DISSERTATION

# A Systematic Approach to the Development of Event-Based Applications

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften

unter der Leitung von

o.Univ.-Prof. Dipl.-Ing. Dr.techn. Mehdi Jazayeri
Institut für Informationssysteme
Abteilung für Verteilte Systeme

eingereicht an der

Technischen Universität Wien
Fakultät für Technische Naturwissenschaften und Informatik

von

**Dipl.-Ing. Pascal Fenkam**

pfenkam@infosys.tuwien.ac.at

Matrikelnummer: 9730024
Schönbrunnerstrasse 223/11
A-1120 Wien, Österreich

Wien, im Oktober 2003

# Kurzfassung

Software bestimmt heute wesentliche Dienste der Informationsgesellschaft: Banken, Geschäftswelt oder Transportwesen sind Beispiele von Applikationen, die auf Software angewiesen sind. Solche Software wird immer mehr auf geographisch getrennte Rechner verteilt und von Benutzern mit einer wachsenden Anzahl von differenzierten Endgeräten verwendet (von leistungsstarken Desktop-Rechnern bis hin zu Mobiltelefonen). Software Technologie wird damit herausgefordert, die komplexen Anforderungen an solche Applikationen zu erfüllen.

Aber die heutige Software Technologie wurde für Applikationen entworfen, um von relativ wenigen homogenen Endgeräten verwendet zu werden. Derzeitige Applikationen behandeln damit keine Tausenden, in der Zukunft vielleicht Millionen von heterogenen Endgeräten. Daher befasst sich ein Teil der Forschung im Bereich verteilter Systeme aktiv mit dem Entwurf von Software Methodologien für Kommunikationstechnologie, Werkzeuge, Mechanismen und Techniken, um die Einschränkungen heutiger Sprachen und Techniken zu überwinden und somit die Anforderungen neu entstehender Rechnerumgebungen und Applikationen zu erfüllen.

Eine der viel versprechenden Techniken dafür ist das Ereignisgesteuerte Paradigma. Wesentlicher Vorteil dieses Paradigmas ist die Unterstützung von lose gekoppelten Komponenten, die gemeinsam eine Applikation definieren und daher für eine große Anzahl von heterogenen Komponenten skaliert. Die Anwendung des Ereignisgesteuerten, architekturellen Stils wurde bereits erfolgreich zur Entwicklung von komplexen Systemen in großem Umfang gezeigt. Es wurde rasch in Forschungsprototypen und auch kommerziellen Produkten und Werkzeugen eingebaut. Die Praxis der Applikationsentwicklung ist jedoch noch immer ad-hoc und informell. Es ist daher schwierig, über die Korrektheit solcher Applikationen zu schlussfolgern. Der Mangel an Systematik und rigoroser Basis für die Entwicklung Ereignisgesteuerter Systeme ist zu einem ernsten Problem geworden, speziell da das Ereignisgesteuerte Paradigma in bedeutenden Anwendungsbereichen wie Flugsicherung, E-Commerce, Fahrzeugtechnik oder Haushaltselektronik Einzug gehalten hat. Existierende Theorien können derzeit aber noch nicht für die Entwicklung solcher komplexen und korrekten Systeme eingesetzt werden.

Diese Dissertation entwickelt einen neuen Ansatz (LECAP) für die Erstellung korrekter, komplexer Ereignisgesteuerter Applikationen. Das LECAP Framework schafft dafür eine Methodologie für die Spezifikation, die schrittweise Entwicklung und die Verifikation von

Ereignisgesteuerten Systemen. Diese Methodologie soll dann eine praktische Anwendbarkeit in allen neu entstehenden Anwendungsbereichen haben.

# ACKNOWLEDGMENTS

I would like to express my deep gratitude to all people who have supported me throughout my research.

In particular, I offer my sincerest gratitude to my supervisor, Mehdi Jazayeri. You introduced me to research and kept me on the right track with your advice while giving me the freedom and the possibility to pursue my research ideas.

It is a pleasure to acknowledge the great collaboration with the MOTION team at the Distributed Systems Group of TU Vienna: Harald Gall (our Dad!), Engin Kirda, and Gerald Reif. The nights in the train to Milano as well as the debugging session in the Zefiro hotel (disregarding our dinner with Georgio Armani) and the demos at the European commission headquarter are unforgettable events. The Distributed Systems Group deserves my gratitude for the enjoyable working environment. This thesis would not have been possible without your technical and administrative support: organizing trips, ordering new books, fixing IMAP problems, installing new security patches, troubleshooting the NFS server, etc.

Many thanks are due to Ketil Stølen for the valuable discussions concerning the adaption of his work to my work. My external committee member Cliff Jones deserves thanks for carefully reading my thesis and providing valuable insights and feedback.

This project thesis was supported by the European Commission in the Framework of the IST Program, projects MOTION (MObile Teamwork Infrastructure for Organizations Networking) and EASYCOMP (Easy Composition in Future Generation Component Systems).

I am indebted to my girl-friend Irmtraud Hutfless who accompanied me on the long way leading to this dissertation, accepting me working (too) long hours.

Finally, I would like to thank my family and friends for the perfect study environment and the unfailing and great support of all my endeavors regardless of the destination, the duration, or the sacrifices involved.

# CONTENTS

# Abstract

Today's software technology was created for applications that used a relatively small number of homogeneous devices. Current applications need to deal with thousands, and in the future perhaps millions, of highly heterogeneous devices. As a result, an active area of research in distributed systems is currently trying to invent software methodologies consisting of communication paradigms, tools, mechanisms, and techniques that overcome the limitations of current languages and techniques. One of the promising techniques for this purpose is the event-based communication paradigm (also called implicit invocation or publish/subscribe)

The primary benefit of the event-based paradigm is that it supports the loose coupling of components that compose an application and therefore scales to large numbers of heterogeneous components. The use of the event-based architectural style has been successfully demonstrated in the development of large-scale and complex systems. It has therefore been rapidly incorporated in not only research prototypes but also commercial products and toolkits and even in software communication standards. The practice of application development based on this paradigm is, however, ad hoc and informal. As a result, it is often difficult to reason about the correctness of the resulting applications. The lack of a systematic and rigorous basis for the development of event-based systems has become a serious problem as the event-based paradigm is being used increasingly in important domains such as flight-control, e-commerce, automotive, and home applications. The existing theory of specifying and verifying such applications cannot be applied for the development of large-scale and complex systems.

This thesis proposes a novel approach (LECAP) for the construction of correct event-based applications. The LECAP framework includes a methodology for the specification, stepwise development, and verification of event-based applications. The approach is compositional, hence, intrisically oriented towards the construction of complex software systems. The methodology will have practical application in all emerging application domains that exploit the event-based paradigm. Such domains include pervasive computing, telecommunications, electronic commerce, and Internet-based applications.

# List of Figures

# Chapter 1

# Introduction

This thesis develops a novel formal methodology called LECAP for the construction of an important class of software systems, namely event based ones. The approach includes techniques for the formal specification of event-based components, for the composition of these specifications and for the stepwise development of components. LECAP is based on Jones's rely/guarantee technique for the stepwise development of concurrent systems. In this chapter, we introduce the event-based paradigm, motivate its importance in the development of emerging applications and summarize our approach.

## 1.1 Background

The dependability of software systems is increasingly required as they are used for controlling a substantial part of the machinery surrounding us. This machinery can be found in many domains: household appliances (e.g. refrigerators, washing machines), mobile phones, traffic control (including air and railway), automotive, aeronautic, electronic commerce, and medical equipment.

Most of these applications are constructed using today's software technology [61, 32] which was proposed for building applications that have a relatively small number of homogeneous devices. In this methodology, when developing an application, the developer has (or is supposed to have) a relatively complete knowledge of the components of his application. Applications constructed using this methodology are typically strongly coupled, that is, their parts heavily depend on each other; modifying one component results in non trivial modifications in the remainder of the system. Further, these components are homogeneous in the sense that they must be written in the same language. In some cases, it is even required that the same compiler be used for the components of an application. Although the dependability of such systems is easier to ensure (compared to loosely coupled systems), they have some shortcomings that hinder from applying them to emerging applications.

Emerging applications need to deal with thousands, and in the future perhaps millions, of highly heterogeneous devices. Such applications must, not only scale to this huge amount of

devices, but they must also foresee the evolution of these devices as well as the integration of completely new devices. As a result, an active area of research in distributed systems is currently trying to invent software methodologies consisting of communication paradigms, tools, mechanisms, and techniques that overcome the limitations of current practices. The development of such complex software systems requires well established approaches that guarantee robustness of the product, economy of the development process, and rapid time to market [24]. One of the promising techniques for this purpose is the event-based communication paradigm (also called implicit invocation or publish/subscribe) whose adequacy was already demonstrated in the development of large and complex systems.

## 1.2 The Event Based Architectural Style

The event-based architectural style is a communication paradigm where components talk to each other by publishing and consuming events.

At the abstract level, the event-based paradigm includes publishers, subscribers, events, subscriptions, and an event-based infrastructure. The system allows publishers to announce information that are dispatched to interested components. A piece of information announced by a publisher is called an event. On the other hand, a subscriber is a component that is interested in receiving some events. The interest in receiving some information is formulated by means of subscriptions. The event-based infrastructure is responsible for matching subscriptions to events and for informing subscribed components on the occurrence of the matched events.

At the concrete level, event-based systems may vary depending on their architectures and their uses. Architectures of event-based systems vary from client/server to peer-to-peer. Similarly, there exist different uses of the event-based-paradigm e.g. middleware [23, 97, 125, 21, 4, 31, 83] integration frameworks [17, 111, 123, 131], programming languages, and operating systems [6]. In each of these systems, the event-based paradigm takes a different connotation and its abstract concepts (subscriptions, events, subscribers, publishers, event-based infrastructure) also take different names such as dispatchers, notifications, publications, broadcasters, receivers, etc. Figure 1.1 shows an example of event-based system in which the matching task is distributed among three computers. The many faces of the the event-based paradigm are discussed in [40].

The main advantage of the event-based paradigm is the loose coupling of components. A component that announces an event does not need to know which components will receive this information. It does not even need to know if there is a component that is interested in the event it announces. Finally, it does not need to be connected to the communication network, or even be active, at the same time as the subscribers or other publishers. The different components of a system can thus work independently of each

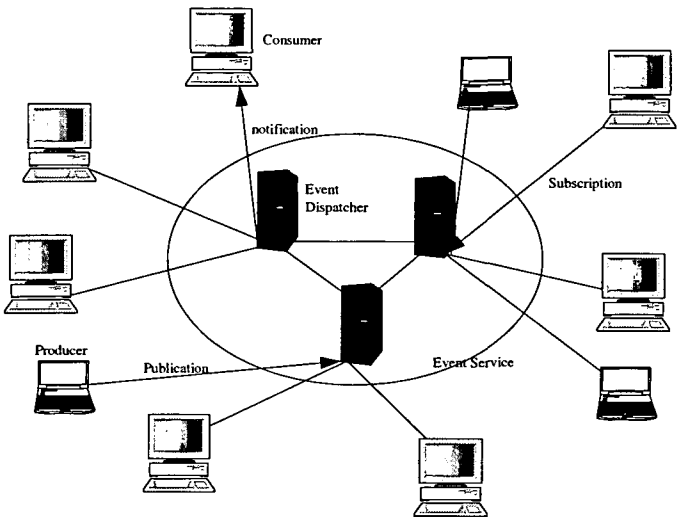


Figure 1.1: A Distributed Event Based Architecture

other. In particular, replacing a component does not require changing other components. This implies that applications can, for instance, be monitored without need for modifying their behaviors. Also, from a theoretical point of view, other communicating mechanisms such as broadcasting, multicasting and point-to-point communication can be encoded in the event-based paradigm, while this is not true in the other direction. This loose coupling of components makes this paradigm an intriguing candidate for many areas of distributed computing: mobile computing, pervasive computing, collaborative applications, peer-to-peer systems, distributed workflow systems, etc.

## 1.3 General Motivation

The importance of the event-based paradigm is witnessed by the increasing number of domains and tools in which it is deployed. Examples of such domains/tools are programming environments (e.g. Smalltalk), operating systems (e.g. AppleEvents [6]), communication middleware (e.g. Corba [97], Siena [23], JEDI [31], Gryphon [4], Spear [21], Elvin [125], PeerWare [101]), integration frameworks (e.g. OLE [17], JavaBeans [111], FIELD [110], SunSoft [124], Polylith [108], ISIS [15], Yeast [83]), and message oriented middleware (e.g. TIB/Rendezvous [131]). In this section, we overview five emerging application areas for the event-based architectural style. The intend of this description is to give a flavor of the diversity of this style.

### 1.3.1 Event Based Systems for Wireless Ad Hoc Networks

It has been exhaustively argued that the event-based architectural style is the most prevalent concept for the development of middleware for mobile computing [30, 24]. STEAM (Scalable Timed Events and Mobility) [88] is one of the middleware that explicitly take the notion of mobility in event-based middleware into consideration. Other middleware assume an existing communication infrastructure which does not always exist in wireless ad hoc network. It is argued by [88] that the event-based architectural style is particularly well suited to wireless ad hoc networking environments where communication relationships among application components are established very dynamically during the lifetime of the components.

The main issues addressed by STEAM is the dynamic nature of the network. Components establish direct communication relationship with any other application component without having to channel the transmission through an access point. This allows these components to communicate in a spontaneous manner in the absence of conventional fixed networks.

STEAM addresses this issue for the special case of systems where subscribers are only interested in events announced by components in their vicinity. The solution adopted is to limit event forwarding to the area surrounding the producers. This solution reduces the susceptibility of an event system to radio frequency interference.

### 1.3.2 Internet Based Distributed Application

OPELIX [48](OPen ELectronic Information Commerce System) is an Internet based platform for information commerce that was constructed in the European project with the same name. The aim of the project was to develop an open and scalable architecture for eCommerce on the Internet. The event-based architectural style is the core of the OPELIX prototype whose architecture includes two kinds of computing devices: broadcasters and receivers. The push system has a broadcaster that can be fed via a flexible information source interface. The broadcaster is in charge of controlling and scheduling the information dissemination process. To make the broadcasting process scalable, the system uses a hybrid protocol that is realized by the transport system implemented by the JEDI event-based middleware [31]. A receiver has two main components: channel access and user interface. It provides an interface that facilitates the interaction between users and channels. The event-based infrastructure promotes the extensibility and scalability of the system.

