Reasoning about implicit invocation*

J. Dingel D. Garlan S. Jha
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{jurgend|garlan|sjha}@cs.cmu.edu

D. Notkin
Dept. of Computer Science and Engineering
University of Washington
Seattle, WA 98195
notkin@cs.washington.edu

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. Based on standard notions from process algebra and trace semantics, we define a formal computational model for implicit invocation. A verification methodology is presented that supports linear time temporal logic and compositional reasoning. First, the entire system is partioned into groups of components (methods) that behave independently. Then, local properties are proved for each of the groups. A precise description of the cause and the effect of an event supports this step. Using local correctness, independence of groups, and properties of the delivery of events, we infer the desired property of the overall system. Two detailed examples illustrate the use of our framework.

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]

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.

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 [Sun93], DecFuse), communication standards (e.g., Corba [Cor91]), integration frameworks (e.g., OLE, JavaBeans [Jub98]), and programming environments like Smalltalk [Gol84].

However, there is currently no established methodology for reasoning about II systems. In particular it is difficult to

*Effort sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-96-1-0299 and F30602-96-1-0301, and the National Science Foundation under Grant No. CCR-9633532 and No. CCR-9633462. 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.

SIGSOFT '98 11/98 Florida, USA

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

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

answer questions like: What will be the effect of announcing a given event? Have enough event bindings been declared to achieve the 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? Moreover, to fully support the intent of II, the reasoning should be *compositional*. More precisely, the verification of a given component should as much as possible be decoupled from the verification of 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.

This paper presents a formal model for systems designed using the II architectural style. The model combines standard notions from process algebra and trace semantics [Mil80, Hoa85] and allows the development of a compositional verification methodology for II systems. Informally, an II system $\mathcal{S}$ consists of a set of methods m_i and a distinguished dispatcher method *disp* which explicitly models the delivery and storage of events E. An event-method binding B determines which methods are triggered by which events. Each event $e \in E$ has a semantics associated with it that gives precise meaning to the generation and consumption of events. The *cause* of an event captures the state change that caused the generation of the event. The *effect* of an event captures the state change that the event will give rise to.

Suppose system $\mathcal{S}$ with methods

$$M \equiv \{m_1, \ldots, m_n, disp\}$$

is to be verified with respect to some specification φ. Our methodology consists of the following three phases.

- **Phase 1** (Decomposition)
  The set of methods M is partitioned into groups
  $$\{G_1, \cdots, G_k\}$$
  with $1 \leq k \leq n$. For each group G_i we find a local property φ_i. Groups are independent in the following sense: if G_i satisfies φ_i, then the entire system also satisfies φ_i. We also prove a local property φ_{disp} about the dispatcher method *disp*. The property φ_{disp} captures the minimal requirements on the binding and the dispatch policy of events. For instance, in all non-trivial cases the binding needs to be non-empty and the dispatcher is required not to lose certain or even all events.
- **Phase 2** (Local reasoning)
  Each group G_i is verified with respect to the local property φ_i. Moreover, the dispatcher is verified with respect to φ_{disp}. Typically, this step uses both the event-method binding B and the semantics of the events used by group G_i.
- **Phase 3** (Global reasoning)
  We show that the local correctness of each of the groups and the dispatcher implies the correctness of $\mathcal{S}$ with respect to φ. Independence ensures soundness of this phase.

In general, the tractability of this methodology depends on the number of independent groups that the system can be split into. We believe that the loosely-coupled nature of II systems naturally supports the formation of a large number of independent groups; that is, we expect the number of groups k to be linear in the number of methods n rather than a small constant.

1.1 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 a 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 [GKN92] in which the authors illustrated how and why the ideas of II systems are pervasive in software systems. Sullivan and Notkin showed how a particular style of use of II, which they call mediators, simplifies some specific classes of system change [SN92]. More recently, [BCTW96] produced a taxonomic survey of event-based mechanisms, together with a generic object model for comparison of them. By providing a general framework for all systems that use events as a communication mechanism (including, for example, remote procedure call) their work is concerned with a much broader class of systems than is our research. By focusing on the more specialized domain of implicit invocation systems, our models need not include all of the taxonomic entities that they propose, but are tailored to provide a more substantial analytic basis for formal reasoning about the behavior of such systems.

Closer to our line of research, some efforts have attempted to provide a formal characterization of certain aspects of II systems. An early characterization of II in Z captured structural and basic behavioural aspects, but no fundamentals of semantics [GN91]. More recently, researchers in software architecture have looked at some of the formal properties of II architectural styles [AAG95], but this research has focussed on taxonomic issues, and does not provide an explicit computational model that permits compositional reasoning about the behaviour of such systems.

In an earlier paper [DGJN98], we investigated the use of Jones' rely-guarantee framework [Jon83]. Here, compositionality is achieved by restricting the behaviour of the environment with a single logical formula, called environment assumption or invariant. To discharge this assumption the environment then has to be shown to satisfy this invariant. Since the invariant has to be preserved by *every* transition, this is a very strong requirement that typically can only be met after weakening the invariant with location predicates that describe the value of the program counter. A weakened invariant thus typically expresses that either the invariant already holds or certain statements are about to be executed which reestablish it. Consequently, the reasoning becomes unnecessarily detailed. We are forced to explicitly keep track and expose the number and identity of intermediate states even if this information is completely irrelevant to the correctness of the system. In the present paper we strive to overcome this deficiency with the help of temporal logic.

Other researchers have investigated at formal aspects 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.

As we will see, this paper uses the UNITY program-

ming language [CM88] augmented with a few communication primitives to provide a semantic base for implicit invocation. One possible alternative would have been to use Linda's tuple space [GZ97] as the underlying model. However, the match between tuple spaces and implicit invocation did not appear to be a natural one: II systems are sensitive to the relative order in which events are communicated, and a tuple space's inherent non-determinism would thus have to be restricted.

In the remainder of this paper we describe a formalization of implicit invocation systems. The next section introduces a formal model for II systems. Section 3 describes the specification formalism. Section 4 presents the verification methodology. Section 5 concludes and outlines further work.

2 Modeling implicit invocation systems

An implicit invocation system will be modeled as a collection of methods that anonymously exchange messages (events) by means of a dispatcher and an event-method binding. A method is a UNITY program augmented with communication primitives for sending and receiving messages. We employ a notion of communication similar to Milner's CCS [Mil80]. There are three types of actions a. a is either

- the *silent action* τ,
- an *input action* $\langle m, z\rangle?$ or $\langle m, e\rangle?$, or
- an *output action* $\langle m, e\rangle!$,

where m is some method, e is some event in E, and z is some variable ranging over events. An input action $a_1 \equiv \langle m_1, z\rangle?$ or $a_1 \equiv \langle m_1, e\rangle?$ and an output action $a_2 \equiv \langle m_2, e\rangle!$ are said to *match*, if $m_1 = m_2$. Synchronization is achieved through matching actions. Intuitively, if a method m_1 announces an event e meant for a method m_2, it performs the output action $\langle m_2, e\rangle!$. Method m_2 consumes the event e by synchronizing with the above action by performing one of the input actions $\langle m_2, z\rangle?$ or $\langle m_2, e\rangle?$. The synchronization then gives rise to the silent action τ and also assigns e to z in case $\langle m_2, e\rangle!$ is matched with $\langle m_2, z\rangle?$.

