbf{consume}(e(x))$ in method m blocks until pe contains a pair (e', m) such that e' matches $e(x)$ with some substitution $[x = v]$. Then, the current state s is updated to $[s|x = v]$ and the event e' is considered delivered and the pair (e', m) is removed from the pending event data structure.

$$\frac{}{\prec \mathbf{consume}(e(x)), s, pe \succ \ \rightsquigarrow_{(EM,m)} \ \prec [s|x = v], remove(pe, (e', m)) \succ}$$

if $(e', m) \in pe$ and e' matches $e(x)$ with $[x = v]$.

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 are updated with their local values, C has a transition to $\prec C', s', pe' \succ$, then $\prec \mathbf{local}\ dl\ \mathbf{in}\ C, s, pe \succ$ 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'.

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

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.

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

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

- environment transitions:

  $$\langle \langle c_i \rangle_i, s, pe \rangle \xrightarrow{env}_{EM} \langle \langle c_i \rangle_i, s', pe' \rangle$$

  for all c_i, s, pe, s', pe' and
- program transitions:

  $$\langle \langle c_1, \ldots, c_i, \ldots, c_n \rangle, s, pe \rangle \xrightarrow{pro}_{EM} \langle \langle c_1, \ldots, c_i', \ldots, c_n \rangle, s', pe' \rangle$$

  whenever

1. $\prec c_i, s, pe \succ \;\rightsquigarrow_{(EM,m_i)}\; \prec c_i', s', pe' \succ$, or
2. $\prec c_i, s, pe \succ \;\rightsquigarrow_{(EM,m_i)}\; \prec s', pe' \succ$ and $c_i' = prog(m_i)$. □

The intuition behind the above definition is the following: The environment has access to the global state and the pending event data structure and can change these arbitrarily in an environment transition. A program transition arises from a local transition of the program of one of the methods. The program of a method is restored upon termination.

Note that an event cannot trigger a method that is already executing. In other words, at most one "incarnation" of each method is executing at any time. More precisely, if method m is currently executing, a pair (e, m) will remain in the pending event data structure until m is again ready to perform a matching consume. Also note that the operational semantics can readily be extended to handle, for instance, the use of more specific method activation strategies.

A configuration is called *disabled* if it has no transitions.

Definition 2.4. A configuration $\langle\langle c_i\rangle_i, s, pe\rangle$ is *disabled* under *EM* if there are no c_i', s' and pe' such that

$$\langle\langle c_i\rangle_i, s, pe\rangle \stackrel{pro}{\longrightarrow}_{EM} \langle\langle c_i'\rangle_i, s', pe'\rangle.$$

□

Note that a configuration in which all methods are blocked at **consume** statements is regarded disabled.

Definition 2.5. A *computation* under some binding *EM* is a possibly infinite sequence of program and environment transitions

$$\langle\langle c_{1_i}\rangle_i, s_1, pe_1\rangle \stackrel{l_1}{\longrightarrow}_{EM} \ldots \stackrel{l_{j-1}}{\longrightarrow} \langle\langle c_{j_i}\rangle_i, s_j, pe_j\rangle \stackrel{l_j}{\longrightarrow}_{EM} \ldots$$

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

Note that computations are not subject to a fairness constraint. As we will see, our approach is geared towards partial correctness, that is, only terminating computations will be considered. Consequently, a fairness constraint is not required.

Given a computation σ, then $C(\sigma)$, $S(\sigma)$, $PE(\sigma)$ and $L(\sigma)$ are the obvious projection functions to sequences of programs, states, pending events and transition labels. $\sigma[i]$, $C(\sigma, i)$, $S(\sigma, i)$, $PE(\sigma, i)$ and $L(\sigma, i)$ denote, respectively, the ith configuration $\langle\langle c_{i_j}\rangle_j, s, pe\rangle$, the ith vector of programs $\langle c_{i_j}\rangle_j$, the ith state s_i, the ith pending event data structure pe_i, and the ith label l_i of σ. Let $S{\times}PE$ be the product of the two projection functions S and PE, that is, $S{\times}PE(\sigma, i) = (S(\sigma, i), PE(\sigma, i))$.