### 1.3.3 Event Based Distributed Workflow

A key requirement for enterprise business process optimization is the capability to encode these business processes as workflow specifications. For the purpose of defining such specifications and implementing their functionalities, workflow management systems (WFMS) are required. Such WFMS need to support the integration of various information sources to be useful to enterprises. One of the key problems that developers of WFMS must, hence, face is to develop them such that third party heterogeneous information producers can be integrated trivially. Other issues for emerging WFMS include an effective solution to problems related to the representation, control, and coordination of process entities that are geographically distributed across organizational entities.

As the event-based architectural style is the most prevalent approach to loose coupling, more and more distributed WFMS are developed based on it. Examples of such WFMS are described in [69, 20, 22]. These approaches are however based on a client/server oriented approach that renders distribution, openness, and scalability hard to achieve [58]. The use of highly distributed event-based middleware helps addressing these issues in EVE [58, 132].

### 1.3.4 Pervasive and Ubiquitous Computing

Ubiquitous computing is the method of enhancing computer uses by making many computers available throughout the physical environment, but making them effectively invisible to the users [133]. Computers may be interconnected and embedded in a wide range of appliances ranging in size from door locks to vehicle controllers performing tasks, such as automatically opening doors and routing vehicles to their human users [88]. An increasing number of research prototypes are being developed in this area. An example of such applications is the RFID Chef [84] whose intent is to facilitate the task of supply keeping and cooking.

In the RFID Chef project, various kitchen objects are tagged with RFID tags that, hence, allow tracing their movements and those of the cooks. A tagged element in RFID Chef is called a smart thing. The event-based architectural style is of prime importance to the RFID Chef system which proposes a model for supporting the large variety of interaction styles that are possible among the multitude of artifacts in a kitchen. A layered event infrastructure is used to provide the expected level of flexibility and efficiency.

These examples illustrate the various application possibilities of the event-based paradigm. In fact, there exists countless software systems that are based on this concept. This should be motivating enough for the development of a methodology for supporting the construction of such applications. Yet, ones own experiences seem to be more relevant, more motivating, and more instructive. We have attempted to construct a reliable non-trivial event-based

application for mobile computing called MOTION.

### 1.3.5 Mobile Collaboration

The MOTION project [79, 80, 81, 82, 109] ( MObile Teamwork Infrastructure for Organizations Networking) was supported by the European Commission in the Framework V of the IST Program. The aim of the project was to develop a highly scalable and distributed service platform that supports organizations whose employees are distributed across the globe and are frequently traveling. In fact, these nomadic and mobile employees use a wide range of computing devices such as PDAs, notebooks, and desktop computers. We have developed a service architecture that supports mobile teamwork by providing multi-device service access, metadata for information sharing and locating, and a query language (XQL) for distributed searches and event subscription.

The event-based paradigm is at the heart of the prototype we have developed. Distributed searches, peer-to-peer file sharing, loose coupling of components, user awareness, were all shown to be supported by the event-based paradigm. To ensure the dependability of our system, the key components of the MOTION platform were formally specified, the specifications validated, and the Java implementation systematically tested against the formal specification. Despite this, there are still some important deficiencies. In fact, the components were validated and verified in a client/server architecture where they were shown to be robust. In a loosely coupled environment (based on the event-based paradigm), the components were not robust anymore. The investigation of their deficiencies revealed serious challenges in the construction of event-based applications: there is neither a methodology for building such applications, nor suitable techniques for verifying or testing them. We give a detailed description of the MOTION prototype in Chapter 2.

## 1.4 Problem Statement

The use of the event-based architectural style has been successfully demonstrated in the development of large-scale and complex systems. It has, therefore, been rapidly incorporated in not only research prototypes but also commercial products and toolkits and even in software communication standards. The practice of application development based on this paradigm is, however, ad hoc and informal. As a result, it is often difficult to reason about the correctness of the resulting applications. The lack of a systematic and rigorous basis for the development of applications based on the event-based paradigm has become a serious problem as the event-based paradigm is being used increasingly in important domains such as flight-control, e-commerce, automotive, and home applications. The existing theory of specifying and verifying such applications cannot be applied for the development

of large-scale and complex systems. The development of complex systems is demanding well founded methodologies that guarantee robustness of products and economy of the development process [24].

## 1.5 Requirements for a Methodology

We present the requirements for a methodology for developing correct event-based applications. To elicit our requirements for a software engineering methodology for emerging systems, we agree with Pankaj Jalote [73] that a software engineering methodology must allow development of software systems that "scale up for large systems and that can be used to consistently produce high-quality software at low cost and with a small cycle time". This requirement exhibits at least three issues:

1. **Scalability Issue.** The development of large software systems does not obey the same rules as the development of small applications. In particular, methods used for developing small applications can not be applied to the development of large applications. Small applications can typically be controlled by a few persons while large applications are typically constructed by different teams. Various criteria come into consideration for the development of large scale systems that cannot be solved without proper methodologies.

2. **Quality Issue.** There is no doubt anymore that software systems need to be dependable. For building dependable applications, engineers, however, need methodologies that indeed allow developing such applications. There is increasing evidences about the poor quality of software systems. Neumann [93] reports about a multitude of such cases; for instance, DC Metro system incapable of labeling rerouted trains; computers shutting down aircraft engines in flight, collapse of air traffic control computers, bank web sites hacked, etc.

3. **Cost and Schedule Issue.** Anecdotal evidences also abound about software projects costs and schedule over-runs. The U.S. Air Force command-and-control software project initial cost estimate was $\$400,000$. Subsequently, the cost was renegotiated to $\$700,000$, to $\$2,500,000$ and finally to $\$3,200,000$. A case is also reported about a product that needed to be developed in 9 months at the cost of $\$250,000$. Two years later after spending $\$2,500,000$ the job was still not done, and it was estimated that another $\$3,6$ milions would be needed. The project was canceled. Many such disaster examples are given [112].

An emerging software engineering methodology that promises to adequately address many of these issues is component-based software engineering where components are intended to

[18, 126, 65]:

- be off-the shelf that come either from commercial sources or from another system,
- have non trivial number of functionalities,
- be of non trivial functionality,
- be self contained and possibly execute independently,
- be used as is,
- be integrated with other components to achieve required system functionality.

The composition of applications starting from their components is called software integration. The event-based architectural style is currently the most used communication paradigm for composing applications. *Therefore, it is required that a methodology that supports the development of event-based applications fully supports the development of component based systems.* This means that such an approach must be compositional, both at the conceptual level as well as the implementation level. At the abstract level, it must be possible to compose specifications of components as well as it must be possible to compose proofs about properties of the components. In particular, the methodology must allow building models and specifications, with a clearly defined and intuitive meaning. This clarity makes reuse and learning easier, because one can understand the whole by understanding the parts and then recombine them in a predictable way [37].

*On the other hand, any approach that supports the development of event-based applications must be oriented towards stepwise development of systems.* In particular, it should not be required that a complete application be developed before proof of correctness or testing be achieved. Such approaches have been recognized as unacceptable program development methodologies; erroneous design decisions are propagated until the system is implemented and attempted to be proven correct.

## 1.6 Contribution

This dissertation proposes a novel formal approach for building correct applications using the the event-based paradigm. This approach is called *LECAP*: **L**ogic of **E**vent **C**onsumption **A**nd **P**ublication. This logic is compositional; hence, intrinsically oriented towards construction of complex systems. LECAP is based on Jones's rely/guarantee [77, 121, 136] program derivation technique which is extended in two respects. First, we extend the specification of a program $z$ to include assumptions about the behavior of future subscribers. These assumptions need only be fulfilled when verifying properties of the

whole system. Second, we provide a rule for composing separately developed specifications into one large specification. Let us assume that we want to build a software system that satisfies the requirements $\phi_1$, ..., $\phi_n$. Our methodology consists of four steps (depicted in Figure 1.2):

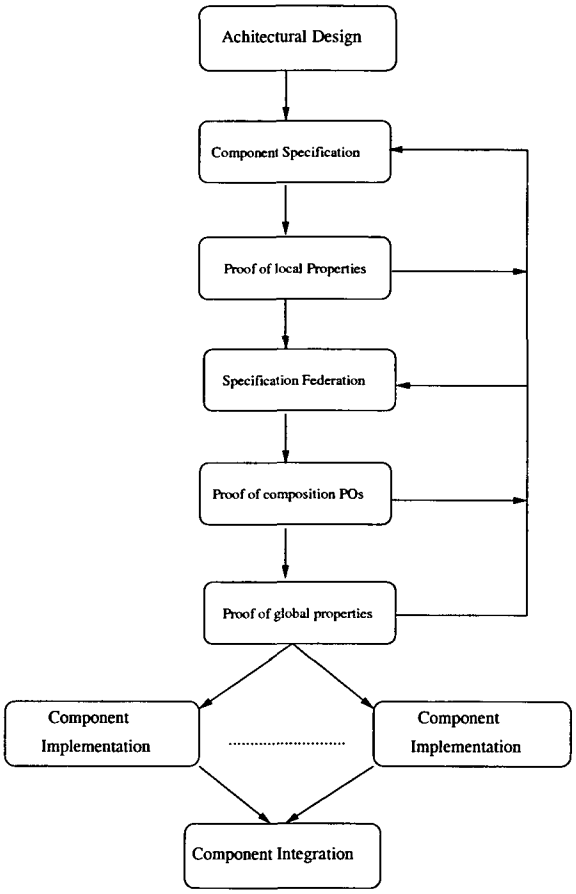


Figure 1.2: Software Development Process in the Context of Event-Based System

1. Architectural design of the system (identification of components).
2. Developing the formal specifications $S_1, \cdots, S_m$ of these components and verification of some local properties.
3. Composing the specification $S$ of the whole application starting with the specifications $S_1, \cdots, S_m$ of the components.
4. Independent refinement of the specifications $S_1, \cdots, S_m$ to some implementations $I_1, \cdots, I_m$.

It is important to stress that the development of $I_1, \cdots, I_m$ can be performed by different teams that know nothing about each other. Each of them receives some specification $S_i$ and

is required to deliver some code that satisfies this specification. In other words, $I_1, \cdots, I_m$ might be off-the-shelf components that satisfy the specifications $S_1, \cdots, S_m$. Indeed, this is one of the expected benefits of the loose coupling of components.

The approach we propose is a combination of bottom-up and top-down approaches. It is bottom-up in the sense that we start with some components that we specify (or that exist), build the specification of the application starting from that of the components and verify the properties of the system. The approach is top-down in the sense that the specified components can be developed following the usual top-down development process. This combination was shown to be suitable for the development of component based systems [12]. In particular, this is the way the EB paradigm is used to be applied, namely, for the integration of components [10].

The LECAP methodology consists of:

1. a core programming language with a clearly defined semantics for the development of programs that announce and consume events,
2. a technique for the specification of programs that announce events,
3. rules for the top-down development of components of an application,
4. a process and rules for the composition of specifications,
5. rules for tackling deadlock freedom in event based applications.

Intermediate results of this work have been previously published in the literature:

- The MOTION prototype was constructed and the different issues related to its architecture, its requirements, its design, and its implementation are described in [79, 80, 81, 82, 109].
- The reliability of the MOTION platform has been discussed in [43, 42].
- The fail attempt to ensure reliability in the MOTION platform led to the first steps towards a methodology for constructing correct event-based applications which are documented in [46].
- The systematic approach at the heart of this thesis is described in the papers [45, 47].
- The redesign of the MOTION platform is discussed in [48].

## 1.7 Roadmap

The remainder of this thesis is organized as follows. The next chapter presents the MOTION platform that we further use in this thesis as case-study. Chapter 3 discusses related works. Chapter 4 introduces the different mathematical concepts needed for understanding this thesis. Chapter 5 presents the LECAP programming language, a core programming language that allows developing programs that announce events. Chapter 6 presents techniques for the specification of event based applications. Chapter 7 is concerned with the top down development of event-based components. Chapter 8 discusses the issue of synchronization and mutual exclusion in event-based applications. The chapters 9 and 10 present some simple examples for illustrating the approach. In chapter 11, a simplification of the event-based architectural style is proposed and the set of rules for constructing related applications are presented. The notion of method invocation is added to our programming language in Chapter 12 while Chapter 13 and Chapter14 present the design and analysis of two non trivial case studies. Chapter 15 proofs that the proof system in this dissertation is sound. Future work are discussed in Chapter 16 and chapter 17 concludes the thesis.

# Chapter 2

# A Service Architecture for Mobile Teamwork

This thesis results from failed attempts to achieve dependability in the MOTION platform that we present in this chapter[1]. In effect, this platform was developed following a rigorous development process: the critical components were formally specified, the specification validated against the informal requirements, test cases derived from the specification, and the implementation tested against the formal specification. Despite this development process, the MOTION platform revealed some severe misbehaviors when put in the context of an event-based middleware.

## 2.1 Introduction

Mobile teamwork has become an emerging requirement in the daily business of large enterprises. Employees collaborate across locations and need support while they are on the move. Business documents (artifacts) and expertise need to be shared independently of the actual location or connectivity (e.g., access through a mobile phone, laptop, Personal Digital Assistant, etc.) of employees. Although many collaboration tools and systems exist, most do not deal with new requirements such as locating artifacts and experts through distributed searches, advanced information subscription and notification, and mobile information sharing and access. The MOTION service architecture that we have developed supports mobile teamwork by taking into account the different connectivity modes of users, provides access support for various devices such as laptop computers and mobile phones, and uses XML meta-data and the XML Query Language (XQL) for distributed searches and subscriptions. In this chapter, we describe the architecture and the components of our generic MOTION service platform for building collaborative applications. The MOTION Teamwork Services Components are currently being evaluated in two industry case-studies. The remainder of the chapter is structured as follows. Section 2.2 gives an overview of the

[1]This chapter is based on the MOTION paper published in the Proceedings of the International Conference on Software Engineering Knowledge Engineering SEKE 2002.

layered architecture. Section 2.3 discusses the main components of the MOTION's architecture. Section 2.5 gives an overview of the challenges for reliability that we encountered during the development of the MOTION platform.