To allow for a "selective receipt" of events, input actions could be augmented with a predicate p, such that $\langle m, z, p\rangle?$ matches $\langle m, e\rangle!$ only if e satisfies p. As in Field [Rei90], different methods could thus "listen" for different sets of actions.

Definition 2.1 *A* method *m is a 4-tuple*

$$m \equiv (V, E, P, S)$$

where

- *V is the set of variables that m accesses. Each variable x has a domain Dom_x associated with it,*
- *E is a set of events that m announces,*
- *P is a boolean expression over V describing the set of initial states,*
- *S is a set of* statements *of the form*

$$g \xrightarrow{a} x := exp$$

where

  - *g is a boolean expression over V called* guard,
  - *a is an* action,
  - *$x := exp$ is an assignment where $x \in V$ and exp is an expression over V.* □

The semantics of a method is similar to that of a UNITY program [CM88]. The method executes the statements in an infinite loop using the following strategy. First, a statement

$$g \xrightarrow{a} x := exp \in S$$

is chosen non-deterministically. If g holds in the current state, the action a is carried out. If $a = \langle m, z\rangle?$, then we input the next event addressed to method m and assign it to z. Next, the assignment $x := exp$ is executed by evaluating the expression exp in the current state and then updating variable x. If the environment of m does not offer a matching output action, we get a stuttering step, that is, the assignment is not carried out and the execution of the statement terminates in the same state. The case $a = \langle m, e\rangle?$, is similar except that no variable update takes place. The communication thus only has a synchronizing effect. If $a = \langle m, e\rangle!$, we output the event e to method m and then evaluate the assignment. Again, if the environment does not offer a matching input action, the statement terminates with a stuttering step. Finally, if $a = \tau$, we immediately evaluate the assignment. Note that execution of an assignment is assumed to be atomic. If the guard is not true in the current state, the execution of the statement terminates immediately in the same state. Just like in UNITY, we adopt the fairness constraint that every statement will be executed infinitely often.

The recipients of an event are determined by the binding.

Definition 2.2 *Let E be a set of events and M a set of methods. A (possibly empty) set $B \subseteq E \times M$ is called a* binding over *E and M.* □

A binding associates each event e with zero or more methods that are to be triggered when that event is announced. Note that an event need not be bound to any methods and that several methods can be bound to the same event.

Given a binding B, the delivery of events is modeled explicitly through a distinguished dispatcher method $disp_B$, frequently also denoted by *disp* if the binding is understood or irrelevant. A method announces an event e by sending it to the dispatcher. In practice, the number of events that a dispatcher can handle at a given time is bounded by some number *max*. If the dispatcher is not filled to its capacity *max*, it consumes the event, looks up which methods e is bound to and then stores all resulting pairs (e, m) in a *pending events datastructure* D that keeps the events that are yet to be delivered. Concurrently, the dispatcher can retrieve a pending event from D and send it to a method it is bound to. The dispatcher is given in Figure 1. For notational convenience and without loss of generality, we will always represent the list of statements S in terms of a simple, imperative, shared-variable concurrent language augmented with two communication primitives for sending and receiving messages. The translation from this representation to the one in Definition 2.1 is straightforward [CM88]. To model sequential execution, for instance, a program counter pc is introduced for each method m that always points to the next statement in m to be executed. Moreover, we use the following abbreviations. $\langle m, z\rangle?$ and $\langle m, e\rangle?$ stand for

$$true \xrightarrow{\langle m,z\rangle?} \mathbf{skip}$$

$disp_B$:

$\{D, z, m\}$	V
$\emptyset$	E
$empty(D)$	P
[**if** $size(D) < max$ **then**	S
consume(z);	
for $(z, m) \in B$ **do**	
$D := store(z, m, D)$ ||	
if $\neg empty(D)$ **then**	
$(z, m) := next(D)$;	
$\langle m, z \rangle!$;	
$D := remove(z, m, D)$]	

Figure 1: The dispatcher method $disp_B$

and

$$true \xrightarrow{\langle m,e \rangle?} \textbf{skip}$$

respectively. $\langle m, e \rangle!$ abbreviates

$$true \xrightarrow{\langle m,e \rangle!} \textbf{skip}.$$

An occurrence of **consume**(z) in method m abbreviates $\langle m, z \rangle?$ and **announce**(e) abbreviates $\langle disp_B, e \rangle!$. The statement $store(e, m, D)$ stores the pair (m, e) in D and returns the updated D; if D is not empty, $next(D)$ returns the next element stored in D; if (e, m) is in D, $remove(e, m, D)$ removes it from D and returns the updated D. $size(D)$ yields the number of elements stored in D and $empty(D)$ returns true if and only if D is empty. For the sake of generality, we intentionally make as few assumptions about an implicit invocation system as possible. For example, the storage policy of pending events in D is left unspecified. An example for a policy would be a first-in-first-out discipline that does not remove duplicate occurrences of pairs. In other words, the model is supposed to abstract from specific event storage policies so that any possible policy can be plugged in easily.

For the dispatcher to fulfill its purpose, all communication needs to be forced through it. In other words, whereas the dispatcher must be able to communicate with every method (except itself), all other methods must be prevented from communicating with each other directly. We thus impose the following *topology constraint*: All methods except the dispatcher must use **announce**(e) and **consume**(z) to send and receive messages. In other words, every output action and every input action in a method m except the dispatcher must be of the form $\langle disp, e \rangle!$ and $\langle m, z \rangle?$ respectively.

A set of methods m_i that satisfy the topology constraint together with a binding B and a dispatcher $disp_B$ form a system. Given a method

$$m_i \equiv (V_i, E_i, P_i, S_i),$$

let $E(m_i)$ and $P(m_i)$ denote E_i and P_i respectively.

Definition 2.3 *An* implicit invocation system $\mathcal{S}$, *or* system *for short, is a 4-tuple*

$$\mathcal{S} \equiv (M, P, E, B)$$

where

- M *is a set of* methods m_i *together with a distinguished* dispatcher method $disp_B$, *that is,*

$$M \equiv \{m_1, \ldots, m_n, disp_B\}$$

with $n \geq 1$, *where* m_1 *through* m_n *satisfy the topology constraint,*

- P *describes the initial states of the system. It must be consistent with the initial states of each of the methods, that is,* $P \Rightarrow \bigwedge_{m \in M} P(m)$,

- $E \equiv \bigcup_{m \in M} E(m)$, *is the set of all events,*

- B *is a binding over* E *and* $\{m_1, \ldots, m_n\}$.

The actions of a system are collected in

$$\begin{aligned} InOut &\equiv \{\langle m, e \rangle?, \langle m, e \rangle! \mid m \in M, e \in E\} \\ Act &\equiv InOut \cup \{\tau\}. \end{aligned}$$

□

Note that the same variable can be accessed by more than one method. Methods thus can also communicate through shared variables.

From an implementation point of view, we can think of a system as a network of processes (methods) that are connected through input ports as shown in Figure 2. p_m denotes the input port of process (method) m. Note how the dispatcher controls the flow of events.

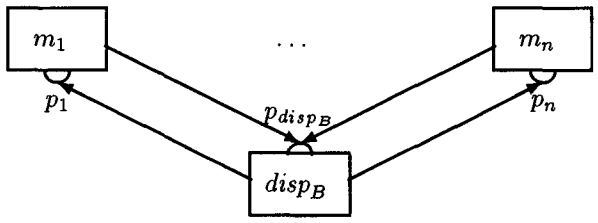

Figure 2: Implicit invocation system as network

2.1 Modeling the environment

Typically, a system is triggered directly by some "top-level" (or "external") events that are provided by the user. The environment model represents all allowed sequences of input and output actions that may be presented to some set of methods.