Given a system $\mathcal{S}$, σ is a computation of $\mathcal{S}$ if the first configuration of σ is initialized with the programs of each of the methods in $\mathcal{S}$.

Definition 2.6. 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 $C(\sigma, 1) = \langle prog(m_i)\rangle_{i=1}^n$. □

Note that a computation σ may start in any initial state $S(\sigma, 1)$ and with any pending event datastructure $PE(\sigma, 1)$. Moreover, the environment can interfere arbitrarily along σ.

3. Specification language

Rely/guarantee reasoning [Jon83, Stø91] has successfully been applied to concurrent systems. We now show how 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 that a specific event e has been announced. To this end we introduce the predicate *pending*(e). Remember that the variable pe is reserved to denote the pending events data structure. Let pe and pe' be two pending event data structures. We say *pending*(e) is true in (pe, pe') if there is a method m such that $(e, m) \in pe'$, that is, $(pe, pe') \models pending(e)$ iff $(e, m) \in pe'$ for some m. A *state-event predicate* is the boolean combination of state predicates and the *pending* predicate and is thus interpreted over 4-tuples $((s, pe), (s', pe'))$ 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 $s_1(x) \neq s_2(x)$.

A specification places two restrictions on the environment of a computation. First, the initial state is assumed to satisfy P. Moreover, every environment transition that changes the state has to meet the rely-condition. For instance, $R = (x' \geq x \wedge y' = y)$ prevents the environment from decreasing x and from changing y while the remaining variables in V can be changed arbitrarily.

Definition 3.1. 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

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

Apart from placing assumptions on the environment, a specification also states commitments for the program. Given an environment which satisfies the assumptions, every program transition that changes the state is required to meet the guarantee-condition G. Moreover, if the computation (of the program and the environment) terminates, the final state has to satisfy Q.

Definition 3.2. Let V be the set of global program variables. Given a binding EM, a guarantee-condition G, an input/output-condition Q, then $prog(V, G, Q)$ denotes the set of all computations σ under EM, such that

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

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 starts in a state satisfying P and satisfies the rely-condition R on environment transitions, then every program transition will satisfy the guarantee-condition G. Also, if σ is finite, the last state will satisfy Q.

Definition 3.3. Let $\mathcal{S} = (M, V, EM, Ex)$ be a system. The judgement

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

is true iff

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

□

Since judgements do not guarantee termination, our reasoning can only address partial correctness behaviour. Jones' and Stølen's work [Jon83, Stø91], on the other hand, is centered around total correctness.

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

Definition 3.4. Let $\mathcal{S} = (M, V, EM, Ex)$ be a system. The set of *executions* of $\mathcal{S}$, $exec(\mathcal{S})$, is given by

$$exec(\mathcal{S}) = \{\sigma \in comp(\mathcal{S}) \mid \sigma \in env(V, empty(pe), R_{Ex}) \text{ and } \sigma \text{ is finite}\}$$

where R_{Ex} is

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

and thus restricts the top-level environment to the announcement of external events. For state-event predicates P and Q, the 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 in which 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 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 of other parts of the state.

As a simple example, consider a system in which the state consists of a set S and a counter C. 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

$$M \quad = \quad \{\mathit{insert}, \mathit{increment}, \mathit{delete}, \mathit{decrement}\},$$

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

$$\begin{aligned} EM \quad = \quad & \{(ins(n), \mathit{insert}), (del(n), \mathit{delete}) \mid n \in \mathbb{N}\} \cup \\ & \{(incr, \mathit{increment}), (decr, \mathit{decrement})\}. \end{aligned}$$

The idea is that a natural number n can be inserted into or deleted from the set S using the method *insert* or *delete*. Analogously, the counter C can be incremented or decremented using *increment* or *decrement*. In this case *EM* provides the necessary bindings for events announced by the methods that change the state of the set, so that the state of the counter can be updated accordingly. The methods are shown in Figure 1. Given the external event $ins(n)$, the formal parameter x of method *insert* is replaced by n and the method is invoked. Similarly for an external event $del(n)$. 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	$\exists x.x \in S \wedge S' = S \backslash \{x\}$