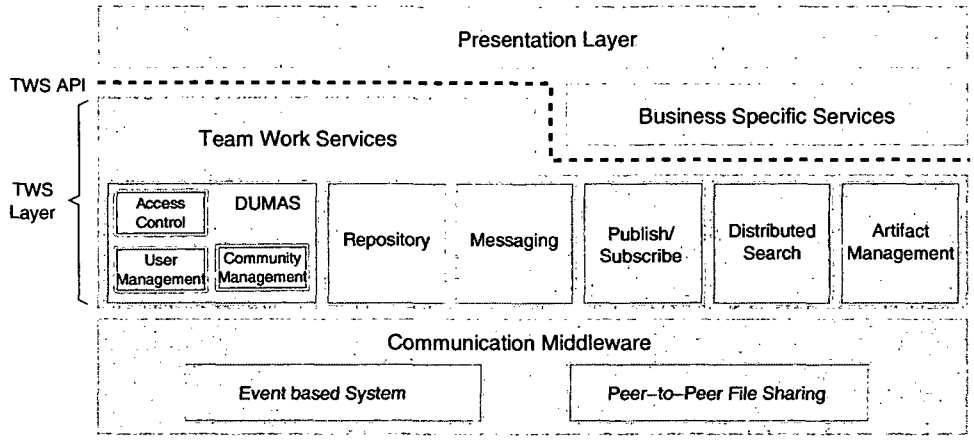


Figure 2.1: Overview of the MOTION Architecture

### Terminology

We first define some basic terms that will be used in relation to this case study.

- **Artifact**: Any document or file in the MOTION system (e.g., a text-processing document, a picture, a sound file, etc.)
- **Peer**: Any computing device connected to the MOTION system (e.g., notebook, Web browser, Personal Digital Assistant (PDA), etc.)
- **Community**: A collection of users in the MOTION system that are interested in a topic or that have a common property (e.g., "researchers", "paper writing","review", etc.)

## 2.2 MOTION Service Architecture

In this section, we give a brief overview of the layered architecture of the MOTION system and provide details about the main components in the following sections. Figure 2.1 depicts the MOTION architecture.

A conceptual view of the MOTION platform is presented in Figure 2.2. The MOTION system is composed of peers. Some act as host to services and some only act as clients. Any peer that is able to run the MOTION libraries can act as a service host. A typical MOTION configuration consists of desktop computers, laptops (i.e., notebooks and sub-notebooks) and PDAs that host services and clients such as Web browsers and WAP-enabled mobile phones that do not host services, but can only be used to remotely access them.

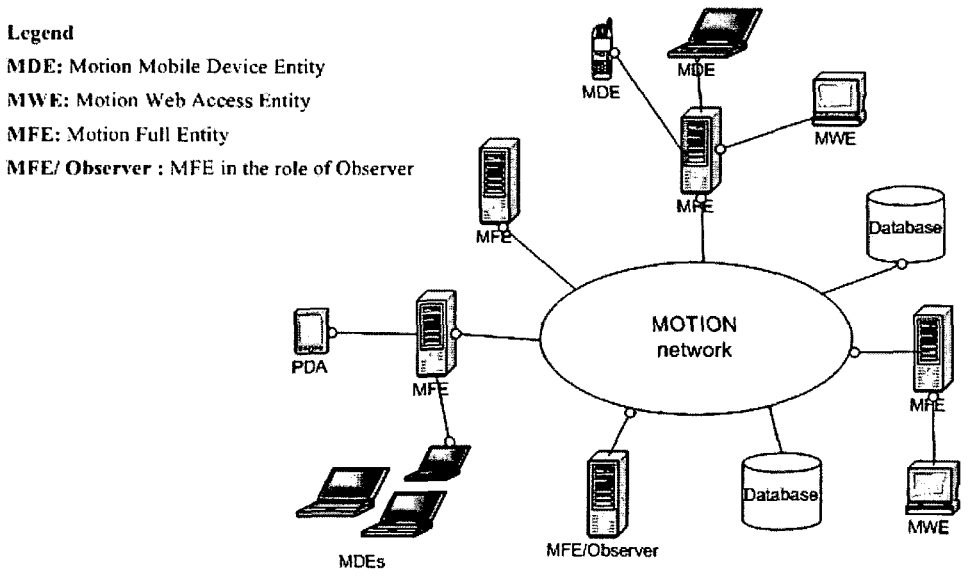


Figure 2.2: A Conceptual View of the MOTION Platform

The lowest layer of the architecture is the communication middleware. It offers basic communication services such as peer-to-peer file sharing through distributed searches and publish/subscribe (i.e., event-based system) mechanisms to the layers above. In the prototype implementation of the MOTION platform, this functionality is provided by PeerWare[101]. The communication layer, however, can be replaced by any other suitable middleware that provides distributed search and publish/subscribe support (e.g., distributed searches with JTella[87] and publish/subscribe with JEDI[31]).

The Teamwork Services (TWS) layer is situated directly above the communication middleware. This layer integrates the basic system components such as the repository and DUMAS (see next section) and provides an Application Programming Interface (API) to the teamwork services. This is a Java API in our prototype.

The TWS API offers services such as (1) storing artifacts and their meta-data (*profile*) in the local repository, (2) managing *resources* (artifacts, users, and communities), (3) sharing artifacts with other users in communities, (4) subscription to specific events in the MOTION system, (5) sending and receiving messages from other users or from the system, (6) managing access rights on resources, (7) and searching for resources based on their

profile information.

An application programmer can build business specific services (BSS) on top of the TWS API. By using the functionality provided by the API, the programmer can implement new functionalities according to the end-users's business requirements. Hence, the basic set of services provided by the TWS API can be customized and extended by businesses and organizations. For example, a company might be interested in integrating workflow support for transistor design into the platform whereas another might be interested in having document versioning support for artifacts.

The top layer of the architecture is the presentation layer. It provides a user interface to the services provided by the MOTION system. The presentation layer is built using the TWS API. Because of the need for mobility, a typical configuration has a number of user interfaces for different devices such as desktop computers, laptops, Personal Digital Assistants (PDAs), Web Browsers and WAP. In the current prototype, we have a native Java user interface that provides full functionality and an experimental lightweight Java PDA interface.

## 2.3 Teamwork Services Components

In this section, we describe the components of the MOTION Teamwork services layer.

### 2.3.1 The Dynamic User Management and Access Control Component

Confidentiality, security and privacy are important in many distributed multi-user applications. This has motivated the design and implementation of a number of access control models (e.g., [49, 113]). In most cases, the access control model is chosen by the software/security engineer and is hard-coded into the application. Hence, users of these applications have little or no support at all for customizing and adapting the security settings to requirements that may change over time.

The Dynamic User Management and Access Control Component [41] (DUMAS) is an access control component that is formally specified, verified, and implemented. Its goal was the creation of a generic, customizable component that satisfies different security requirements. This access control component provides support for managing users and roles (e.g., by creating, deleting, etc.) and assigning users to roles. The functionalities of DUMAS are grouped in three sub-components: a *user management* component, a *community management* component and an *authorization* component. These sub-components are strongly

connected in the sense that each of them is necessary for the two other sub-components to operate.

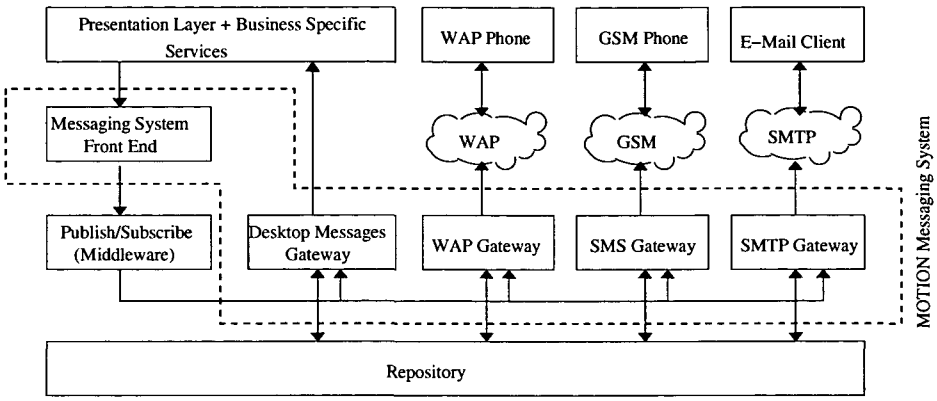


Figure 2.3: The MOTION Messaging Architecture

### 2.3.2 MOTION Messaging Component

MOTION Messaging is an integrated messaging service that enables users to communicate and exchange information. Notifications based on subscriptions are also delivered by this messaging service. MOTION messages are sent to users using technologies such event-based notifications, email (i.e., SMTP), GSM short messages (SMS), and wireless application protocol service indication (WAP SI)[52].

MOTION Messaging enables customers to stay in direct and constant contact no matter what devices they are using and where they are.

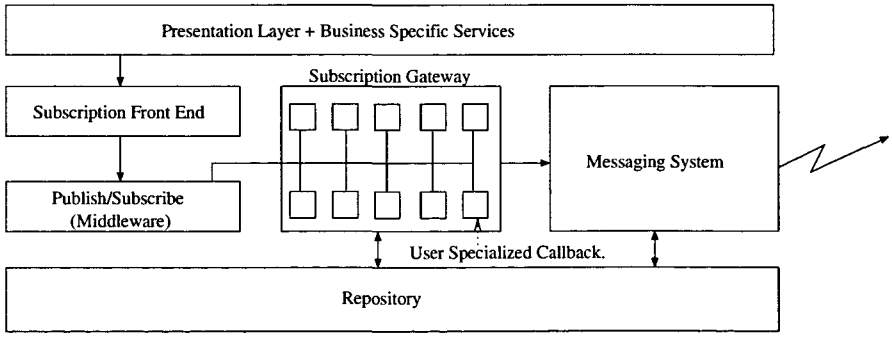


Figure 2.4: The TWS Layer Publish/Subscribe Architecture

The MOTION Messaging component in our prototype consists of five main components. These components are the SMS gateway, the SMTP (email) gateway, the standard messages gateway, the WAP gateway, and the MOTION front end component (see Figure 2.3). The

MOTION front end component is the interface between the business specific services and the MOTION Messaging component. It provides transparency to the business specific services by simple primitives for sending messages. These messages are transformed into XML events that are published through the underlying event-based middleware. Once a message is sent to a specific user, the configured gateway receives the corresponding XML event, transforms it and forwards it using the appropriate protocol (e.g., WAP gateways transform XML events to WAP SIs and SMS gateways to SMS messages). In case a MOTION gateway is unable to send a message for some reason, it can queue it in the repository that is available on the peer (host) the gateway is running on.

### 2.3.3 The Teamwork Services Layer Publish/Subscribe Component

The teamwork services layer's publish/subscribe component bridges the gap between the underlying middleware and the business-specific services and gives a uniform and consistent view of the *event* concept to the application layer.

Event-based middleware such as Peerware[101] and JEDI[31] allow components to subscribe and react to events by specifying a method that is invoked once an event occurs that matches a query. There are different realizations of this concept. In PeerWare, for example, the subscriber specifies a callback. The callback is an object of a class implementing the interface `peerware.EventCallback`. In JEDI, on the other hand, the subscriber directly specifies the name of the method to be invoked. In both cases, however, there is essentially no direct mapping between component-level (system) subscriptions and user-level (application) subscriptions.

To bridge this gap, we use *subscription gateways* and *user specialized callbacks* (see Figure 2.4). A user specialized callback is a component that handles subscriptions of a specific user. Whenever the user wishes to perform a subscription, she informs her specialized callback. This callback mediates between the underlying event-based system and the user. It receives the user's subscriptions and subscribes on her behalf. Once an event occurs that satisfies one of the user's subscription criteria, the corresponding callback is informed and it transforms the received event into a message. This message is sent to the messaging system and the user is informed based on her availability criteria.

The set of callback components running on a particular peer is referred to as the *subscription gateway.* A subscription gateway has to be configured for each user. Choosing a peer that can function as a subscription gateway is a configuration issue. Every peer in the MOTION system can be used as a subscription gateway and the configuration can be decided by organizations depending on deployment policies.

### 2.3.4 The MOTION Repository

Every peer in the MOTION system that runs MOTION services contains a repository that is used to store artifacts and profile information about users, communities and artifacts. This repository component is composed of two parts: an *XML* and an *artifact* repository. The XML repository is used to store XML profile information. The artifact repository is used to store artifacts that belong to a user. For example, when a user *Dr. Jaza* wishes to enter a document she is writing for the SEKE conference into the MOTION system, he would first enter meta-data about it such as the description of the document and its purpose. The meta-data would then be inserted into the XML repository and the document would be physically copied into the artifact repository. The repository component provides method calls for inserting, deleting, editing and querying meta-data that it manages.

### 2.3.5 Artifact Manager

The artifact manager component is composed of the MOTION repository component and the repository manager which is responsible for mapping remote transfer requests to commands in the repository. It retrieves, inserts, deletes or queries the information in the repository and provides the communication infrastructure between artifact exchanging peers. The artifact transfer protocol is HTTP. For example, when *Dr. Jaza* issues a distributed XQL request and sees that *Dr. Marco* has related work on information sharing, he can download that article from *Dr. Marco*'s repository. The repository manager component takes care of transferring the article (i.e., artifact) from the remote repository into *Dr. Jaza*'s local repository.

The Artifact Manager component acts as a wrapper to the MOTION Repository (i.e., XML and artifact repositories) and the Repository Manager. It provides artifact management API calls in the TWS API (e.g., insert an artifact, download an artifact, etc.).

### 2.3.6 Distributed searches

One of the distinguishing features of the services provided by the MOTION platform is its support for distributed searches. A key requirement in the MOTION industrial case-studies was the ability to locate information in a loosely coupled, distributed setting. Large organizations often have employees that do not personally know each other and in most cases cannot benefit from the work others are doing. For example, a group working on transistor design in Austria might have a problem that a group in the company located in South Africa branch has already solved. The ability to query artifacts, hence, is beneficial and in some cases success critical.

The TWS API provides querying mechanisms to search the artifacts that are in the MOTION system. The user can define XQL queries that are propagated through the system. The concept of distributed searches in the system is similar to searching provided by peer to peer systems such as Gnutella[60], Morpheus[91] and Napster[92]. Whereas these systems only provide search support for document file names, searching in the TWS API is more advanced and has a finer granularity.

## 2.4 Achieving Reliability in the MOTION Platform

The MOTION's dynamic user management and access control component was developed based on a lightweight formal method approach.

In a first step the component was formally specified using the VDM formal specification language (Vienna Development Method [130]). The IFAD VDM Toolbox [128] was used for this purpose. The formal specification of DUMAS was an extended explicit specification with about 700 lines of code. The main functionalities specified here are operations for managing users, groups and permissions.