Definition 2.4 *Given a system with input and output actions InOut, an* environment model *Env is a (possibly empty) set of finite sequences of input and output actions, that is,* $Env \subseteq InOut^*$. □

Although the above definition is a lot more general, we will only employ two kinds of environment models in this paper.

- To define the semantics of an event we will need environments that can only execute a single action $a \in InOut$. The corresponding model thus is of the form $\{a\}$.

- Moreover, to model an arbitrary but finite stream of "top-level" actions supplied by a user, we will use environment models of the form $\{a_1, \ldots, a_n\}^*$ where $a_i \in InOut$ for all $1 \leq i \leq n$.

The behaviour of an environment model *Env* will be implemented by the method m_{Env}. The method corresponding to $Env \equiv \{a_1, \ldots, a_n\}^*$ is given in Figure 3 where the execution of

$$n := \mathbf{choose}(\mathbb{N})$$

assigns a random natural number to n and

$$\mathbf{choose}(a_1, \ldots, a_m)$$

non-deterministically chooses an action a_i with $1 \leq i \leq m$.

m_{Env} :

$\emptyset$	V
$\{a_1, \ldots, a_m\}$	E
true	P
$n := \mathbf{choose}(\mathbb{N})$; **for** $i = 1$ **to** n **do** $\mathbf{choose}(a_1, \ldots, a_m)$	S

Figure 3: The method m_{Env} corresponding to $Env \equiv \{a_1, \ldots, a_m\}^*$

2.2 Example: Sets and counters

We show how the above model of an implicit invocation system can be instantiated by a specific example. Consider a system $\mathcal{SC}$ which maintains a set S of elements over some domain Dom_x and a counter C. Initially, $S = \emptyset$ and $C = 0$. Besides the dispatcher the system contains two methods which are given in Figure 4. An element x can be inserted into or

set :

$\{x, z_1, S\}$
$\{ins, del\} \cup \{insert(v), delete(v) \mid v \in Dom_x\}$
$S = \emptyset$

$\mathbf{consume}(z_1)$;
if $z_1 = insert(x)$ **then**
  if $x \notin S$ **then**
  $S := S \cup \{x\}$;
  $\mathbf{announce}(ins)$
elsif $z_1 = delete(x)$ **then**
  if $x \in S$ **then**
  $S := S \backslash \{x\}$;
  $\mathbf{announce}(del)$

cnt :

$\{C, z_2\}$
$\{ins, del\}$
$C = 0$

$\mathbf{consume}(z_2)$;
if $z_2 = ins$ **then**
  $C := C + 1$
elsif $z_2 = del$ **then**
  $C := C - 1$

Figure 4: Methods *set* and *cnt*

deleted from the set S using the method *set*. Analogously, the counter C can be incremented or decremented using *cnt*. The binding is

$$B \equiv \{(ins, cnt), (del, cnt)\}.$$

Thus,

$$M \equiv \{set, cnt, disp_B\}$$

and

$$E \equiv \{ins, del\} \cup \{insert(v), delete(v) \mid v \in Dom_x\}.$$

Execution is triggered by a finite sequence of *insert* or *delete* actions addressed to the *set* method. We define

$$Env \equiv \left\{\langle set, insert(v)\rangle!, \langle set, delete(v)\rangle! \mid v \in Dom_x\right\}^*.$$

Given one of the actions

$$\langle set, insert(v)\rangle!$$

or

$$\langle set, delete(v)\rangle!,$$

the method *set* is invoked. If necessary, the set S is updated by inserting or deleting the element v and the corresponding event is announced. This in turn triggers *cnt*. B provides the necessary bindings for events that announce the update of the set, so that the counter can also be updated correspondingly.

Note that we do not assume that, for instance, the insertion and the increment occur simultaneously. Consequently, it is not the case that the size of the set is always equal to the counter. However, if every announced event has been consumed and "serviced" with the corresponding counter update, then we should have $|S| = C$. As we will see, this paper develops the theory necessary to formally express and prove this kind of property.

2.3 Trace-theoretic model

Before we can present the trace semantics of an II system, we need to show how a method and a system can be modeled as automata (labeled transition systems). We first describe how a single method is mapped to an automaton.

Definition 2.5 *Given a method* $m \equiv (V, E, P, S)$ *we define a* method automaton *as*

$$A_m \equiv (V, \Sigma, I, P, \delta)$$

where

- $\Sigma : V \to \bigcup_{x \in V} Dom_x$ *is the set of* states *of* m*, that is, mappings assigning values to the variables in* m*,*
- $I \subseteq \Sigma$ *is the set of* initial states *of the automaton* A_m*, that is, states in which the program counter of* m *points to the first statement of* m*, that is,* $pc = 1$*. Note that not every state in* I *has to satisfy* P*,*
- $\delta \subseteq \Sigma \times Act \times \Sigma$ *is the* transition relation *and is defined as the smallest relation satisfying*
  - $\{(s, a, s), (s, \tau, [s|x = v])\} \subseteq \delta$ *if there exists a statement*
    $$g \xrightarrow{a} x := exp$$
    in S *such that* g *is true in* s *and* exp *evaluates to* v *in* s*,*
  - $(s, \tau, s) \in \delta$ *if* g *is not true in* s*.* □

Given a state s over variables V_1 and a set of variables $V_2 \subseteq V_1$, let $s|V_2$ be the *projection* of s to V_2.

Definition 2.6 *Given method automata*

$$A_i \equiv (V_i, \Sigma_i, I_i, P_i, \delta_i)$$

for $1 \leq i \leq n$ *their* parallel composition *is given by*

$$A_1 \| \ldots \| A_n \equiv (V, \Sigma, I, P, \delta)$$

where

- $V = \bigcup_{i=1}^{n} V_i$,
- $\Sigma : V \to \bigcup_{x \in V} Dom_x$ *is the set of states over* V,
- $s \in I$ *iff* $s\lceil V_i \in I_i$ *for all* $1 \leq i \leq n$,
- $P = \bigwedge_{i=1}^{n} P_i$, *and*
- $\delta \subseteq \Sigma \times Act \times \Sigma$ *is the smallest relation satisfying*

1. $(s, \tau, s') \in \delta$ *if there exists* $1 \leq i \leq n$ *such that* $(s\lceil V_i, \tau, s'\lceil V_i) \in \delta_i$ *and all variables in* V *but not in* V_i *remain unchanged, that is,* $s\lceil(V - V_i) = s'\lceil(V - V_i)$, *and*
2. $(s, \tau, s) \in \delta$ *if there exist* $1 \leq i, j \leq n$ *such that* $i \neq j$ *and*

$$(s\lceil V_i, \langle m, e\rangle?, s\lceil V_i) \in \delta_i$$

and

$$(s\lceil V_j, \langle m, e\rangle!, s\lceil V_j) \in \delta_j,$$

and

3. $(s, \tau, [s|z = e]) \in \delta$ *if there exist* $1 \leq i, j \leq n$ *such that* $i \neq j$ *and*

$$(s\lceil V_i, \langle m, z\rangle?, s\lceil V_i) \in \delta_i$$

and

$$(s\lceil V_j, \langle m, e\rangle!, s\lceil V_j) \in \delta_j,$$

and

4. $(s, \tau, s) \in \delta$ *if there exist* $1 \leq i \leq n$, m *and* z *such that*