Note that the correctness of the system depends on the assumption that the *remove* operation retrieves the event-method pairs from the pending event datastructure *pe* in the same order as they were stored. In other words, the behaviour of the system is sensitive to the order in which *pe* stores and retrieves its elements. We say that events are not commutative. For instance, assuming $S = \emptyset$, the event sequence

$$ins(3)\ \ del(3)$$

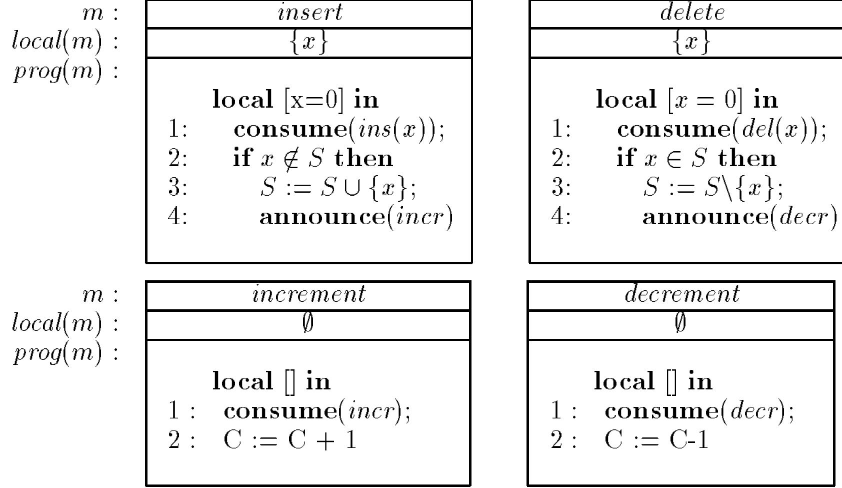


Fig. 1. The methods of the set/counter example

presented by a user leaves the system in a different state than the sequence

$del(3)\ ins(3)$.

Moreover, in general, events must not be dropped. However, consecutive occurrences of the same event form an exception. The sequence of events

$ins(3)\ ins(3)\ del(3)\ del(3)$

for example, has the same effect as

$ins(3)\ del(3)$.

These requirements suggest the implementation of pe as a queue.

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 rely-condition for 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 guarantee-condition 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 $R \vee G_{M\backslash\{m\}}$, then m will change the state according to G_m and if m terminates, 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 $R \vee G_m$, then $M\backslash\{m\}$ will change the state according to $G_{M\backslash\{m\}}$ and if $M\backslash\{m\}$ terminates, 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 guarantee-condition if the other one does and the entire system is executed in an initial state satisfying P and in an environment that satisfies R. Note how the guarantee-condition of a method implies the rely-condition of the rest of the system and vice versa.

(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,

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

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 $empty(pe)$ 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 $P_{\mathcal{S}} \wedge empty(pe) \Rightarrow P$ and $R_{Ex} \Rightarrow R$. Since an event can only be bound to methods that are actually triggered by that event (condition (1) on page 7) and due to the semantics of **announce**, a pair (e, m) in pe actually triggers m. Thus, the final state of every terminating computation must satisfy $empty(pe)$. Thus, if we can also show that in the final state $Q \wedge empty(pe)$ implies $Q_{\mathcal{S}}$, then the above judgement can be weakened to

$$(M, V, EM, Ex) \models (P_{\mathcal{S}} \wedge empty(pe), R_{Ex}, true, Q_{\mathcal{S}}).$$

3. Given the definitions of executions and the partial correctness triple, it is now sound to conclude that the partial correctness property

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

holds.

Following the standard treatments of rely/guarantee reasoning [Jon83, Stø91] a more general formulation of step 1 would be possible. However, the present treatment is sufficient for our purposes.

4.1. Program locations

It will be necessary to express where control resides during an execution of a method. For each method m, we thus reserve a variable pc_m that serves as a

program counter for m. More precisely, for each state s, $s(pc_m)$ is the number of the next statement in *prog*(m) to be executed. We assume that

- $pc_m = 1$ initially, that is, pc_m points to the first statement of m at the beginning of every execution.
- pc_m is implicitly updated by m.
- $pc_m = 1$ after the last statement of m has been executed, that is, methods are executed in an infinite loop.