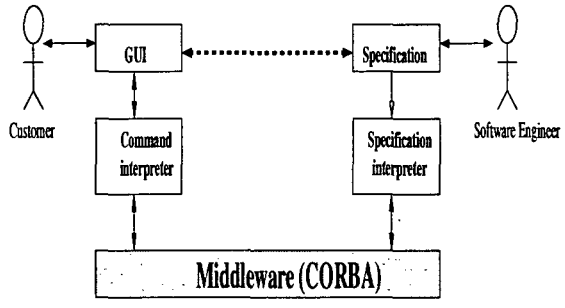


Figure 2.5: Visual Requirement Validation Process

The next step consisted of validating the specification against the informal specification of the intended component. The process was successfully carried on by applying a novel approach combining VDM-SL, CORBA, and Java. The process is illustrated in Figure 2.5 and detailed in [44]. The approach is a formal, visually-supported, approach to requirements validation. Given the formal requirements specification for a software component or system, we develop a graphical user-interface to be invoked by the user as a proxy for the system. User commands are mapped to the formal specification. Thus, the user's invocations of the commands exercise the formal specification. By validating the operations of the graphical user-interface, the user indirectly validates the formal specification.

The final step in this process was to verify the implementation of DUMAS. Formal testing was chosen for this purpose. Test cases were derived from the formal specification and the specification used for constructing a black box oracle. The architecture of the black-box

oracle was borrowed from the CORBA visual model used for the validation. The approach used for testing the DUMAS component consists of:

- executing the implementation,
- transforming its results by means of retrieve functions,
- and finally verifying whether the transformed results satisfy the specification.

This approach is detailed in [43] and provides the following benefits:

- the same specification used for generating test cases is used for constructing the test oracle,
- the post-conditions don't need to be translated to a high-level programming language,
- the approach can be applied to all CORBA-compliant programming languages,
- the approach can handle programs with non-deterministic results.

The architecture of this test oracle is presented in Figure 2.6

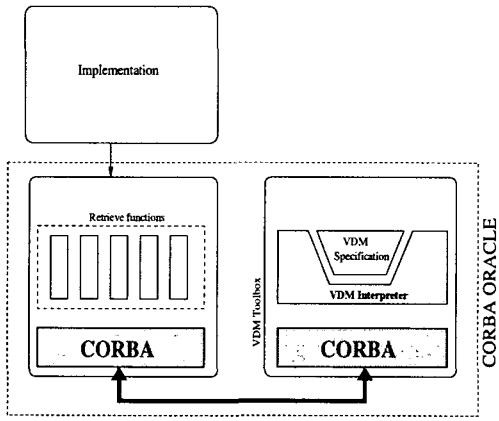


Figure 2.6: Architecture of a CORBA Oracle.

## 2.5 Reliability Challenges in the MOTION Platform

The validation and verification processes described in the above section were performed in a client/server environment. In the validation process, the server was the formal specification

that was interpreted by the VDM toolbox's interpreter which therefore played the role of a server. The client in this case was a graphical user interface that gave the end-users the illusion of manipulating a real application.

In the case of the verification process, the Java implementation of the component is exercised against the result of the interpretation (by the VDM Toolbox) of the specification. The DUMAS's component was shown to be robust in this client/server setting.

This robustness, however, was revealed to be architecture dependent. In fact, the component revealed some severe misbehaviors when put in the event-based peer-to-peer architecture described in section 2.2. Further investigations revealed some serious challenges in the construction of event-based applications: there is neither a methodology for building such applications, nor suitable techniques for verifying or testing them. This led to the investigation of the methodology proposed in this thesis.

## 2.6 Summary

The importance of the event-based paradigm is now widely accepted. A continuously increasing number of applications are constructed based on this paradigm. This reality should be sufficient for motivating the development of an adequate software development methodology that can deliver robust products in a relatively rapid time to market. Yet, one's own experience is more motivating and instructive.

This chapter presented a detailed description of the MOTION platform. One of the main components of this platform was formally specified, the verification validated and the implementation verified. Despite this, some severe misbehaviors were observed that led to the development of the methodology presented in this dissertation. A part of this platform will be used as a case study later in this thesis.

# Chapter 3

# Related Work

Although the event-based architectural style is at the heart of countless software systems, research and products that leverage this paradigm have focused so far on efficiency issues and have neglected methodologies for constructing such systems. The aim of this chapter is to support this claim; a survey of methodologies for constructing event-based applications is given.

The remainder of the chapter presents different contributions to the improvement of the quality of event-based applications. They are presented from the less to the most formal ones. The first of these sections presents the concept of scope in event-based systems. The goal is to achieve the same effect as in object-oriented programming languages: information hiding. In Section 3.2, a concept that in some respects resembles the event-based paradigm is presented, namely broadcasting. Directly related to our work are the issues of formal specification of event-based applications (discussed in Section 3.3), the issue of the verification of the properties of such applications (see Section 3.4), and the issue of model checking them (presented in Section 3.5).

The approach proposed in this thesis is strongly based on techniques for constructing parallel programs. Jones's rely/guarantee technique [77] (extended e.g. by Stølen [120, 121, 122] and Xu [136]) and the work of Owicki and Gries [98] are among the approaches that influenced the construction of parallel programs. In fact, it is argued that Jones's technique is the first and most fundamental compositional method for the correctness proof of state-based parallel programs [94]. We review these approaches in Section 3.6. Section 3.7 summarizes the chapter.

## 3.1 Scope and Mapping in Event-Based Applications

Information hiding and abstraction have been recognized as useful concepts for structuring software systems [99, 100]. They led to techniques such as encapsulation [117], modularization [100], classes, and objects [16]. It is, therefore, legitimate to envision that they may bring the same benefits in event-based systems. Fiege, Mezini, Mühl, and Buchmann

[50, 51] introduced scoping and mapping to achieve information hiding and abstraction into event-based systems. In particular, they intend to contribute to four issues: bundling of components, heterogeneity, flexible configurations, and support of activities.

Bundling consists of tying a set of components together; this is done both at the syntax and at the semantics levels. At the syntax level, a bundle is a collection of components delimiting the visibility of events they announce. A bundle is also a component that has a semantics and, as such, may announce and consume events.

The event-based architectural style is recognized as a suitable communication paradigm for highly heterogeneous environments. The development of applications in these environments must, hence, be based on concepts that are oriented towards the support of this heterogeneity. An example of heterogeneity factor is the potential variety in the semantics of notifications in large distributed environments [50]. Event-based systems must, therefore, support various event models. Consequently, bundling must also delimit common syntax and semantics areas.

The third motivation for bundling components is that event-based applications seem to require that sessions of independent activities be separated from each other. Such a grouping of components may help in controlling interference.

To solve the above issues the concept of scope is proposed as an abstraction that bundles a set of producers and consumers in groups characterized by some criteria defined either semantically or syntactically [50]. A scoped event-based system is defined as an acyclic graph whose nodes are components and scopes and whose edges are binary relations over the set of edges. A scoped event-based system, hence, defines a superscope/subscope relationship over components of an application. Figure 3.1 shows an example of scoped event-based system.

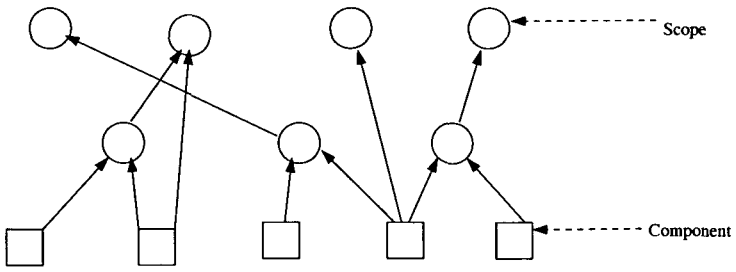


Figure 3.1: A Scoped Event-Based Application

The concept of scope studied by Fiege, Mezini, Mühl, and Buchmann [50, 51] seems to be a promising technique for controlling interference in large scale and distributed applications. Nevertheless, the issue of how to build a correct application is completely open. The authors argue that the development of event-based applications do not need other design and engineering approaches than those already known in software engineering. This position

is questionable for three reasons. First, we are not aware of a single real-life event-based application whose correctness is claimed to have been tackled successfully. This may indicate that there are problems in applying conventional development approaches. Next, our experience in designing the MOTION platform revealed that traditional techniques for ensuring dependability are not (at least not straightforwardly) applicable to event-based applications. Finally, other researchers have investigated new methodologies for designing event-based applications [10, 35, 36, 56] and they indeed claim that event-based applications are hard to reason about and to test [56]. This is a clear sign that existing methodologies are not well-suited for the development of event-based applications.

## 3.2 Broadcasting Systems

Broadcasting is a communication paradigm where one process speaks at a time and is heard instantaneously by all others [103]. A significant amount of work [39, 66, 103, 104, 105, 106] has been done on this topic that led to an important number of theories. This section overviews two of these theories: the calculus of broadcasting systems (CBS) [103], and the $b\pi$ calculus [39]. Essentially, the issue in such works is the notification of all the components in the system.

### 3.2.1 CBS: A Calculus of Broadcasting Systems

CBS is a calculus resembling Milner's CCS (calculus of communicating systems) [89] where the handshake communication concept is replaced with broadcasting; processes speak one at time and all processes are interested in all messages. CBS is mainly a formal model for packets broadcast in Ethernet-like communication media as provided in hardware for local area networks, as well as radio and mobile telephony networks. CBS models an idealized local area network using concepts from process calculi.

As in CCS [89], the behavior of a system consists of communicating actions. A process $p$ may send a message $w$ and evolve as $p'$ (denoted as $p \xrightarrow{w!} p'$) while $q$ may receive a message and become $q'$ (denoted as $q \xrightarrow{w?} q'$). The processes can be combined using the parallel operator: $p \mid q \xrightarrow{w!} p' \mid q'$. This composition is captured by the inference rule:

$$\frac{p \xrightarrow{w!} p' \qquad q \xrightarrow{w?} q'}{p \mid q \xrightarrow{w!} p' \mid q'}$$

which claims that if its premises hold, then so does its conclusion. One-to-many com-

munication is obtained by naturally applying the above rule incrementally. We, for instance, apply the rule twice to derive that if $p \xrightarrow{w!} p'$, $q \xrightarrow{w?} q'$, and $r \xrightarrow{w?} r'$ hold, then $p \mid q \mid r \xrightarrow{w!} p' \mid q' \mid r'$ follows.

$$\dfrac{\dfrac{p \xrightarrow{w!} p' \qquad q \xrightarrow{w?} q'}{p \mid q \xrightarrow{w!} p' \mid q'} \qquad r \xrightarrow{w?} r'}{p \mid q \mid r \xrightarrow{w!} p' \mid q' \mid r'}$$

Clearly, the parallel composition operator '|' is commutative and associative. The calculus supports the *Nil* process which says nothing and is a neutral element for the parallel composition operator. Other constructs are the if-construct, the choice-construct and scoping. Communication is visible to the environment per-default. With static scoping, the communication is hidden to the environment. The calculus defines weak and strong bisimulations which are equivalence relations on processes that group processes with the same behaviors.

### 3.2.2 The $b\pi$ Calculus

The $b\pi$-calculus is another calculus of broadcasting systems which is based on CBS [103] and the $\pi$-calculus [90]. Unlike CBS which does not allow modelling reconfigurable finer topologies of network of processes which communicate by broadcast [39], the $b\pi$-calculus was developed with the explicit requirement of solving this issue. The $b\pi$-calculus can be seen as a $\pi$-calculus where the message-passing primitive is replaced with the broadcast communication primitive.

As the $b\pi$-calculus is based on the $\pi$-calculus it is superior to CBS because channels can also be messages (hence, support for mobility). The concept of dynamic scoping is based on this feature by combining it with static scoping which is the ability to keep a communication session separated so that the risk of interference is limited.

Calculi of broadcasting systems are different from event-based systems in many respects.

- In event-based systems, each component specifies the kind of messages it is interested in while in broadcasting systems, all components receive the broadcasted message. The event-based infrastructure is responsible for invoking the subscribers when events occur that match their subscriptions.
- The event-based communication mechanism is intended for asynchronous systems while broadcasting is an unbuffered communication mechanism. In event-based sys-

tems, components don't need to be ready when events occur that satisfy their subscriptions. This makes this communication paradigm suitable for supporting disconnectedness.

- Arbitrarily many components may announce an event in event-based systems while in broadcasting systems, only one process can talk at a time.

- Another difference (and, perhaps the most important one) is that components interested in an event $e$ announced by the program $p$ are not running from the beginning of the execution of $p$, but are invoked following the announcement of $e$. The execution of such subscribers may, therefore, start at the beginning of the execution of $p$ (in which case they will be executed in parallel with $p$), at the end of the execution of $p$ (in which case we have a sequential composition), or more generally at any point of the execution of $p$; which makes things at least as complicated as parallel composition.

The requirements of event-based systems are, therefore, very different from that of broadcasting systems.

## 3.3 Formalizing Architectural Styles

Several researchers have attempted to provide formal techniques that can support the treatment of event-based applications. Although the ultimate goal of such works is reliability of these applications, the developed approaches have not been successful. Examples of such approaches are that of Garlan and Notkin [55], as well as that of Abowd, Allen, and Garlan [1, 2] who propose frameworks for formalizing architectural styles in general and implicit invocation in particular. Dingel, Garlan, Jha and Notkin argue in [36, 35] that these approaches primarily focused on taxonomic issues, and do not provide an explicit computational model that permits compositional reasoning about the behavior of event-based systems. Let us clarify this statement in the light of Abowd/Allen/Garlan framework for formalizing styles [1, 2, 71].

Software architectures and architectural styles have mainly been defined by means of boxes and lines. This makes it difficult to argue on the reliability of applications. This is particularly unfortunate since the cost of specifying and analyzing a style is amortized across all instances [71]. To remedy this situation, a framework for formalizing architectural styles is proposed [55] in which they are described in terms of mappings from their syntactic domains to their semantic domains. By providing a formal vocabulary for the description of styles, the framework allows new styles to be defined in a way such that they are uniform with existing ones. Essentially, the framework proposes a function for giving meanings to architectures and styles which are defined syntactically using components, connectors,

ports, configurations, attachments, and roles.

A component is the computational part of an architecture. Components may communicate with each other by means of connectors. The component's connection point is called a port. Formally, a component is modeled as a collection of ports and a description of the behavior of the component. This is denoted in VDM as:

$PORT$ = token;

$COMPONENT\text{-}DESC$ = token;

$COMPONENT$ ::
$ports$ : $PORT$-set
$description$ : $COMPONENT\text{-}DESC$;

where $PORT$ and $COMPONENT\text{-}DESC$ are two types that we do not need to further define (declared as token). A graphical representation of this concept is presented in Figure 3.2 which shows a connector capable of connecting three components.