$$(s\lceil V_i, \langle m, z\rangle?, s\lceil V_i) \in \delta_i$$

and

$$(s\lceil V_j, \langle m, e\rangle!, s\lceil V_j) \notin \delta_j$$

for all e *and* $1 \leq j \leq n$ *with* $j \neq i$, *and*

5. $(s, \tau, s) \in \delta$ *if there exist* $1 \leq i \leq n$, m *and* e *such that*

$$(s\lceil V_i, \langle m, e\rangle?, s\lceil V_i) \in \delta_i$$

and

$$(s\lceil V_j, \langle m, e\rangle!, s\lceil V_j) \notin \delta_j$$

for all $1 \leq j \leq n$ *with* $j \neq i$, *and*

6. $(s, \tau, s) \in \delta$ *if there exists* $1 \leq i \leq n$ *such that*

$$(s\lceil V_i, \langle m, e\rangle!, s\lceil V_i) \in \delta_i$$

and

$$(s\lceil V_j, \langle m, z\rangle?, s\lceil V_j) \notin \delta_j$$

and

$$(s\lceil V_j, \langle m, e\rangle?, s\lceil V_j) \notin \delta_j$$

for all z *and* $1 \leq j \leq n$ *with* $j \neq i$. □

The intuition behind the definition of δ is as follows. The first clause covers the case where one of the components moves independently by executing an assignment for instance. The next two clauses model synchronous communication. While the second clause captures synchronization without a data exchange, the third clause defines communication with update of some variable z. The final three clauses allow a component to stutter if the environment does not offer a matching action. Note that only the communication case requires synchronization. In all other cases a component can move independently.

We are now ready to define the trace semantics.

Definition 2.7 *Let*

$$A \equiv (V, \Sigma, I, P, \delta)$$

be an automaton corresponding to some system $\mathcal{S}$. *A* trace α *of* A *is an infinite sequence of the form*

$$s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \ldots$$

where

- $s_0 \in I$,
- $s_0 \models P$, *and*
- $(s_i, \tau, s_{i+1}) \in \delta$ *for all* $i \geq 0$, *and*
- *every statement of* $\mathcal{S}$ *gets executed infinitely often along* α.

The set of all traces of A *is denoted by* $\mathcal{T}[\![A]\!]$. □

The traces of a set of methods are never considered in isolation, but always in the context of an environment.

Definition 2.8 *Let* $\mathcal{S}$ *be a system and let*

$$G \equiv \{m_1, \ldots, m_n\}$$

be a set of methods (including possibly the dispatcher) of $\mathcal{S}$. *Given an environment model Env, the automaton* $A_{G,Env}$ *modeling the behaviour of* G *in the environment Env, is given by the parallel composition of all method automatons* A_{m_i} *and the environment automaton* $A_{m_{Env}}$, *that is,*

$$A_{G,Env} \equiv A_{m_1} \| \ldots \| A_{m_n} \| A_{m_{Env}}.$$

The traces of G *in Env are the traces of* $A_{G,Env}$, *that is,* $\mathcal{T}[\![G, Env]\!] = \mathcal{T}[\![A_{G,Env}]\!]$. □

3 Specifying implicit invocation systems

To specify the ongoing behaviour of an II system, we use first-order linear time temporal logic without the next time operator **X**, denoted by LTL^{-X}.[3]

Definition 3.1 *Given some set* AP *of atomic propositions and assuming* $p \in AP$, *the set of* LTL^{-X} *formulas is inductively defined as:*

$$\phi ::= p \mid \neg\phi \mid \varphi_1 \wedge \varphi_2 \mid \forall x.\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

Other formulas can be introduced as abbreviations in the usual way: $\varphi_1 \vee \varphi_2$ *abbreviates* $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \Rightarrow \varphi_2$ *abbreviates* $\neg\varphi_1 \vee \varphi_2$, *true abbreviates* $p \vee \neg p$, *false abbreviates*

[3] Our model allows for arbitrary, but finite stuttering to be added between two transitions which renders the next time operator useless.

$\neg true$ *and* $\exists x.\varphi$ *abbreviates* $\neg\forall x.\neg\varphi$. *The temporal operator* $\mathbf{F}\phi$ *abbreviates* $true\ \mathbf{U}\ \phi$ *and* $\mathbf{G}\phi$ *abbreviates* $\neg\mathbf{F}\neg\phi$. *Given*

$$\alpha \equiv s_0 \xrightarrow{a_0} s_1 \ldots \xrightarrow{a_{i-1}} s_i \ldots,$$

let $\alpha[i]$ *denote the state* s_i. *Let* $\alpha[i..]$ *denote the infinite suffix* $s_i \xrightarrow{a_i} s_{i+1} \ldots$. *The satisfaction relation* $\models$ *of a* LTL^{-X} *formula with respect to a trace* α *is inductively defined over the structure of the formula.*

$$\begin{array}{ll}
\alpha \models p & \text{if } \alpha[0] \models p \\
\alpha \models \neg\varphi & \text{if not } \alpha \models \varphi \\
\alpha \models \varphi_1 \wedge \varphi_2 & \text{if } \alpha \models \varphi_1 \text{ and } \alpha \models \varphi_2 \\
\alpha \models \forall x.\varphi & \text{if } \alpha \models \varphi[v/x] \text{ for all } v \in Dom_x \\
\alpha \models \varphi_1 \mathbf{U} \varphi_2 & \text{if } \exists 0 \leq i.\alpha[i..] \models \varphi_2 \text{ and} \\
 & \alpha[j..] \models \varphi_1 \text{ for all } 0 \leq j < i.
\end{array}$$

□

Initial, terminated and quiescent states

Typically, events are used to maintain some kind of system invariant. However, just like loop invariants in sequential programming, they usually will not be preserved along *every* transition of the system. The following scenario seems typical for II systems: The execution of a statement in some method m_1 results in the violation of the invariant. m_1 will then announce an event which will trigger some other method m_2. The execution of m_2 will then eventually reestablish the invariant. Note that the invariant might be violated until m_2 has completed. The next definition presents three predicates *init*, *term*, and *quiescent* that allow us to single out certain states along a trace in which the invariant should hold.

Definition 3.2 *Let* α *be a trace of a set of methods* G *in some environment* Env *and let* s *be a state along* α.

1. *The proposition init holds in* s *iff it is an initial state of the automaton* $A_{G,Env}$, *that is, the program counter of all methods in* G *point to the first statement.*
2. *The proposition term holds in* s *iff* s *is a fixed point, that is,* α *does not exhibit any state changes after* s.
3. *If* G *contains the dispatcher, that is,* $disp \in G$, *then proposition quiescent holds in* s *iff it is an initial state of* $A_{G,Env}$ *and the pending events datastructure* D *is empty.* □

In Example 2.2, for instance, the system invariant is $|S| = C$, the size of the set S is equal to the value of the counter C. This invariant is not maintained along every transition. For instance, while an *ins* event is pending in the dispatcher, the counter will lag behind. Let α be a trace of method *set* in some environment *Env* and let s be a state along α. Then, if *init* holds in s, that is, the program counter of *set* points to the first statement of *set*, then the size of S in s is the number of $\langle disp, ins\rangle!$ actions issued so far minus the number of $\langle disp, del\rangle!$ actions issued so far. Also, we expect the counter to have caught up whenever all events have been delivered and the system is back in one of its initial states, that is, if s is quiescent. Note that every terminated state also is quiescent.