Given a method m and a line number n, the predicate at_n^m is true in a state s if the program counter of method m points to the statement with line number n, that is, $s \models at_n^m$ iff $s(pc_m) = n$.

Consider the set/counter example. The predicate at_4^{insert}, for instance, expresses that the *insert* method has just added an element to S but not yet announced the corresponding *incr* event. Moreover, $\bigwedge_{m \in M} at_1^m$ states that all four methods are waiting at their initial **consume** statement. The final state of a terminating computation of $\mathcal{S}$ always satisfies $empty(pe) \wedge \bigwedge_{m \in M} at_1^m$, that is, all pending events have been delivered and all methods are blocked at their initial **consume** statements.

Note that auxiliary variables [OG76, Stø91] could also have been used instead of program locations.

4.2. 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 m_{incr} be

   $$m_{incr} = (\textbf{if } at_4^{insert} \textbf{ then } 1 \textbf{ else } 0) + (\textbf{if } at_2^{increment} \textbf{ then } 1 \textbf{ else } 0)$$

   and let m_{decr} be

   $$m_{decr} = (\textbf{if } at_4^{delete} \textbf{ then } 1 \textbf{ else } 0) + (\textbf{if } at_2^{decrement} \textbf{ then } 1 \textbf{ else } 0).$$

   Note how $m_{incr} = 1$ expresses that either

   (a) an element has just been added to S, but the corresponding *incr* event has not yet been announced, or

   (b) an *incr* event has just been consumed, but the corresponding increment of C has not yet taken place.

   We have $m_{incr} = 2$, if both cases hold. m_{decr} has an analogous interpretation. Let I_1 be given by

   $$I_1 \;=\; (|S| = C + \#incr + m_{incr} - \#decr - m_{decr})$$

   where $\#e$ abbreviates the number of occurrences of e in pe, that is, $\#(pe, e)$. 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 $G_m = G_{M\backslash\{m\}} = Q = I_1$ for all $m \in M$. We can show that all methods preserve I_1.

$$(m, V, EM, Ex) \quad \models \quad (C = |S|, R_{Ex} \vee I_1, I_1, I_1)$$

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

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

$$(M\backslash\{m\}, V, EM, Ex) \models (C = |S|, R_{Ex} \vee I_1, I_1, I_1)$$

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

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

2. We weaken the specification $(C = |S|, R_{Ex}, I_1, I_1)$ to

$$(C = |S| \wedge empty(pe), R_{Ex}, true, C = |S|).$$

Note that $C = |S| \wedge empty(pe) \Rightarrow C = |S|$. Also, remember that every final state of a terminating computation of $\mathcal{S}$ must satisfy $\forall m \in M.at_1^m$. Moreover, $I_1 \wedge empty(pe) \wedge \forall m \in M.at_1^m$ implies $C = |S|$. Thus,

$$\mathcal{S} \quad \models \quad (C = |S| \wedge empty(pe), R_{Ex}, true, C = |S|)$$

which implies

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

4.3. 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 $\mathcal{FS}$. Suppose *Src* is a set of source files. We assume that all the files in *Src* correspond to an executable file *exe* and that **make**(*Src*, *exe*) rebuilds the system by consulting some make file and creating a new executable with respect to the current contents of *Src*. Source files are changed by means of an editor method *edit* whereas the executable is updated by a compiler method *cmpl*. Let f range over files in $\mathcal{FS}$, that is,

$$Dom_f = \{v \mid v \text{ is file in } \mathcal{FS}\}.$$

The system $\mathcal{FS}$ contains the methods $M = \{edit, cmpl\}$, the internal event *modified*, the external events $Ex = \{ed(v) \mid v \in Dom_f\}$, and the binding EM with

$$EM = \{(ed(v), edit) \mid v \in Dom_f\} \cup \{(modified, cmpl)\}.$$

Let *fresh* denote the fact that the last modification date of *exe* is more recent than that of all files in *Src*, that is, for all $f \in Src$,