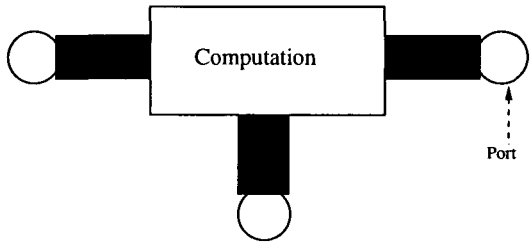


Figure 3.2: A Component

A connector embodies the communication mechanism between components. Instead of being pre-determined between two components, a connector provides placeholders called roles that permit the connections to ports of components. A graphical representation of a connector is depicted on Figure 3.3. Formally, a connector is represented as a set of roles and the description of its behavior:

$ROLE$ = token;

$CONNECTOR\text{-}DESC$ = token;

$CONNECTOR$ ::
$roles$ : $ROLE$-set
$description$ : $CONNECTOR\text{-}DESC$;

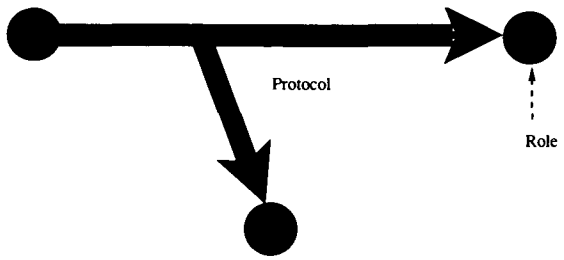


Figure 3.3: A Connector

A configuration represents a set of component instances that communicate with each other by means of some connector instances. This is depicted in Figure 3.4. The formal specification on the other hand has some invariants that require that any association role-port in the attachment be such that the role is that of one of the defined connectors and the port is that of one of the defined components.

$$
\begin{aligned}
&\mathit{CONFIGURATION} :: \\
&\qquad \mathit{components} : \mathit{ROLE}\text{-set} \\
&\qquad \mathit{connectors} : \mathit{CONNECTOR\text{-}DESC}; \\
&\qquad \mathit{attachment} : \mathit{ROLE} \mapsto \mathit{Port}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{inv}\ \mathit{cf} \triangleq \mathsf{let} \\
&\quad \mathit{inv}0 = \forall r \in \mathsf{dom}\ \mathit{cf}.\mathit{attachment} \cdot \exists c \in \mathit{cf}.\mathit{connectors}, r \in c.\mathit{roles}, \\
&\quad \mathit{inv}1 = \forall p \in \mathsf{rng}\ \mathit{cf}.\mathit{attachment} \cdot \exists c \in \mathit{cf}.\mathit{components}, r \in c.\mathit{ports}\ \mathsf{in} \\
&\mathit{inv}0 \wedge \mathit{inv}1;
\end{aligned}
$$

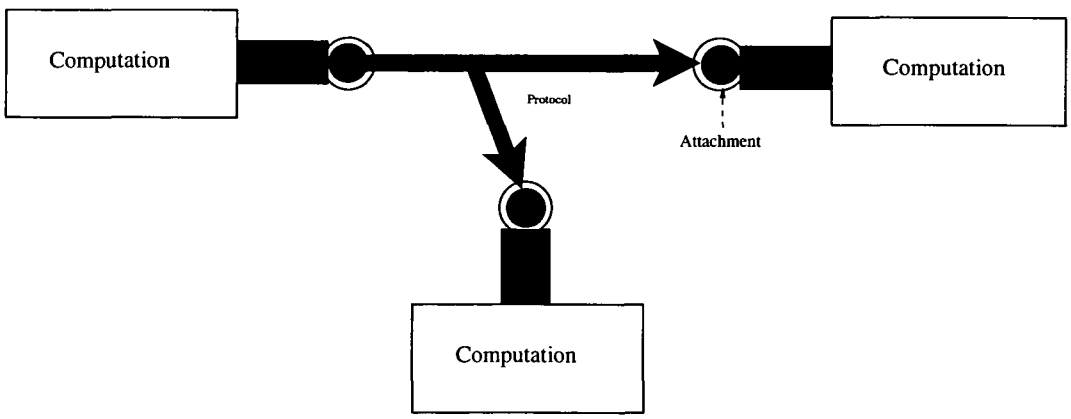


Figure 3.4: A Configuration

Based on these concepts the formal specifications of the event-based paradigm can be given. A component is represented as an object with a private set of variables that represents its internal state and a collection of methods that can be invoked externally. The behavior

of such an object is modeled as a state machine with a transition function that relates a method invocation to a tuple consisting of a state and a set of events.

$$Event = \text{token};$$

$$METHOD = \text{token};$$

$$STATE = \text{token};$$

$$\begin{aligned}
&Object::\\
&\quad methods : METHOD\text{-set}\\
&\quad events : EVENT\text{-set}\\
&\quad states : STATE\text{-set}\\
&\quad start : STATE\\
&\quad transitions : (METHOD \times STATE) \mapsto (STATE \times EVENT\text{-set})
\end{aligned}$$

$$\begin{aligned}
&\text{inv } cf \triangleq \text{let}\\
&\quad inv0 = cf.start \in cf.states,\\
&\quad inv1 = \text{dom } cf.transitions = \{(m, s), m \in cf.methods \wedge s \in cf.states\},\\
&\quad inv2 = \text{rng } cf.transitions = \{s, m \in cf.methods \wedge s \in cf.states\} \text{ in}\\
&inv0 \wedge inv1;
\end{aligned}$$

In the context of the event-based paradigm, a connector becomes a distributor which takes an announced event and invokes the set of methods subscribed to that event. The description of a distributor is therefore:

$$\begin{aligned}
&DISTRIBUTOR::\\
&\quad methods : METHOD\text{-set}\\
&\quad events : EVENT\text{-set};
\end{aligned}$$

A configuration is now known in the event-based context as a set of objects that interact by means of distributors. The overall binding of methods to events is derived from the individual distributors. The formal specification is:

$$\begin{aligned}
&InteractingObjectSet::\\
&\quad objects : Object\text{-set}\\
&\quad distributor : DISTRIBUTOR\text{-set};\\
&\quad binding : EVENT \overset{m}{\longleftrightarrow} METHOD;
\end{aligned}$$

$$
\begin{aligned}
&\mathsf{inv}\ io \triangleq \mathsf{let} \\
&\quad inv0 = \forall\, o_1, o_2 \in io.objects \cdot o_1 \neq o_2 \Rightarrow o_1.methods \cap o_2.methods\{\}, \\
&\quad inv1 = io.binding = \bigcup_{d \in io.distributors} d.events \times d.methods, \\
&\quad inv2 = \forall\, e \in \mathsf{dom}\ io.binding \cdot \exists o \in io.objects,\ e \in o.events, \\
&\quad inv3 = \forall\, m \in \mathsf{rng}\ io.binding \cdot \exists o \in io.objects,\ e \in o.methods\ \mathsf{in} \\
&\quad inv0 \wedge inv1 \wedge inv2 \wedge inv3;
\end{aligned}
$$

At this level of detail, we can justify the claim that this work mainly discusses taxonomic issues. In the first place, one can note that the concept of method has no precise meaning (defined with token). This means that it is not possible to reason about the behavior of the method (at least not without extending the framework). It is not said how a method must be specified. For instance, a key question in formally specifying event-based applications is how does a designer specify that a method $m$ announces an event $e$ whenever the state satisfies the condition $Q$? The next issue in such approaches is that there is no indication of whether the specifications are realizable or not. Given such a specification, what is the next step in building an application? How does a designer show that a given application satisfies such a specification?

This is not to say that no property can be verified in this framework. For instance, in [71], Daniel Jackson presents a framework for automatically analyzing architectural styles where the event-based architectural style is taken as case study; general properties related to the style are checked. Such properties are for instance showing that the chain of causality in the event-based system is acyclic. Nevertheless, the issue of constructing correct event-based applications remains largely open.

## 3.4 Correctness of Event-Based Applications

The only approach that directly tackles the issues of building correct event-based applications is by Dingel, Garlan, and Notkin [35, 36]. A method for reasoning about event-based applications is proposed. This approach, which we call Dingel's approach is also based on Jones's rely/guarantee paradigm. The framework includes a formal computational model for the event-based paradigm, techniques for specification of systems and an approach for reasoning about the correctness of programs.

The proposed formal model is based on a programming language which is essentially a while-language extended with the announce and the consume constructs. The announce construct is intended for publication of events while the consume construct allows methods to declare the kind of events they are interested in. The syntax of this language is given as follows:

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

The execution of such a program and therefore its semantics is given relatively to an event-based system which is modeled as a tuple ($M$, $V$, $EM$, $Ex$) composed of a set of programs $M$, a set of variables $V$ accessible to programs in $M$, a set of external events $Ex$ and a binding of methods to events. The set of methods $M$ represents the set of methods to be invoked when events are matched by the event-based infrastructure. The binding $EM$ determines the set of events that a method $m$ in $M$ is interested in. The set of external events is the set of events announced by programs not in $M$. The semantics of the announce construct is that when the event $e$ is announced, an entry $(e, m)$ is added to the set of active events for any method $m \in M$ that is interested in this event (specified by the binding $EM$). On the other hand, the execution of the consume statements corresponds to the removal of the entry $(e, m)$ from the set of active events. The other imperative constructs have the standard semantics.

In addition to the computational model briefly described above, Dingel's framework proposes a way of specifying the behavior of an event-based system. A specification consists of a pre-condition, a rely-condition, a guar-condition, and a post-condition. Let us consider an event-based system $\mathcal{S}$ and a specification $(P, R, G, Q)$. $\mathcal{S}$ satisfies the specification $(P, R, G, Q)$ iff any computation of $\mathcal{S}$ that starts in a state satisfying the pre-condition $P$ and is executed in an environment whose transitions satisfy $R$ will terminate in a state satisfying $Q$ and any of its transitions will satisfy $G$. Based on this, the process of proving that an event-based system satisfies a specification is given.

Let $\mathcal{S} = (M, V, EM, Ex)$ denote an event-based system. To show that the system $\mathcal{S}$ satisfies some partial correctness property $T$, 4 steps are required:

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

This approach has a number of shortcomings.

1. It assumes a programming language with a *consume* construct. Each method must start with this statement that specifies which events the method is interested in.

Dingel et al. use the consume construct to model invocation of methods by the event-based system and to trace changes in the pending event infrastructure. They, however, recognize that this construct "introduces an unnecessary dependency between the event-method binding and the program of a method [36, 35]." Further, no real programming language or event-based system needs such a construct.

2. The underlying specification technique is based on a pending event infrastructure. The primary intent of an event-based system is not to queue events, but to dispatch them to subscribers. Queuing events results from the fact that an event-based system might not be able to forward events at the speed at which they are received. Hence, we suggest that, although it may be important to take it into consideration at the implementation level, a mechanism for queuing events should only influence the abstract model in such a way that it does not complicate the reasoning too much.

3. Dingel et al. [36, 35] assume in their work that when a program is running it cannot be triggered anymore. No mechanism is however given for achieving this. On the other hand, there are applications where such a limitation is not acceptable.

4. The approach does not take the definition of new subscriptions into consideration. A static binding $EM$ is assumed. In this sense, the approach seems to miss a fundamental aspect of the event-based paradigm which is (because of loose coupling) to ease the integration of new components.

5. Dingel's approach is intended for a-posteriori verification of systems instead of stepwise construction of systems: components of the completed programs are verified in isolation and then put together where general properties are proved. Jones [77] argues that such approaches are unacceptable as program development methods: erroneous design decisions taken in early steps are propagated until the system is implemented and attempted to be proven correct.

Although Dingel et al. [36, 35] do not claim to propose a method for the stepwise construction of systems, the fact that their approach is based on Jones's rely/guarantee might lead one to expect that it can also be used for such a purpose. To see why this is difficult, let us consider the following development method naively derived from the above reasoning technique:

To construct a system $S$ that satisfies some partial correctness property $T$, 6 steps must be followed:

1. Define the pre-, rely, and post-conditions $P, R, Q$ of the system.

2. Identify the set of methods $M$ of the system.

3. For each method $m \in M$ (not yet implemented), define the guarantee conditions $G_m$ and $G_{M\setminus\{m\}}$ such that $(m, V, EM, Ex)$ satisfies $(P, R \vee G_{M\setminus\{m\}}, G_m, Q)$ and $(M \setminus$

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

4. Conclude that $(M, V, EM, Ex)$ satisfies $(P, R, \bigvee_{m \in M} G_m, Q)$
5. Show that any system that satisfies $(P, R, \bigvee_{m \in M} G_m, Q)$ also satisfies $T$.
6. Now, refine each method $m$ to some implementation.

This approach, however, does not work since there is nothing that relates the specifications $(P, R \vee G_{M\setminus\{m\}}, G_m, Q)$ of the different methods to each other. This relation should be provided by the event-based system. The methods should communicate with each other through the event-based system by announcing and consuming events. This notion of announcement and consumption of events is, however, absent from the specification, hence the insufficiency of the specification and the inadequacy of the approach.

## 3.5 Model Checking Event-Based Applications

Model checking [27] which is a formal technique based on inspection of the state space of a system is an intriguing alternative to the formal proof of software systems as a substantial part of the process is carried out automatically. The approach is particularly well-suited when the state space of the system is small enough. Garlan and Khersonsky [56, 57] have tried to develop a framework for model checking event-based applications.

In fact, one of the issues in model checking a system is the construction of an abstract finite state model of the system under analysis without eliminating the class of errors that one wants to detect. In addition, a suitable structure for the abstract state model must be developed. The function relating this structure to the concrete system must be such that it eases the mapping of errors found in the abstract model to the real system.

A way of alleviating the difficulties in the model checking process is to find a generic structure for each class of applications such that they can simply be reused when verifying an application that falls into one of these classes. The intent of Garlan's work is to create such a generic structure for event-based applications by identifying the main structural elements of an event-based system that are suitable for model checking. For this purpose, six main constituents are identified in event-based systems: publishers and consumers, event types, shared variables, event bindings, event delivery policy, and the concurrency model. From this, two stumbling blocks are identified for creating a state model for event-based systems:

- the construction of finite-state approximations for each consumer/publisher and

- the construction of a run-time mechanism that models the event-based infrastructure.

A set of restrictions are recommended for achieving the first step: all data has a finite range, the event alphabet and the set of consumers and bindings are fixed at runtime, there exists a specified limit on the size of the event queue, there is a limit on the size of invocation queues.