Properties of the behaviour of a set of methods G in some environment *Env* can be described using the following notion of specification.

Definition 3.3 *Given a set of methods* G *and an environment model* Env, *a* specification *is a 4-tuple*

$$\{p\}\ (G, Env)\ \{\varphi\}$$

where p *is the pre-condition given as a boolean expression, and* φ *is a* LTL^{-X} *formula. The specification*

$$\{p\}\ (G, Env)\ \{\varphi\}$$

is satisfied, if

$$\forall \alpha \in \mathcal{T}[\![G, Env]\!].\text{if } \alpha[0] \models p \text{ then } \alpha \models \varphi.$$

□

3.1 Event semantics

The key feature of II systems is that the notion of events allows for a temporal and spatial separation of the *cause* and the *effect* of certain designated state changes. For instance, consider a set of source and executable files. Suppose we want our II system to automatically maintain consistency of the executables with respect to the source files. The modification of one of the source files causes the editor to announce a *modified* event. Assuming that this event is bound to the compiler, the effect of this event will be the invocation of the compiler at some later point in time and in some possibly remote location. This kind of separation between cause and effect seems essential to the easy integration of loosely-coupled software components. However, it also makes formal reasoning about II systems very difficult.

We will now define causes and effects more formally. We say that an event e is *announced* by method m whenever it is passed to the dispatcher, that is, m executes **announce**(e). Remember that in this case m performs a transition labeled with $\langle disp, e\rangle!$. The cause of an event, *cause*(e) for short, characterizes the state change that gave rise to the announcement of e.

Definition 3.4 *cause*(e, m) *is the strongest* LTL^{-X} *formula* φ *that validates the specification*

$$\{true\}\ \left(\{m\}, \{\langle disp, e\rangle?\}\right)\ \{\varphi\}.$$

cause(e) *is*

$$cause(e) \equiv \bigvee_{m \in G} cause(e, m)$$

where G *is the set of all methods that announce* e. □

In the above definition m is run in an environment that can accept event e if it is addressed to the dispatcher, that is, it offers the action $\langle disp, e\rangle?$. Let α be a trace of m in that environment. Due to the restricted shape of the environment, the only communication that m can engage in along α is sending e to the dispatcher. Moreover, it can do so at most once. Due to the fairness assumption that every statement is executed infinitely often, m will thus announce e exactly once along α. Note that m can still perform an infinite number of internal τ-actions.

The effect of e, *effect*(e), describes the state change with which the rest of the system will react. An event invokes the methods it is bound to. Suppose e is bound to m, that is, $(e, m) \in B$. We say that an event e is *consumed* by m whenever m receives e from the dispatcher, that is, m executes **consume**(z) after which z is bound to e for some variable z. Remember that in this case m performs a transition labeled with $\langle m, z\rangle?$. Note that in contrast to the cause, *effect*(e) depends on the methods that e is bound to and thus on the binding. An unbound event will not have any effect.

Definition 3.5 *effect(e, m) describes the state change by m that the consumption of e will give rise to. Formally, effect(e, m) is the strongest LTL^{-X} formula φ such that*

$$\{true\}\ (\{m\}, \{\langle m, e\rangle!\})\ \{\varphi\}.$$

The effect of the event is then given by

$$effect(e) \equiv \bigwedge_{(e,m)\in B} effect(e, m).$$

Cause and effect of an event are referred to as its semantics. □

The intuition behind the definition of the effect is analogous to that of the cause. m is run in an environment that can send the event e to m once, that is, it offers the action $\langle m, e\rangle!$. Let α be a trace of m in that environment. The only communication that m can engage in along α is receiving e. Moreover, it can do so at most once. Due to the fairness assumption that every statement is executed infinitely often, m will thus consume e exactly once along α.

For instance, consider the set-counter example of Section 2.2. Whenever an element x is added to the set S with $x \notin S$, then the action $\langle disp, ins\rangle!$ announces the event *ins* by communicating it to the dispatcher. The consumption of *ins* subsequently causes the counter C to be incremented. Similarly for the event *del*. For specification purposes we need *logical variables*. A logical variable is never mentioned in a program and its value can thus be assumed to remain unchanged across program transitions.[4] Let T and w be logical variables. Also, let $follows(\varphi, \psi)$ abbreviate $\psi\ \mathbf{U}\ (\mathbf{G}\varphi)$. Informally, $follows(\varphi, \psi)$ holds for α if there exists a state s_i along α up to which ψ holds and from which φ holds forever. The reason for announcing *ins* is that there is some value $x \in Dom_x$ such that $x \notin S$ and the value of S changes from T to $T \cup \{x\}$ for some T. Note that only the method *set* announces *ins*.

$$\begin{array}{ll} & cause(ins) \\ = & cause(ins, set) \\ \Rightarrow & \forall T \in Dom_S.S = T \Rightarrow \exists x \in Dom_x.x \notin T \wedge \\ & \qquad follows(S = T \cup \{x\}, S = T) \end{array}$$

The effect of *ins* is an increment of C. Remember that *ins* is bound to *cnt*.

$$\begin{array}{ll} & effect(ins) \\ = & effect(ins, cnt) \\ \Rightarrow & \forall w \in Dom_C.C = w \Rightarrow follows(C = w + 1, C = w). \end{array}$$

Similarly, for the *del* event we get

$$\begin{array}{ll} & cause(del) \\ = & cause(del, set) \\ \Rightarrow & \forall T \in Dom_S.S = T \Rightarrow \exists x \in Dom_x.x \in T \wedge \\ & \qquad follows(S = T - \{x\}, S = T) \end{array}$$

and

$$\begin{array}{ll} & effect(del) \\ = & effect(del, cnt) \\ \Rightarrow & \forall w \in Dom_C.C = w \Rightarrow follows(C = w - 1, C = w). \end{array}$$

Note that in the above formalization the event semantics can only express state changes. More precisely, given an event e, neither the announcement nor the consumption of some other event can be part of the semantics of e. In other words, an event cannot cause the announcement of some other event, for instance.

[4] Sometimes also called *rigid variables*.

4 Verifying implicit invocation systems

Before we can introduce our verification methodology, we need to define the notion of independence.

Definition 4.1 *Let S be a system with methods M and environment model Env. Let G be a set of methods of S with environment model Env_G. We say that (G, Env_G) is independent with respect to p and φ, if*

$$\{p\}\ (G, Env_G)\ \{\varphi\}$$

implies

$$\{p\}\ (M, Env)\ \{\varphi\}.$$

□

Independence thus allows us to "lift" a specification from a subset of methods to the entire system. It attempts to reconcile concurrency and compositionality, which is a central problem in concurrency theory: Under what circumstances can a property of a composite system be obtained from properties of its components despite the presence of concurrency [dR85]? Unfortunately, our methodology crucially depends on our ability to prove independence. To ease this task, we will now isolate a few syntactic conditions that guarantee independence.

Let $\overline{G}$ be the environment (complement) of G, that is, the set of methods in M but not in G. First of all, we need to prevent the environment from interfering with the computation of G via shared variables. More precisely, we assume that G and $\overline{G}$ do not share any variables. Moreover, we need to prevent the environment from changing the truth value of either p or φ, that is, we require $\overline{G}$ to not mention any of the variables in p or φ. However, the absence of variable conflicts implied by the above two conditions is not sufficient. The reason is that an enlarged environment Env may offer communication actions that Env_G did not offer. These additional actions may allow G in Env to exhibit traces that were impossible for G in Env_G. We say that an environment model Env_G *complements* a set of methods G, if every action mentioned in G has a matching action in Env_G. Consequently, a complementing environment will allow G to engage in all communications it could be interested in.