$$date_last_modified(exe) \geq date_last_modified(f).$$

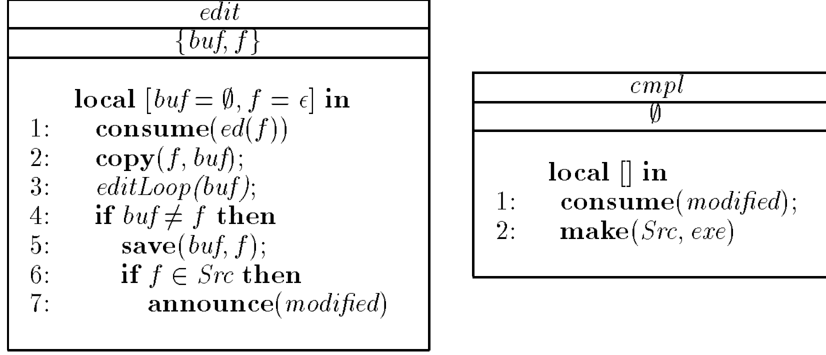


Fig. 2. Methods of the file system example where ϵ denotes the empty file

The semantics of the *modified* event is

$$sem(modified) \quad = \quad \neg fresh.$$

We assume that the methods are of the form given in Figure 2. An external $ed(f)$ event causes the file f to be edited. The *edit* 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, that is, $buf \neq f$ holds, then f is updated with *buf*. If f also is a source file relevant to *exe*, the *modified* event is announced. The *modified* event triggers the *cmpl* method which rebuilds the system with respect to the current contents of *Src* and thus generates a new executable. Like in the set/counter example, the events are not commutative. For instance, the sequence

$$ed(v)\ modified$$

cannot be expected to always have the same effect as the sequence

$$modified\ ed(v).$$

Thus, *pe* needs to follow a queue discipline. We would like to show that

$$\{fresh\}\ \mathcal{FS}\ \{fresh\}.$$

We again follow the methodology presented at the beginning of this section and first establish an invariant. Remember that $pending(e)$ abbreviates $\exists m.(e, m) \in pe'$. Let

$$I_2 \;=\; fresh \vee (\neg pending(modified) \Rightarrow at_6^{edit} \vee at_7^{edit} \vee at_2^{cmpl}).$$

Informally, I_2 expresses that either the system is fresh or *modified* is pending or the system is about to either announce *modified* or compile. We can show that I_2 is maintained by *edit* and *cmpl*.

$$(edit, V, EM, Ex) \quad \models \quad (fresh, R_{Ex} \vee I_2, I_2, I_2)$$

$$(cmpl, V, EM, Ex) \quad \models \quad (fresh, R_{Ex} \vee I_2, I_2, I_2)$$

Thus, we can conclude

$$\mathcal{FS} \quad \models \quad (fresh, R_{Ex}, I_2, I_2).$$

Note that the final state in a terminating computation of $\mathcal{FS}$ must satisfy $at_1^{edit} \wedge$

at_1^{cmpl}. Since $fresh \wedge empty(pe)$ implies $fresh$ and $I_2 \wedge empty(pe) \wedge at_1^{edit} \wedge at_1^{cmpl}$ implies *fresh*, the above judgement can be weakened to

$$\mathcal{FS} \quad \models \quad (fresh \wedge empty(pe), R_{Ex}, true, fresh)$$

from which we obtain the desired result

$$\{fresh\}\ \mathcal{FS}\ \{fresh\}.$$

Note that a pending *modified* event does not necessarily indicate that the system is not fresh. Suppose, for instance, the user wants to change some source file f_2. Right after f_2 has been updated, the system is not fresh. However, by the time the corresponding *modified* event is announced, the system may be fresh again. This is because the *cmpl* method maybe executing in parallel due to a previous editor invocation involving some other source file f_1. So, when the *modified* event is announced, the system may already be fresh. Consequently, the second compile may be unnecessary. In general, if the update of f_2 is completed before the compile corresponding to the update of f_1 starts, the compile corresponding to the update of f_2 will be unnecessary. However, this situation does not threaten correctness, because compilation always generates a fresh system, no matter if the system was fresh initially or not. In other words, *cmpl* still maintains the invariant. A pending *modified* event should thus be thought of as a need for recompilation at some point in the past which may or may not have been satisfied in the meantime. A pending *incr* event in the set/counter example, on the other hand, represents an absolute need to increment the counter which cannot be met by a parallel increment corresponding some other instance of *incr*. Also note that the efficiency of the file system example could thus be improved by requiring **make**(*Src*, *exe*) to cause recompilation only if the system is not fresh and to do nothing otherwise.