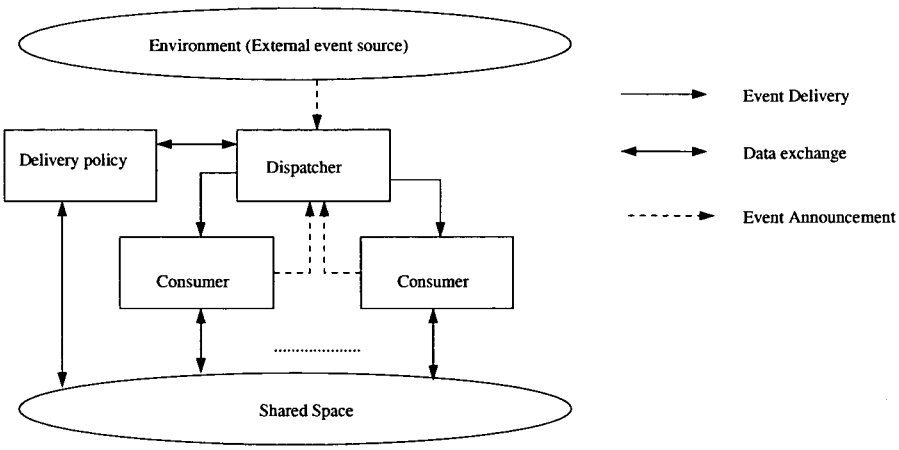


Figure 3.5: Structure of an Event-Based System Model for Model Checking

The second issue is addressed by providing pluggable state modules that allow modelers to choose from one of the possible run-time models. The problem is factored as depicted in Figure 3.5 adapted from [56]. The user provides:

- a model of the consumers and publishers,
- a set of variables accessed by the consumers and publishers,
- a finite list of events,
- the event-method binding,
- a model that specifies the interference of the environment,
- a concurrency model (chosen from a list of given concurrency models),
- a dispatch policy (chosen from a list of models).

These parts are translated into a set of interacting state machine descriptions that can be executed by a model-checker such as NUSMV [26].

A close look at this approach reveals that in fact, the authors only propose a model for the event-based infrastructure which is typically a middleware, or an integration framework

(e.g. PeerWare [101], Spear [21], OLE [17], Yeast [83], TIB/Rendezvous [131]). The support for the verification of the application constructed on top of such middleware is limited to some recommendations such as restricting the range of data and restricting the event alphabet. This is a very limited contribution to helping the developer in ensuring the reliability of his application. On the other hand, as Dingel's approach, this approach is an a-posteriori verification process: the properties of a completely developed application are checked. Fixing an erroneous design decision detected after the implementation of an application may require re-designing the whole application. This approach, however, is inferior to Dingel's approach as it is not compositional.

## 3.6 Development of Interfering Programs

Our methodology for the stepwise construction of correct event-based applications is based on Jones's rely/guarantee technique for the development of interfering programs. This directly results from our interpretation of the announce construct: the set of subscribers to an event are invoked and executed in parallel with the remainder of the announcing program. More precisely, we base our work on that of Stølen [120, 121, 122] and partly on that of Xu [135, 136], which are extensions of Jones's work to support the development of deadlock free concurrent systems. Stølen's approach also supports the use of auxiliary variables both as a verification tool and as a specification tool. It can be seen as a compositional reformulation of Owicki/Gries [98] method of verifying parallel programs.

Formal methods based on model-oriented specifications like VDM or B are applicable to the development of sequential operations. In such approaches, state components can be common to several operations but only one operation is executed at a time. A sequential operation can then be interpreted as a binary relation on the state space and specified with pre- and post- conditions. The additional complexity of concurrent versus sequential operations is due to the presence of interference: operations access state components that can be modified by the execution of the other operations during their own execution.

The usual way of mastering interference in parallel programs is to specify processes in terms of assumptions and commitments. This approach was first proposed by Francez and Pnueli [53]. The basic idea is: if the environment, by which is meant the set of processes running in parallel with the one in question, fulfills the assumptions, then the actual process is required to fulfill the commitments. Jones employs rely- and guar- (antee) conditions [77] in a similar way. However, while earlier approaches essentially focus on program verification, the goal of the rely/guarantee method is top-down program development.

In fact, pre/post-conditions specifications for sequential programs are examples of assumption/commitment specifications, in which the pre-condition expresses the conditions on the program variables that the program relies on when it starts its execution, and the

post-condition expresses the condition that the program guarantees when it terminates its execution. Termination of an interfering program in an acceptable state also requires assumption about the initial state, but this is not sufficient, one needs assumptions about the interference of other operations.

The specification of a program in interfering environments is, therefore, given by a formula of the form $z$ *sat* $(P, R, G, Q)$ where $z$ is the program in question and the specification $(P, R, G, Q)$ consists of the pre-condition $P$, the rely-condition $R$, the guar-condition $G$, and the post-condition $Q$. The program $z$ satisfies its specification if when executed in a state satisfying the pre-condition $P$ and in an environment whose interference satisfy $R$, it will terminate in a state satisfying $Q$ and is such that each of its steps that changes the state satisfies the guar-condition $G$.

Other methods (e.g. [98, 7, 78, 25] ) have been proposed for tackling the correctness of interfering programs. They are, however, characterized by proving the components of the completed programs in isolation and then proving that the proofs do not interfere. It is argued [77, 120, 121, 122] that such approaches are unacceptable as program development methods. The rely/guarantee method is superior since it allows erroneous decisions to be stopped and corrected at the level where they are taken.

## 3.7 Summary

We have given an overview of the current status in constructing event-based applications. From this, it is obvious that there is a clear need for a sound, applicable and useful methodology for developing event-based applications. Other researchers have tried to fill this gap, but many issues remain open.

Essentially, a methodology for the development of event-based applications must allow leveraging its key concept: loose coupling of components. All the approaches we described above fail in this respect since they don't address the issue of adding new components/methods in an existing system. Even the work of Garlan and Notkin [55] which is just a framework for describing the taxonomy of a system fails in this respect: they assume that the subscribers to an event are known when designing the various components. Also, Dingel's work is based on the same assumption that the binding is pre-defined.

One of the most advocated works in this area is in fact the framework for specifying architectural styles proposed by Garlan and Notkin [55]. This framework, however, is silent about how to specify the methods (publishers and consumers) of an event-based application. Worst, there is no bridge between specifications and implementations; there is no way to verify that a program satisfies its specification.

A more recent work is that of Dingel, Garlan, Jha, and Notkin [36, 35] which proposes

a framework for verifying event-based applications. In addition to being based on the assumption that there is a fixed set of subscriptions, the approach is inadequate for the verification of large scale and complex software systems as it is an a-posteriori approach.

We propose an approach, LECAP which is intended to overcome these limitations. In this framework, programs are specified by means of their pre-, rely-, guar-, and post-conditions. In our approach, the components are specified independently from each other. One does not need to know which components are subscribed to which events. Subsequently, the development of a system can be decomposed into the development of small parts. In addition to this, our approach is a top-down development approach, which therefore supports the stepwise development of software systems. The next chapter gives an overview of the theoretical background needed for understanding this thesis.

# Chapter 4

# Mathematical Introduction

## 4.1 Motivation

The chapter introduces the symbols, the basic concepts, and the notation used in this thesis. Our framework for constructing correct event-based applications is built in a many-sorted logic. This has four advantages. On one hand, many-sorted logic exhibit many of the properties that first-order logic are interesting for (e.g. strong completeness, compactness, Löwenheim-Skolem properties). Second, it is well known that many-sorted logic reduces elegantly to first-order logic. The reduction is composed of two steps. At the syntax level many-sorted formulas are translated to first-order formulas. At the semantics level, many-sorted structures are translated into one-sorted structures. Third, although the many-sorted language can be reduced to a one-sorted language, any one-sorted language can trivially be included in a many-sorted language. Fourth, the many-sorted nature of the many-sorted logic makes it a very expressive logic that is viewed as a unifier logic; a logic into which other logic such as higher-order logic, modal logic, or dynamic logic can be translated. The many-sorted logic presented in this chapter and assumed throughout this thesis has a higher-order logic look which will allow us to quantify over states.

To read this thesis, the intuitive everyday knowledge of first-order logic, set theory, program specification and verification is needed. In effect, most readers may already be familiar with the theoretic apparatus presented in this chapter. Nonetheless, for the sake of a self-contained thesis, we recall these notions. In case, however, the introductory notions presented in this chapter are not sufficient, readers may refer to [19, 38, 61, 86].

The remainder of the chapter is organized in the following manner. In Section 4.2, we introduce the concept of many-sorted language and its constituents. Section 4.3 introduces operators for composing and manipulating assertions. Section 4.4 discusses the algebraic properties of the operators on assertions. Section 4.5 concludes the chapter.

## 4.2 Many-Sorted Language

We introduce the basic concepts underlying a many-sorted language. Many-sorted logic which is an extension of first-order logic plays an important role in many branches of computer science in general and in program specification and verification in particular. The important paradigms sustaining this logic include its signature, its alphabet, its structures, formulas, and terms.

### 4.2.1 Signature

A *many-sorted signature* is a 3-tuple $(\mathcal{N}, \mathcal{F}, ar)$ consisting of:

- a non-empty set of sorts $\mathcal{N}$.
- A set $\mathcal{F}$ of *operation symbols.* For each $n \geq 0$ and each $n$-tuple $(i_1, \cdots, i_n)$ of sorts, there exists a possible empty set of $n$-ary predicate symbols, each of which is said to be of sort $(i_1, \cdots, i_n)$.
- an arity function $ar$ defined from $\mathcal{F}$ to the set of positive integers $\mathbb{N}$.

The arity of an operation symbol $f$ is $ar(f)$. If $ar(f) = n$, $f$ is called an *n-ary* operation symbol. We use the words nullary, unary, binary, and ternary for 0-ary, 1-ary, 2-ary, and 3-ary respectively. The sort $i \in \mathcal{N}$ is a composed sort if it can be written in the form $(i_1, \cdots, i_n)$ where $i_1, \cdots, i_n$ are some non-composed sorts in $\mathcal{N}$. A sort that is not composed is also called an individual sort.

We allow operation symbols to have multiple sorts, called many-sorted operation symbols. This may be compared to operation overloading in object-oriented programming languages. For any operation symbol $f$ of sorts $i_1$ and $i_2$ $(i_1 \neq i_2)$, two operation symbols $f_1$ and $f_2$ (of sort $i_1$ and $i_2$ respectively) must be assumed; $f$ is, therefore, understood as behaving as $f_1$ given an element of sort $i_1$ and $f_2$ given an element of sort $i_2$.

It is required that:

- any two of the above sets be disjoint if they are different from each other;
- no symbol is a sequence of other symbols;
- the sort $\mathbb{B}$ be a sort in $\mathcal{N}$,
- the set of operation symbols includes at least the four symbols $\neg$, $\vee$, $=$, and $\in$ which are for negation, disjunction, equality, and membership respectively. The

first is a binary operation symbol while the others are ternary operation symbols. The operation symbols $\neg$ and $\vee$ are of sorts $(\mathbb{B}, \mathbb{B})$ and $(\mathbb{B}, \mathbb{B}, \mathbb{B})$ respectively. The operation symbol $=$ is a many-sorted operation symbol defined on any sort; if $i$ is a sort in $\mathcal{N}$, $(\mathbb{B}, i, i)$ is a sort of $=$. The operators $\vee$, $=$, and $\in$ are infix operators. That is, instead of writing $\vee\ a\ b$, we will write $a \vee b$.

- The membership operation symbol $\in$ is also a many-sorted operation symbol such that if $i_1, \cdots i_n$ are individual sorts in $\mathcal{N}$ and $(\mathbb{B}, i_1, \cdots, i_n)$ is a composed sort in $\mathcal{N}$ then $(\mathbb{B}, i_1, \cdots, i_n, (i_1, \cdots, i_n))$ is a sort of $\in$.

### 4.2.2 Variable

Given a many-sorted signature $(\mathcal{N}, \mathcal{F}, ar)$, we define a set of variables $\mathcal{V}$ such that each of them has its own sort $i \in \mathcal{N}$. We write $x : \Sigma$ or $x \in \Sigma$ to stipulate that the variable $x$ is of sort $\Sigma$ since the sort of $x$ is also its type. The set of variables of sort $i$ is denoted $\mathcal{V}_i$.

In particular, there exist variables of individual sorts and variables of composed sorts. A variable is not allowed to have two different sorts; that is, if $x_1$ is of sort $\Sigma_1$, $x_2$ of sort $\Sigma_2$, and $\Sigma_1$ is different from $\Sigma_2$, then, it must be the case that $x_1$ is different from $x_2$.

In the style of VDM [75, 130, 102], we admit some special variables called hooked variables. For any unhooked variable $x \in \mathcal{V}$, there exists a hooked variable $\overleftarrow{x} \in \mathcal{V}$. We anticipate and intuitively justify the use of hooked variables as a means for comparing the results of operations at different states. Some use dynamic logic for this purpose [54].

### 4.2.3 Alphabet

The alphabet of a many-sorted language with signature $(\mathcal{N}, \mathcal{F}, ar)$, consists of:

1. The elements of $\mathcal{F}$; in particular, $\vee$, $\neg$, $\in$, and $=$ are in the alphabet.

2. The quantifiers: $\forall$, $\exists$.

3. Punctuation symbols: the opening parenthesis (, the closing parenthesis ), and the comma.

4. Variables as defined in the previous subsection. This includes variables of individual sorts, variables of composed sorts, hooked variables, and unhooked variables.

### 4.2.4 Expressions: Terms and Formulas

We consider the alphabet $\mathcal{A}$ of a many-sorted language and the set $\mathcal{A}^*$ of finite strings over $\mathcal{A}$. The set of *well-formed terms* (wft) (or simply *terms*) is the smallest subset $\mathcal{T}$ of $\mathcal{A}^*$ such that:

1. if $x$ is a variable, then $x \in \mathcal{T}$,
2. if $c$ is a nullary operation symbol then $c \in \mathcal{T}$,
3. if $f$ is an $n$-ary operation symbol of sort $(i_0, i_1, \cdots, i_n)$ and $t_1, \cdots, t_n \in \mathcal{T}$ are terms of sorts $i_1, \cdots, i_n$ respectively, then $f(t_1, \cdots, t_n) \in \mathcal{T}$ and is of sort $i_0$.

The set $\mathcal{W}$ of well-formed formulas (wff) over the many-sorted alphabet $\mathcal{A}$ is the smallest subset of $\mathcal{A}^*$ such that:

1. if $\alpha$ is an expression of sort $\mathbb{B}$, then $\alpha \in \mathcal{W}$,
2. If $\alpha \in \mathcal{W}$, $x$ is a variable of sort $\Sigma$, then $\forall x \cdot \alpha$ and $\exists x \cdot \alpha$ are well-formed formulas.
3. We further require that a variable can not be both quantified and unquantified in a formula. That is, we do not support formulas of the form $x = a \wedge \forall x \cdot x > y$.