We thus arrive at the following lemma.

Lemma 4.1 *Let $G \subseteq M$ be a non-empty set of methods and let $\overline{G}$ be the methods in M but not in G. (G, Env_G) is independent with respect to p and φ, if*

- *all methods in $\overline{G}$ do not mention any of the variables used in G, and*
- *all methods in $\overline{G}$ do not mention any of the variables used in p or φ, and*
- *Env_G complements G.* □

Let $M \equiv \{m_1, \ldots, m_n, disp\}$ be the set of methods of some system $\mathcal{S}$ with environment model Env. Suppose we want to show that

$$\{p\}\ (M, Env)\ \{\varphi\}.$$

Our verification methodology consists of the following three phases.

Decomposition Partition M into groups $G_1, \ldots, G_k$ with $1 \leq k \leq n$. Typically, the dispatcher is analyzed in isolation and forms a singleton group. For each group G_i find an environment model Env_i and subspecifications p_i and φ_i such that (G_i, Env_i) is independent with respect to p_i and φ_i.

Local reasoning Prove subspecifications

$$\{p_i\}\ (G_i, Env_i)\ \{\varphi_i\}$$

for each $1 \leq i \leq n$. Typically, this step uses both the event-method binding and the semantics of the events.

Global reasoning Lift the subspecifications to the entire system using independence, and prove

$$\{p\}\ (M, Env)\ \{\varphi\}.$$

4.1 Example: Sets and counters

As indicated at the end of Section 2.2, we would like to show that after an arbitrary but finite number of insert and delete events have been passed to the system, the size of the set is equal to the value of counter in every quiescent state. Formally,

$$\begin{array}{l}\{S = \emptyset \wedge C = 0\}\\ (M, Env)\\ \{\mathbf{G}(quiescent \Rightarrow |S| = C)\}\end{array}$$

where

$$Env \equiv \{\langle set, insert(v)\rangle!, \langle set, delete(v)\rangle! \mid v \in Dom_x\}^*.$$

4.1.1 Decomposition

Each method in $\mathcal{SC}$ forms a group. Independence will be shown later.

4.1.2 Local reasoning

Let $\#\langle m, e\rangle?$ stand for the number of times that event e was received by m so far along the current trace. Also, let $\#\langle m, e\rangle!$ stand for the number of times that event e was sent to m so far along the current trace. Formally, this operator can be implemented using auxiliary variables.

Due to our synchronous notion of communication, a communication action cannot occur without a matching action. We thus get the following lemma.

Lemma 4.2 *Along every trace α of some system S, the number of matching input and output actions must be equal, that is, we must have $\#\langle m, e\rangle? = \#\langle m, e\rangle!$.* □

Set method *set*

Given the *cause*(*ins*) and *cause*(*del*), we can see that whenever an element is added to the set, an *ins* event is announced and that whenever an element is removed from the set, a *del* event is announced. Thus, in initial states, the size of S is the number of *ins* events sent to the dispatcher so far minus the number of *del* events sent to the dispatcher so far. The validity of this correspondence is limited to initial states, because it does not hold when control is between updating the set and posting the appropriate event. Formally,

$$\begin{array}{l}\{S = \emptyset\}\\ (set, Env_{set})\\ \{\mathbf{G}(init \Rightarrow |S| = \#\langle disp, ins\rangle! - \#\langle disp, del\rangle!)\}\end{array}$$

where

$$Env_{set} \equiv \{\langle set, insert(v)\rangle!, \langle set, delete(v)\rangle! \mid v \in Dom_x\}^*.$$

Counter method *cnt*

The local specification of the counter is analogous. Given the *effect*(*ins*) and *effect*(*del*), we can see that whenever an *ins* event is consumed, the counter is incremented and that whenever an *del* event is consumed, the counter is decremented. Thus, in initial states, the value of C is the number of *ins* actions received from the dispatcher so far minus the number of *del* actions received from the dispatcher so far. Formally,

$$\begin{array}{l}\{C = 0\}\\ (cnt, Env_{cnt})\\ \{\mathbf{G}(init \Rightarrow C = \#\langle cnt, ins\rangle? - \#\langle cnt, del\rangle?)\}\end{array}$$

where $Env_{cnt} \equiv \{\langle cnt, ins\rangle!, \langle cnt, del\rangle!\}^*$.

Dispatcher method *disp*

Note that no assumptions about the binding B or the storage policy of the dispatcher have been made yet. For instance, we have not yet required B to be non-empty or the dispatcher not to lose every message. However, it is clear that for the verification to go through, certain minimal requirements have to be imposed. The following specification captures these requirements.

Every *ins* event input by the dispatcher is first stored in D and then passed on to the counter. Similarly for *del* events. In other words, the dispatcher must eventually pass on every *ins* and *del* event received. More precisely, in every initial state, the number of $\langle disp, ins\rangle?$ actions performed by the dispatcher is the sum of the number of $\langle cnt, ins\rangle!$ actions performed by *cnt* plus the number of *ins* events still pending in D. A similar correspondence holds for the *del* event. Formally,

$$\begin{array}{l}\{true\}\\ (disp, Env_{disp})\\ \{\mathbf{G}(init \Rightarrow\\ \quad (\#\langle disp, ins\rangle? = \#\langle cnt, ins\rangle! + \#(cnt, ins, D) \wedge\\ \quad \#\langle disp, del\rangle? = \#\langle cnt, del\rangle! + \#(cnt, del, D)))\}\end{array}$$

where

$$Env_{disp} \equiv \{\langle disp, ins\rangle!, \langle disp, del\rangle!, \langle cnt, ins\rangle?, \langle cnt, del\rangle?\}^*$$

and $\#(m, e, D)$ denotes the number of occurrences of the pair (m, e) in D. Note that the above specification would fail, if, for instance, the binding was empty, or the dispatcher simply discarded some of the incoming events.

4.1.3 Global reasoning

Note that *set*, *cnt* and *disp* do not share any variables and that Env_{set}, Env_{cnt} and Env_{disp} complement *set*, *cnt* and *disp* respectively. Due to Lemma 4.1, the three group and environment pairs above are independent with respect to their respective specifications. Thus,

$$\begin{array}{l}\{S = \emptyset\}\\ (M, Env)\\ \{\mathbf{G}(init \Rightarrow |S| = \#\langle disp, ins\rangle! - \#\langle disp, del\rangle!)\}\end{array}$$

and

$$\begin{array}{l}\{C = 0\}\\ (M, Env)\\ \{\mathbf{G}(init \Rightarrow C = \#\langle cnt, ins\rangle? - \#\langle cnt, del\rangle?)\}\end{array}$$

and

$$\{true\}\ (M, Env)\ \{\mathbf{G}(init \Rightarrow \#\langle disp, ins\rangle? = \#\langle cnt, ins\rangle! + \#(ins, cnt, D) \wedge \#\langle disp, del\rangle? = \#\langle cnt, del\rangle! + \#(del, cnt, D))\}.$$