5. Conclusion and future 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.

While the rely/guarantee setting has proven robust and powerful enough to handle II in general, our work has also exposed the following limitations.

- The problem of atomicity is inherent to concurrent systems with shared resources. It resurfaces in this work with the following consequences. To allow for fine-grained parallelism we 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. The fine-grained parallelism thus forced us to weaken our invariants by location predicates which allow us to express that a method is about to change the

state in a certain way. In the file system example, for instance, the update of a source file and the announcement of the *modified* event do not coincide. The location predicate $at_6^{edit} \vee at_7^{edit}$ expresses that the edit method has just updated a source file and is about to announce the *modified* event. While this treatment is sufficient, it also is unnecessarily restrictive. What is essential in the file system example is that every update of a source file is eventually followed by the announcement of the *modified* event. The precise number and identity of the intermediate states that separate the update from the announcement is immaterial to the correctness of the system. Nonetheless, the rely/guarantee framework with its emphasis on invariants forces us to make this information explicit.

- Another shortcoming of our framework is the need for an explicit **consume** statement. On the one hand, it allows us to model the "invocation" of a method and to pinpoint changes to the pending event data structure. On the other hand, it compromises practicality and maintainability. II systems in general do not have an explicit **consume** statement. Instead, system runtime mechanisms invoke the method bound to an event, automatically removing that event from pending event set. Moreover, the explicit consumption of events introduces an unnecessary dependency between the event-method binding *EM* and the program of a method as witnessed by condition (1) on page 7. Changes to the binding *EM* must be reflected by changes to the **consume** statements and vice versa.
- An important requirement of the correctness of an II system is that events get announced precisely when they should. This aspect consists of two parts. On the one hand, the announcement of an event e should be justified by a state change that satisfies the semantics of e. On the other hand, every state change that satisfies the semantics of e, should be "covered" by an announcement of e. Note that we do not impose any assumptions on the temporal order or relative timing between event announcement and state change. In other words, the announcement of events should be sound and complete with respect to the event semantics. The event semantics should thus constitute a useful abstraction and reasoning tool. However, our verification methodology does not mention it all. Unfortunately, Jones' rely/guarantee approach is not suited to express this kind of property. Temporal logic, however, allows us to express that, for example, a specific state change is always eventually followed by the announcement of a specific event and thus seems more promising in this respect.

As already mentioned, this paper reports on work in progress. We expect the above results and insights to influence our future work as follows. The most important focus will be the development of a verification framework that does not impose the restrictions discussed above. More precisely, we would like the reasoning to be more abstract and not depend on the use of low-level invariants or program locations. Moreover, the framework should not depend on the explicit consumption of events. Finally, the fact that events get announced precisely when they are supposed to should be directly expressible, so that the event semantics can take on a more prominent role. More precisely, we envision the event semantics to form the basis of a two stage process: First, each of the methods is shown to behave correctly, that is, some suitably chosen local property is established. The reasoning during this phase is entirely local and involves, apart from the program of the method, only the event semantics and the binding.

During the second phase, the local properties are then combined to show the overall property. Obviously, some kind of independence property akin to the interference freedom property [OG76] will be needed to reconcile concurrency and compositionality.

In our opinion, the above requirements suggest the use of temporal logic. Collette has shown how compositional reasoning about temporal logic specifications can be introduced into a UNITY setting [Col94], and his work may be relevant.

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.

[Col94] P. Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications — Application to UNITY*. PhD thesis, Université Catholique de Louvain, Belgium, June 1994.

[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.

[Jub98] H. Jubin. *Javabeans by example*. Upper Saddle River: Prentice Hall, 1998.

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

[OG76] S.S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 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.