### 4.2.5 Hooked Formulas

A hooked formula $\overleftarrow{\alpha}$ denotes the formula obtain from $\alpha$ by replacing any occurrence of a free variable $v$ with its hooked version $\overleftarrow{v}$.

### 4.2.6 Occurrence

A particular variable $x$ may appear several times in the string of symbols which constitute a formula; each of these is called an occurrence of $x$.

### 4.2.7 Basic Sorts and Operations

In addition to the sort $\mathbb{B}$, the many-sorted languages in this thesis are supposed to include the sort of events *Event* that we do not further define. The many-sorted logic used in this thesis will also include other operation symbols such as $|$, $;$, $\|$. Similarly to other sorts, these operation symbols will be presented progressively.

### 4.2.8 Scope of a quantifier

If (QX)F is a formula where Q is a quantifier, then F is the scope of Q.

### 4.2.9 Free and Bound Variables

A variable $x$ is free in the formula $Q$ if $x$ is not in the scope of a quantifier. A variable that is not free in $Q$ is said to be bound.

A variable is not allowed to be bound and free in the same formula.

### 4.2.10 Closed Formula

A formula is closed if it does not contain free occurrences of variables.

### 4.2.11 Structure

To assign meaning to formulas of a many-sorted language, we need to know in which set to interpret the variables and how to assign meanings to operation symbols. This is done by means of a structure.

**Definition 1** *A structure $\pi$ of a many-sorted language $\mathfrak{L}$ is a function that maps*

- *any sort $t$ in $\mathfrak{L}$ to a nonempty set of values $\pi(t)$ called the carrier of $t$;*
- *any operation symbol $f$ of sort $(t_0, \cdots t_n)$ to a total function $\pi(f)$ called the interpretation of $f$ such that:*

$$\pi(f) : \pi(t_1) \times \cdots \times \pi(t_n) \rightarrow \pi(t_0).$$

In particular, it is required that the carrier of $\mathbb{B}$ be the set of truth values $\{\mathsf{true}, \mathsf{false}\}$.

### 4.2.12 Valuation

While a structure determines how to interpret sorts, and operation symbols, a valuation assigns values to variables.

Formally, a valuation $\Omega_\pi$ in a structure $\pi$ is a mapping of all variables to values in the structure. Any variable $v$ of sort $t$ is mapped to an element of $\pi(t)$. It is clear that valuations are structure dependent. We will, however, simply write $\Omega$ when the structure is obvious from the context.

### 4.2.13 State

A state is a one-to-one mapping of all unhooked variables to values. For any state, it is required that each variable be mapped to a value of the same sort. Given the state $s$ and the variable $v$, the value of $v$ in this state will be denoted $s(v)$.

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

### 4.2.14 Interpretation

We have seen how to interpret sorts, operation symbols, and variables; the issue of interpreting expressions is still open. This is done by a recursive application of valuations. Fix a structure $\pi$ and a valuation $\Omega_\pi$. We define an interpretation $\overline{\Omega}_\pi$ of expressions as follows:

- For each variable $x$, $\overline{\Omega}_\pi(x) = \Omega_\pi(x)$,
- For any n-ary function symbol $f$ of sort $(i_1, \cdots, i_{n+1})$ and $n$ terms $t_1, \cdots, t_n$ of sort $i_1, \cdots, i_n$ respectively, $\overline{\Omega}_\pi(f(t_1, \cdots, t_n)) = \pi(f)(\overline{\Omega}_\pi(t_1), \cdots, \overline{\Omega}_\pi(t_n))$.

In particular:

- If $A$ is a wff, we write $\Omega \models_\pi A$ to say that $\overline{\Omega}_\pi(A) = \mathsf{true}$
- For any term $t$, $\Omega \models_\pi (t_1 = t_2)$ iff $\overline{\Omega}_\pi(t_1) = \overline{\Omega}_\pi(t_2)$;
- If $t_1, \cdots, t_n$ are terms of sort $i_1, \cdots, i_n$ and $X$ is a variable of sort $(i_1, \cdots, i_n)$, then $\Omega \models_\pi (t, \cdots, t_n) \in X$ holds iff $(\overline{\Omega}_\pi(t_1), \cdots, \overline{\Omega}_\pi(t_n)) \in \overline{\Omega}_\pi(X)$ holds;
- If $P$ is an $n$-ary predicate of sort $(i_1, \cdots, i_n)$ and $t_1, \cdots, t_n$ are terms of sorts $i_1, \cdots, i_n$ respectively, then $\Omega \models_\pi P(t_1, \cdots, t_n)$ iff $\pi(P)(\overline{\Omega}_\pi(t_1), \cdots, \overline{\Omega}_\pi(t_n)) = \mathsf{true}$;
- For any wff $\phi$, $\Omega \models_\pi (\neg\phi)$ iff $\Omega \models_\pi \phi$ does not hold;
- For any two wffs $\phi_1$ and $\phi_2$, $\Omega \models_\pi (\phi_1 \wedge \phi_2)$ iff $\Omega \models_\pi \phi_1$ and $\Omega \models_\pi \phi_2$ hold;

- For any two wffs $\phi_1$ and $\phi_2$, $\Omega \models_\pi (\phi_1 \vee \phi_2)$ iff either $\Omega \models_\pi \phi_1$ or $\Omega \models_\pi \phi_2$ or both hold.

As usual, $\phi_1 \Rightarrow \phi_2$ is introduced as a shortcut for $(\neg\phi_1) \vee (\phi_2)$ and $(\phi_1 \Longleftrightarrow \phi_2)$ is a shortcut for $((\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1))$.

### 4.2.15 Validity

We now discuss the validity of wffs.

The wff $A$ is valid in the structure $\pi$ (denoted as $\models_\pi A$) iff for any valuation $\Omega_\pi$ defined in $\pi$, $\Omega \models_\pi A$ holds. The wff $A$ is valid (denoted as $\models A$) iff it is valid in any structure and for any valuation.

Similarly, if $s_1$ and $s_2$ are two states and $A$ is a formula, then, $(s_1, s_2) \models_\pi A$ is valid iff $\Omega \models_\pi A$ is valid when $\Omega_\pi$ represents the valuation constructed such that for any unhooked variable $v$, $\Omega_\pi(v) = s_2(v)$ and for any hooked variable $\overleftarrow{v}$, $\Omega_\pi(\overleftarrow{v}) = s_1(v)$.

We may also write $s \models_\pi A$ if $A$ has no occurrence of hooked variable.

### 4.2.16 Unary and Binary Assertions

In the remainder of the thesis, we will use the term *assertion* to denote any wff formula. That is, an expression that evaluates to a boolean value is an assertion. The justification for this denomination is that such formulas will be used to assert that they should be true when the control flow of a program reaches a given point [61].

An assertion can be viewed as a relation on states since it defines the set of pairs $(s_1, s_2)$ such that $(s_1, s_2) \models_\pi A$ is valid.

If $A$ has no occurrence of hooked variable, $A$ may be thought of as the set of states $s$ such that $s \models_\pi A$ is valid. Such assertions are, therefore, called unary assertions. If $A$ has occurrences of hooked variables, $A$ is said to be a binary assertion. Note, however, that a unary assertion can be transformed to a binary assertion by, e.g. conjoining it with an assertion such as $\overleftarrow{x} = \overleftarrow{x}$ for any variable $x$. It will, therefore not be surprising that we sometimes use unary assertions as binary assertions.

## 4.3 Operations on Assertions

We introduce some operations for manipulating assertions.

### 4.3.1 Substitution

Given the wff $A$, the set of expressions $(r_i)_{i\in[1,n]}$, the set of variables $(v_i)_{i\in[1,n]}$ such that any $v_i$ is of the same sort as the corresponding $r_i$, the expression $A(v_1/r_1, \cdots, v_n/r_n)$ denotes the expression obtained from $A$ by simultaneously replacing any free occurrence of $v_i$ $(1 \leq i \leq n)$ with the corresponding $r_i$.

### 4.3.2 Skolemization

Skolemization is a technique for elimination of existential quantifiers in formulas. A formula $\exists x \cdot \phi$ is replaced with the formula $\phi[x/c]$ if the $c$ does not occur in $\phi$. The term $c$ is called *skolem constant.*

### 4.3.3 Identity Assertion

Given a set of variables $V$, $I_V$ denotes the binary assertion such that $(s_1, s_2) \in I_V$ iff $s_1 \overset{V}{=} s_2$. That is, the variables in $V$ are kept unchanged from the state $s_1$ to the state $s_2$.

### 4.3.4 Composing Assertions

We discuss the composition of tuples and assertions. The operator used for this purpose is denoted $|$.

The composition of two tuples $(t_1, \cdots, t_n)$ and $(s_1, \cdots, s_m)$ (denoted $(t_1, \cdots, t_n) \mid (s_1, \cdots, s_m)$) is defined iff $t_n = s_1$ and is constructed by catenating the two tuples obtained by removing $t_n$ and $s_1$, from $(t_1, \cdots, t_n)$ and $(s_1, \cdots, s_m)$ respectively. That is,

$$(t_1, \cdots, t_n) \mid (s_1, \cdots, s_m) = (t_1, \cdots, t_{n\text{-}1}, s_2, \cdots, s_m).$$

The composition of two assertions $A$ and $B$ (denoted $A \mid B$) is the assertion constructed by composing each tuple of $A$ with each tuple of $B$ if possible. There is no constrain on the arity of these two assertions.

In other terms, given two binary assertions $A$ and $B$, the tuple $(s_1, s_2)$ is in $A \mid B$ iff there exists some $s$ such that $(s_1, s)$ is in $A$ and $(s, s_2)$ is in $B$.

### 4.3.5 Transitive Closure

The transitive closure of an assertion $R$ (denoted $R^+$) is defined as the least binary relation on the set of states such that if $s_1$, $s_2$, and $s_3$ are states:

- $(s_1, s_2) \models_\pi R \Rightarrow (s_1, s_2) \models_\pi R^+$,
- $(s_1, s_2) \models_\pi R^+$ and $(s_2, s_3) \models_\pi R^+ \Rightarrow (s_1, s_3) \models_\pi R^+$.

### 4.3.6 Reflexivity and Transitivity

A binary assertion $A$ is reflexive iff any state $s$ is such that $(s, s) \models_\pi A$ holds.

On the other hand, the assertion $A$ is transitive iff for any three states $s_1$, $s_2$, and $s_3$, $(s_1, s_2) \models_\pi A$ and $(s_2, s_3) \models_\pi A$ results in $(s_1, s_3) \models_\pi A$.

### 4.3.7 Reflexive Transitive Closure

The reflexive transitive closure of an assertion, is obtained from its transitive closure by allowing not only a positive number of steps, but also zero step $R^* = (I_\vartheta \vee R)^+$.

## 4.4 Algebraic Properties

We now give some algebraic properties of the composition operator $\mid$ discussed in the previous section.

### 4.4.1 Associativity

The associativity criterion claims that it does not matter in which order the composition operator is applied. Formally, for any three binary assertions $E_1$, $E_2$, $E_3$, the following claim hold:

$$E_1 \mid (E_2 \mid E_3) \iff (E_1 \mid E_2) \mid E_3.$$

### 4.4.2 Idempotence

The idempotence law claims that if composing a binary assertion $R$ with itself results in nothing else than $R$, then $R$ is equal to its transitive closure. Formally, for any assertion $R$, the following assertions hold:

$$R \mid R \iff R \quad \text{iff} \quad R^+ \iff R \quad \text{and}$$
$$R \mid R \iff R \quad \text{iff} \quad R^* \iff (R \cup I_\vartheta).$$

### 4.4.3 Distributivity with Or

The sequential composition operator distributes left and right with $\vee$. That is, for any binary assertions $E_1$, $E_2$, and $E_3$,

$$E_1 \mid (E_2 \vee E_3) \iff (E_1 \mid E_2) \vee (E_1 \mid E_3) \text{ and}$$
$$(E_1 \vee E_2) \mid E_3 \iff (E_1 \mid E_3) \vee (E_2 \mid E_3).$$

### 4.4.4 Non Distributivity with And

Unlike with $\vee$, the sequential composition operator does not distribute with $\wedge$. The implication holds only in one direction. Assume three binary assertions $E_1$, $E_2$, and $E_3$, then,

$$E_1 \mid (E_2 \wedge E_3) \Rightarrow (E_1 \mid E_2) \wedge (E_1 \mid E_3) \text{ and}$$
$$(E_1 \wedge E_2) \mid E_3 \Rightarrow (E_1 \mid E_3) \wedge (E_2 \mid E_3).$$

### 4.4.5 Stability of Assertions

We introduce the notion of stability as in [136] and extend it to binary assertions. Let $R$ be a binary assertion and $Q$ be a binary or a unary assertion.

The assertion $Q$ is stable when $R$ iff one of the following formulas hold:

- $R \iff \mathsf{false}$,
- $Q \mid R \Rightarrow Q$.

The stability of unary assertion can also be defined as follows.

The unary assertion $Q$ is stable when $R$ iff one of the following formulas hold:

- $R \iff \mathsf{false}$,
- $\overleftarrow{Q} \wedge R \Rightarrow Q$.

Intuitively, the concept of stability allows characterizing assertions that are not affected by some transformation. We will simply write $Q$ **stable when** $R$.

### 4.4.6 Monotonicity of the Transitive Closure

For any two binary assertions $E_1$ and $E_2$, if $E_2$ follows from $E_1$ then so does the transitive closure of $E_2$ from that of $E_1$. Formally,

if $E_1 \Rightarrow E_2$ holds then $E_1^* \Rightarrow E_2^*$ and $E_1^+ \Rightarrow E_2^+$ also holds.

## 4.5 Summary

The chapter presented the symbols, the basic concepts, and the notation that we will use in the remainder of the thesis. We gave a brief introduction to the many-sorted logic in general. In particular, we did this such that the logic has a higher-order logic look. This allows us to introduce some constructs that would otherwise be higher-order constructs. We, hence, directly exploit the expressiveness of the many-sorted logic. The next chapter continues the construction of our logic. We add the sort of programs and operations for composing them.

# Chapter 5

# The Core Programming Language

## 5.1 Motivation

We define the LECAP programming language, a core programming language for constructing event-based applications. The LECAP language is a traditional while-parallel language extended with an event-announcement construct and later with method invocation. Despite its simplicity, the LECAP language is flexible enough to support many other methods of synchronization and communication such as semaphores, synchronous and asynchronous communications, and static and dynamic bindings. We give a detailed discussion about the expressiveness of the LECAP language in Chapter 17.