Let α be a trace of (M, Env) that starts in a state satisfying $S = \emptyset \wedge C = 0$ and let s_i be a quiescent state along α. s_i satisfies the implication

$$init \Rightarrow \begin{array}{l} |S| = \#\langle disp, ins\rangle! - \#\langle disp, del\rangle! \wedge \\ C = \#\langle cnt, ins\rangle? - \#\langle cnt, del\rangle? \wedge \\ \#\langle disp, ins\rangle? = \#\langle cnt, ins\rangle! + \#(ins, cnt, D) \wedge \\ \#\langle disp, del\rangle? = \#\langle cnt, del\rangle! + \#(del, cnt, D). \end{array}$$

Moreover, quiescence implies *init* and *empty*(D) which implies that the number of (cnt, ins) and (cnt, del) pairs in D is zero, that is,

$$\#(ins, cnt, D) = \#(del, cnt, D) = 0.$$

Thus, s_i satisfies

$$\begin{array}{l} |S| = \#\langle disp, ins\rangle! - \#\langle disp, del\rangle! \wedge \\ C = \#\langle cnt, ins\rangle? - \#\langle cnt, del\rangle? \wedge \\ \#\langle disp, ins\rangle? = \#\langle cnt, ins\rangle! \wedge \\ \#\langle disp, del\rangle? = \#\langle cnt, del\rangle!. \end{array}$$

Using Lemma 4.2 we get

$$\#\langle disp, ins\rangle? = \#\langle disp, ins\rangle!$$

and

$$\#\langle disp, del\rangle? = \#\langle disp, del\rangle!.$$

Consequently, $s_i \models |S| = C$ which allows us to conclude

$$\{S = \emptyset \wedge C = 0\}\ (M, Env)\ \{\mathbf{G}(quiescent \Rightarrow |S| = C)\}.$$

4.2 Example: File system

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 the files in *Src* correspond to an executable file *exe* and that **make**(*Src*, *exe*) creates a new executable with respect to the current contents of *Src*. In the following, the variable f will range over files in $\mathcal{FS}$, that is, $Dom_f = \{v \mid v \text{ is a file in } \mathcal{FS}\}$. The system $\mathcal{FS}$ contains the events

$$E \equiv \{modified\} \cup \{ed(v) \mid v \in Dom_f\},$$

and the methods

$$M \equiv \{edit, cmpl, disp_B\}$$

where

$$B \equiv \{(modified, cmpl)\}.$$

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 $f \in Src$,

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

The *modified* event gets announced, whenever the file system is not fresh. Moreover, whenever the *modified* event is consumed the file system will eventually be fresh. The semantics of the *modified* event thus is

$$\begin{array}{rcl} cause(modified) & \Rightarrow & \mathbf{FG}\neg fresh \\ effect(modified) & \Rightarrow & \mathbf{FG}fresh. \end{array}$$

The methods are given in Figure 5. An $ed(v)$ event trig-

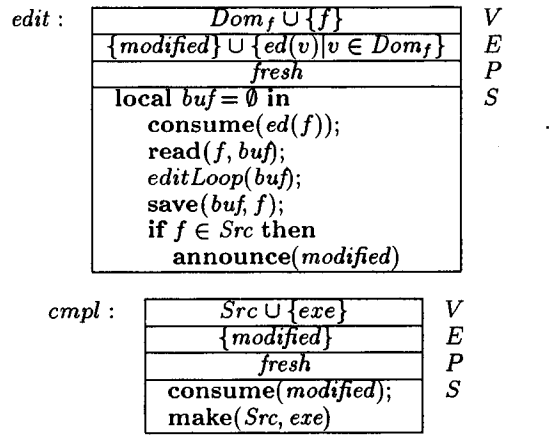

Figure 5: The methods *edit* and *cmpl*

gers the *edit* method. Method *edit* copies the contents of v into a local buffer *buf* and at the end of the edit session, v is updated with *buf*. If v also is a source file relevant to *exe*, the *modified* event is announced. The *modified* event triggers the compile method *cmpl* which updates the executable. We would like to show that after a finite but arbitrary sequence of $ed(v)$ events the file system will always be fresh upon termination. Formally,

$$\{fresh\}\ (M, Env)\ \{\mathbf{G}(term \Rightarrow fresh)\}$$

where $Env \equiv \{\langle edit, ed(v)\rangle! \mid v \in Dom_f\}^*$.

4.2.1 Decomposition

Like in the set-counter example, each method forms a group. An independence argument is given later.

4.2.2 Local reasoning

We abuse notation slightly and use an input or output action a also as an atomic proposition. A state s along some trace α satisfies $\langle m, e\rangle?$ if e has just been received by m. Also, s satisfies $\langle m, e\rangle!$ if e has just been sent to m.

Edit method *edit*

The fact that one of the source files in *Src* is to be edited, is abbreviated by *update*(*Src*), that is,

$$update(Src) \equiv \exists f \in Src.\langle edit, ed(f)\rangle?.$$

We will also need a *weak until operator* $\varphi \mathbf{U}_w \psi$ which expresses that either φ holds forever or at least until ψ holds, that is,

$$\varphi \mathbf{U}_w \psi \equiv \mathbf{G}\varphi \vee (\varphi\ \mathbf{U}\ \psi).$$

Whenever the executable is fresh, it will either remain so forever or until a source file is edited, that is, *update(Src)* holds.

$$\{fresh\}\ (edit, Env_{edit})\ \{\mathbf{G}(fresh \Rightarrow (fresh\ \mathbf{U}_w\ update(Src)))\} \quad (1)$$

where $Env_{edit} \equiv \{\langle edit, ed(v)\rangle! \mid v \in Dom_f\}^*$. Also, every update eventually leads to the *modified* event being announced.

$$\{true\}\ (edit, Env_{edit})\ \{\mathbf{G}(update(Src) \Rightarrow \mathbf{F}\langle disp, modified\rangle!)\} \quad (2)$$

This step uses *cause(modified)*.

Compiler method *cmpl*

The receipt of a modified event triggers recompilation and thus eventually creates a fresh executable. The semantics *modified* allows us to conclude that the file system eventually stays fresh forever.

$$\{true\}\ (cmpl, Env_{cmpl})\ \{\mathbf{G}(\langle cmpl, modified\rangle? \Rightarrow \mathbf{FG}fresh)\} \quad (3)$$

where $Env_{cmpl} \equiv \{\langle cmpl, modified\rangle!\}^*$. This step uses the effect of *modified*. The above specification is too strong for our purposes, because it cannot be lifted to the entire system. We thus employ the following weaker specification.

$$\{true\}\ (cmpl, Env_{cmpl})\ \{\mathbf{G}(\langle cmpl, modified\rangle? \Rightarrow \mathbf{F}fresh)\} \quad (4)$$

Dispatcher method *disp*

The requirements for the binding and storage policy are as follows. An arriving *modified* event eventually leads to a pending event $(cmpl, modified)$ being stored in D.

$$\{true\}\ (disp, Env_{disp})\ \{\mathbf{G}(\langle disp, modified\rangle? \Rightarrow \mathbf{F}(cmpl, modified) \in D)\}$$

where $Env_{disp} \equiv \{\langle disp, modified\rangle!, \langle cmpl, modified\rangle?\}^*$. An event pending in D eventually is delivered.

$$\{true\}\ (disp, Env_{disp})\ \{\mathbf{G}((cmpl, modified) \in D \Rightarrow \mathbf{F}\langle cmpl, modified\rangle!)\}.$$

This implies

$$\{true\}\ (disp, Env_{disp})\ \{\mathbf{G}(\langle disp, modified\rangle? \Rightarrow \mathbf{F}\langle cmpl, modified\rangle!)\}. \quad (5)$$

Note that in contrast to the set-counter example, the dispatcher now is allowed to lose some (but not all) incoming events. More precisely, suppose a non-empty sequence of $\langle disp, modified\rangle!$ actions are passed to the dispatcher. Then, only at least one $\langle cmpl, modified\rangle!$ action needs to be passed to the compiler.

4.2.3 Global reasoning

In contrast to the set-counter example, the $\mathcal{FS}$ system contains two methods (*edit* and *cmpl*) that share variables (files). Obviously, this complicates the verification since Lemma 4.1 cannot be applied as readily. However, since the dispatcher does not share any variables with *edit* or *cmpl*, Lemma 4.1 can still be used to lift (5), the local specification of the dispatcher. Moreover, the sharing is limited enough such that the remaining specifications can still be lifted. $(edit, Env_{edit})$ is independent with respect to the specification (1) because the environment (the compiler and the dispatcher) can never change the value of *fresh* from true to false (only from false to true) nor can it change the value of *update(Src)*. Also, $(edit, Env_{edit})$ is independent with respect to the specification (2) because the environment (the compiler and the dispatcher) cannot prevent *edit* from eventually announcing *modified*. Moreover, $(cmpl, Env_{cmpl})$ is independent with respect to (4), because the environment (the editor and the dispatcher) cannot prevent the compiler from creating a fresh executable once it has received a *modified* event. Note, however, that the environment can prevent an executable from staying fresh forever and thus the original specification (3) cannot be lifted.

Using the lifted versions of (2), (5), and (4) we get

$$\{fresh\}\ (M, Env)\ \{\mathbf{G}(update(Src) \Rightarrow \mathbf{F}fresh)\}. \quad (6)$$

Let α be a trace of (M, Env) that starts in a state satisfying *fresh*. There are two cases.

Case 1: No state along α satisfies *update(Src)*. Then, the executable is always fresh and thus

$$\alpha \models \mathbf{G}(term \Rightarrow fresh).$$

Case 2: There is at least one state along α that satisfies *update(Src)*. Since the environment *Env* issues only a finite number of $ed(f)$ events, there must be a state s_i that is the last such state, that is,

$$\forall j.i < j.\neg update(Src).$$

By (6), there exists $k \geq i$ such that $\alpha[k] \models fresh$. Since there are no more updates after s_i, we also have with (1),

$$\alpha[k..] \models \mathbf{G}fresh.$$

Thus, every terminated state along α must also be fresh.

Thus,

$$\{fresh\}\ (M, Env)\ \{\mathbf{G}(term \Rightarrow fresh)\}.$$

5 Conclusion and future work

We have presented a formal framework for reasoning about implicit invocation systems. The framework rests on a formal semantics that combines standard notions from process algebra and trace semantics. It formally captures the cause and the effect of an event and thus offers a useful abstraction mechanism and reasoning tool. A three-phase verification methodology supporting linear time temporal logic properties is presented. In the decomposition phase the entire system is partitioned into groups of components and for each group a suitable subspecification is found. In the local reasoning phase, each group is verified with respect to its respective subspecification. The global reasoning phase lifts the local properties to the entire system and uses them to show the overall specification. The notion of independence ensures soundness of this step.

Future work

The weakness of this work clearly lies in decomposition phase. Little support is offered for partitioning the system into suitable groups, finding subspecifications for them and proving independence. Future work will attempt to identify more heuristics and sufficient conditions to aid this phase. Compositionality is achieved through independence. In the presence of concurrency, however, compositionality has proven to be a difficult goal which most of concurrency theory has been concerned with for a long time [dR85]. Hopefully, we will be able to make use of the existing work here.

While the present paper is aimed at a rather general modeling of II systems, an approach to find support for verification is to analyze existing II systems and to distill constraints which can safely be imposed on the construction of II systems without overly compromising expressiveness [BG99]. For instance, the examples used in this paper seem to be representative of two important classes of operations.

- The first class is probably best described as *reset* or *update* operations. An operation falls into this class if it establishes its postcondition from any initial state and in any environment. An example is the make operation of the file system example. Another example is the update operation on multiple (possibly distributed) views in the model-view-controller paradigm [KP88, GHJV95].
- The second class is characterized as follows. Suppose two operations f and g act on disjoint sets of variables V_f and V_g respectively. Suppose the invariant I expresses some kind of relationship between the values of V_f and V_g that behaves as follows. A single application of either f or g leaves I violated. However, the application of the second, corresponding operation (g or f) reestablishes I. Consider the set-counter example, for instance. The two operations are the insert operation $S := S \cup \{x\}$ and the increment operation $C := C + 1$.

As we have seen, both, the independence of operations from initial states and environment interference on the one hand, and the disjointness of variables on the other, can greatly aid the verification process. More work needs to be done to identify more classes of operations and investigate how the inherent constraints can support the verification. Ideally, this would lead to lemmas and proof rules that would make the global reasoning phase more mechanic.

Moreover, the size and complexity of the independent groups that arise during the decomposition phase determine the tractability of the methodology for large-scale systems. In general, there seems to be a tradeoff between the size of a group and the ease of proving its independence. Large groups are more likely to be independent, but also tend to be more complex. However, we believe that the loosely-coupled nature of II systems naturally supports the formation of small independent groups. More experience on large-scale examples is needed before we can support this claim more formally.

We also intend to investigate the hierarchical (or recursive) use of our methodology. This would allow us to view an entire system as a component of yet another system and would thus allow for the development of a stepwise refinement strategy. Previous work on refinement for UNITY (e.g., [CM88, San90, Din97]) may be helpful here.

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.

[BG99] A. Berry and D. Garlan. Making architectural analysis reasonable. In *Proceedings of First Working IFIP Conference on Software Architeture (WICSA1)*, February 1999. To appear.

[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.

[Cor91] The Common Object Request Broker: Architecture and specification. OMG Document Number 91.12.1, December 1991. Revision 1.1 (Draft 10).

[DGJN98] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Towards a formal treatment of implicit invocation using rely/guarantee reasoning. *Formal Aspects of Computing*, 1998. To appear.

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

[Din97] J. Dingel. Approximating UNITY. In *Second International Conference on Coordination Models and Languages*, LNCS 1282, pages 320–337. Springer Verlag, September 1997.

[dR85] W.P. de Roever. The quest for compositionality — a survey of assertion-based proof systems for concurrent programs. Part I: Concurrency based on shared variables. In E.J. Neuhold and G. Chroust, editors, *Formal Methods in Programming.* IFIP, Elsevier Science Publishers, 1985.

[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.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design.* Addison-Wesley, 1995.

[GKN92] D. Garlan, G.E. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *IEEE Computer*, 25(6), June 1992.

[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.

[Gol84] A. Goldberg. *Smalltalk-80 — The Interactive Programming Environment.* Addison-Wesley, Reading, MA, 1984.

[GZ97] D. Gelernter and L. Zuck. On what linda is: Formal description of Linda as a reactive system. In *Second International Conference on Coordination Models and Languages*, LNCS 1282, pages 187–204. Springer Verlag, September 1997.

[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.

[KP88] G.E. Krasner and S.T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, August/September 1988.

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

[San90] B.A. Sanders. Stepwise refinement of mixed specifications of concurrent programs. In M. Broy and C.B. Jones, editors, *Proceedings of IFIP Working Conference on Programming and Methods*, pages 1–25. Elsevier Science Publishers (North Holland), May 1990.

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

[Sun93] SunSoft. *Tooltalk 1.1.1 Users's Guide*, November 1993.