The definition of the LECAP language includes 3 main parts: 1) the definition of an event-based system, 2) the definition of the syntax of the language, and 3) the definition of the operational semantics of the language. The first part is discussed in the next section, the second part in discussed in Section 5.3. Section 5.4 introduces labeled transition systems that we use in Section 5.5 for defining the operational semantics of the LECAP programming language. Section 5.6 concludes the chapter.

## 5.2 Abstract Model for Event-Based Systems

Event-based systems take a variety of forms in practice. Their architectures vary from client/server to peer-to-peer styles. Consequently, the constituents of an event-based system exhibit different names and forms; for instance, a set of interacting entities may be called modules, components, programs, tasks, processes, objects, or actors [40]. At the abstract level, however, not all of these concepts are needed. This is reasonable and justified since the aim of a model is to abstract from unnecessary details and retain those concepts that determine the view that we want to have from the real system. An abstract model must, therefore, simultaneously be rich, flexible and simple. We have identified five concepts for an abstract event-based system: events, subscriptions, bindings, shared variables, interacting programs (consumers and producers).

A program—called producer—announces an event by sending it to the event-based infrastructure. Based on the binding which records what programs are interested in which events, the event-based infrastructure is responsible in invoking—also said triggering—the interested subscribers—also called consumers.

An event is a piece of data that may be published by a program (producer). The set of events that may be announced by the producers is defined by a sort $\mathbb{E}$ in our logic. Note that various typing techniques may be adopted at the concrete level. In Java for instance, the type event may be defined by means of an interface, an abstract class, a final class, or a simple normal class. In these cases, the concept of subtyping is directly supported. In the C programming language on the other side, an event may just be a `struct` construct. Our abstract model gives no properties and structures to events. The only operation we need on events is the matching of events to subscriptions that is done by the event-based infrastructure based on the binding.

The way for a subscriber to define its interest in receiving some kind of events is to specify a subscription which is a template categorizing a set of events.

The producers and consumers in an event-based system may not only interact through the event-announcement paradigm, but they may also share some variables. In practice such variables may be printers, database entries, or even real variables encapsulated inside a Java object.

### 5.2.1 Subscription

**Definition 2** *A subscription is a unary relation over events.*

Given a subscription $s$ and an event $e$ we naturally write $e \in s$ to say that the event $e$ is matched by the subscription $s$. The notation $e \in s$ is justified by the view of a subscription as a unary relation.

### 5.2.2 Binding

**Definition 3** *A binding is a map of some set of programs to some set of subscriptions.*

A binding states which events a program is interested in. If $\mathcal{B}$ is a binding and $z$ is a program in the domain of $\mathcal{B}$ (denoted $\mathsf{dom}\ \mathcal{B}$), then $\mathcal{B}(z)$ defines the subscription of $z$. Subscribing and unsubscribing are done by redefining this subscription.

We define $\mathit{subscribers}_{\mathcal{B}}(e) = \{z \in \mathsf{dom}\ \mathcal{B} \mid e \in \mathcal{B}(z)\}$ as the set of programs that are

subscribed to $e$. Note that $subscribers_{\mathcal{B}}(e)$ indeed depends upon the binding $\mathcal{B}$. We will simply write $subscribers(e)$ when the binding is obvious from the context. A precise definition of the concept of program is given in the next sections.

**Definition 4** *An event-based system is a tuple $(\vartheta, \mathcal{B})$ consisting of a binding $\mathcal{B}$ and a set of variables that programs in the domain in $\mathcal{B}$ share. Any variable accessed by a program in the binding $\mathcal{B}$ is required to be in $\vartheta$.*

## 5.3 Syntax of the LECAP Language

The LECAP language is a while-language augmented with the parallel, the synchronization, and the event publication constructs. Its syntax is defined as follows:

$$\begin{aligned} P ::= x := e \quad &| \; P_1; P_2 \; | \; \textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2 \textbf{ fi} \; | \; \textbf{while } b \textbf{ do } P \textbf{ od} \\ &| \; \{P_1 \| P_2\} \; | \; \textbf{announce}(e) \; | \; \textbf{skip} \; | \; \textbf{await } b \textbf{ do } P \textbf{ od}. \end{aligned}$$

Many of the constructs in this language are well-known traditional constructs: the assignment, the sequential composition of programs, the if construct, the while construct, and the skip construct.

In the assignment statement, $x$ represents a variable in the set of variables of the current event-based system. This variable is assigned the value of the expression $e$ which is supposed to be of the same type as $x$.

Although not obvious, the parallel and synchronization constructs are also well studied. The first models nonderministic interleaving of the atomic actions of $P_1$ and $P_2$. Synchronization and mutual exclusion are achieved by means of the await construct. We extend the semantics of the await construct to support the announcement of events.

The announce construct allows announcement of events. It is intended for the notification of the event-based system which in turn triggers some subscribers. In this construct, $e$ represents an expression of type *Event* whose free variables are all in the set of variables of the current event-based system. We will use the term application to denote a set of programs tied by means of some subscription-event announcement relationship.

### 5.3.1 Restrictions

To simplify the deduction rules, it is required that:

1. any assignment is such that the expression on the right side is of the same type as the variable on the left side;

Note that there is no restriction on programs that occur in an await statement. In particular, an await statement may include another await statement or an event announcement.

**Definition 5** *In the remainder of this thesis, we will assume that for any binding $\mathcal{B}$ and for any event e, the program* **skip** *is subscribed to e, i.e.* **skip** $\in$ *subscribers*$_{\mathcal{B}}(e)$. *The binding that has only the program* **skip** *subscribed to any event is called empty binding and denoted $\mathcal{B}_0$.*

This restriction on bindings allows us to present uniform rules without need to distinguish the cases of events with no subscribed method.

### 5.3.2 SUBPROGRAM

We say that a program $z_0$ is a subprogram of another program $z$ iff the latter can be written in one of the following forms:

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

We denote this as $z_0 \subseteq z$; we write $z_0 \not\subseteq z$ is $z_0$ is not a subset of $z$.

## 5.4 LABELED TRANSITION SYSTEM

The operational semantics of programs is commonly given in terms of a labeled transition system. We, therefore, recall the definition of this concept in this section.

A labeled transition system is a structure $(S, S_0, Act, \rightarrow)$ where:

- S is the set of configurations,

- $S_0$ is the set of initial configurations,
- Act is a set of action labels, and
- $\rightarrow$ is a transition relation such that $\rightarrow \subseteq S \times Act \times S$. Given a configuration and a label, the transition relation says what should be the next configuration to evolve into.

The execution of a program is modelled by capturing its behavior through configurations that include the program to be executed and the current assignment of values to variables. Transitions are either program transitions or environment transitions.

## 5.5 Semantics of the Lecap Language

We give the operational semantics of the LECAP programming language in the style of [3]. This style has been increasingly used for the definition of operational semantics (see [11, 119, 135, 120]). The semantics of the Lecap programming language is given relative to an event-based system $(\vartheta, \mathcal{B})$ and is centered around 3 concepts: transitions, configurations, computations.

### 5.5.1 Configurations

A configuration is a pair $\langle z,\ s\rangle$ composed of a program $z$ and a state $s$. The program $z$ may also be the empty program $\epsilon$.

### 5.5.2 Transitions

**Environment transition**

An environment transition $\xrightarrow{v}$ is the least binary relation on configurations such that the following rule holds.

$$\frac{}{\langle z,\ s_1\rangle \xrightarrow{v} \langle z,\ s_2\rangle.}$$

Environment transitions are allowed to modify only the state of the event-based system. They may, however, do so only for variables that do not occur in the test of **if**, **while**, and **await** instructions.

### Program transition

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

SEMANTICS OF SKIP

The program does nothing but terminates. The state is kept unchanged.

$$\frac{}{\langle \mathbf{skip},\ s\rangle \xrightarrow{i} \langle \epsilon,\ s\rangle}$$

SEMANTICS OF ASSIGNMENTS

The value of the expression $r$ is assigned to the variable $u$. $s[r/u]$ denotes the state obtained from $s$ by mapping the variable $u$ to the value of $r$ and leaving all other state variables unchanged.

$$\frac{}{\langle u := r,\ s\rangle \xrightarrow{i} \langle \epsilon,\ s[r/u]\rangle}$$

SEMANTICS OF EVENT ANNOUNCEMENT

The set of programs that subscribed to the event $e$ are invoked (triggered) and executed in parallel. The first case presents the case where the announcement is the last construct in the program.

$$\frac{s(x) = e \quad \mathit{subscribers}(e) = \{z_1, \cdots, z_n\}}{\langle \mathbf{announce}(x),\ s\rangle \xrightarrow{i} \langle \{z_1 \| \cdots \| z_n\},\ s\rangle}$$

If however, the announcement construct is not the last construct, the subscribers are executed in parallel with the remainder of the announcing program. The programs triggered by an event announced by the running program are part of this program and their transitions are internal transitions.

$$\frac{s(x) = e, \quad m, n \geq 0, \quad \mathit{subscribers}(e) = \{z_1, \cdots, z_n\}}{\langle \{^n\mathbf{announce}(x); z\}^m,\ s\rangle \xrightarrow{i} \langle \{\{z_1 \| \cdots \| z_n\} \| \{^n z\}^m\},\ s\rangle}$$

$$\frac{s(x) = e, \quad m, n \geq 1, \quad \mathit{subscribers}(e) = \{z_1, \cdots, z_n\}}{\langle \{^n\mathbf{announce}(x) \| z\}^m,\ s\rangle \xrightarrow{i} \langle \{\{z_1 \| \cdots \| z_n\} \| \{^n z\}^m\},\ s\rangle}$$

In this formula, $\{^n$ denotes a sequence of $n$ left braces $\{$. Similarly $\}^m$ denotes a sequence of $m$ right braces. It is important that the number of braces be taken into consideration since omitting some of them results in a malformed program. Let us take for instance the program $\{\mathbf{announce}(x); z_1 \| z_2\}$. The rule says that it evolves into $\{\{z_1(e) \| \cdots \| z_n(e)\} \| \{z_1 \| z_2\}\}$. Ignoring the first brace would however result in the following malformed program $\{\{z_1(e) \| \cdots \| z_n(e)\} \| z_1 \| z_2\}\}$.

### Semantics of Sequences

If the program $z_1$ terminates after one internal transition, the sequence composed of $z_1$ and $z_2$ evolves into $z_2$ after one internal transition.

$$\frac{\langle z_1,\ s_1\rangle \xrightarrow{i} \langle \epsilon,\ s_2\rangle}{\langle z_1; z_2,\ s_1\rangle \xrightarrow{i} \langle z_2,\ s_2\rangle}$$

If $z_1$ instead evolves into a program $z_3$ different from the empty program and $z_1$ does not start with an event announcement, then $z_1; z_2$ evolves into $z_3; z_2$.

$$\frac{\langle z_1,\ s_1\rangle \xrightarrow{i} \langle z_3,\ s_2\rangle \quad z_3 \neq \epsilon \quad z_1 \notin \{\mathbf{announce}(x); z,\ \mathbf{announce}(x)\}}{\langle z_1; z_2,\ s_1\rangle \xrightarrow{i} \langle z_3; z_2,\ s_2\rangle}$$

Note that it is necessary that $z_1$ does not start with an event announcement for the program to evolve this way. In fact, as shown by the semantics of the announce construct, $\mathbf{announce}(x); z; z_2$ instead evolves into $\{\{z_{e_1} \| \cdots \| z_{e_n}\} \parallel z; z_2\}$.

### Semantics of If

The semantics of the **if** construct is not difficult to understand. If the test holds, then $z_1$ is executed, otherwise, $z_2$ is executed.

$$\frac{s(b) = \mathsf{true}}{\langle \textbf{if } b \textbf{ then } z_1 \textbf{ else } z_2 \textbf{ fi},\ s\rangle \xrightarrow{i} \langle z_1,\ s\rangle}$$

$$\frac{s(\neg b) = \mathsf{true}}{\langle \textbf{if } b \textbf{ then } z_1 \textbf{ else } z_2 \textbf{ fi},\ s\rangle \xrightarrow{i} \langle z_2,\ s\rangle}$$

### Semantics of while

If the expression $b$ evaluates to true in the current state, then, the body of the while construct is executed and the while construct is re-executed. If the expression $b$ evaluates to false the execution of the loop terminates.

$$\frac{s(b) = \mathsf{true}}{\langle \textbf{while } b \textbf{ do } z \textbf{ od},\ s\rangle \xrightarrow{i} \langle z; \textbf{while } b \textbf{ do } z \textbf{ od},\ s\rangle}$$

$$\frac{s(\neg b) = \mathsf{true}}{\langle \textbf{while } b \textbf{ do } z \textbf{ od},\ s\rangle \xrightarrow{i} \langle \epsilon,\ s\rangle}$$

### Semantics of Concurrency

In the first case, the program $z_2$ terminates and the parallel composition of $z_2$ and $z_1$ evolves into $z_1$. Note that $z_2$ can not be an event announcement construct since we have assumed that at least the program **skip** is subscribed to any event.

$$\frac{\langle z_2,\ s_1\rangle \xrightarrow{i} \langle \epsilon,\ s_2\rangle}{\langle \{z_1 \parallel z_2\},\ s_1\rangle \xrightarrow{i} \langle z_1,\ s_2\rangle \quad \langle \{z_2 \parallel z_1\},\ s_1\rangle \xrightarrow{i} \langle z_1,\ s_2\rangle}$$

On the other hand, if $z_1$ evolves into $z_1'$, then, $\{z_1 \| z_2\}$ evolves into $\{z_1' \| z_2\}$.

$$\frac{\langle z_1,\ s_1\rangle \xrightarrow{i} \langle z_1',\ s_2\rangle \quad z_1' \neq \epsilon\ z_3 \quad z_1 \notin \{\mathbf{announce}(x); z,\ \mathbf{announce}(x)\}}{\langle \{z_1 \parallel z_2\},\ s_1\rangle \xrightarrow{i} \langle \{z_1' \parallel z_2\},\ s_2\rangle \quad \langle \{z_2 \parallel z_1\},\ s_1\rangle \xrightarrow{i} \langle \{z_2 \parallel z_1'\},\ s_2\rangle}$$

SEMANTICS OF AWAIT

The meaning of an await statement is not very clear when its body does not terminate [135]. When it, however, terminates the final state is required to satisfy the post-condition or to deadlock. Given that we are not interested (in this work) in non-terminating programs we can stipulate that any computation of an await-statement has a finite length.

The semantics of the await construct distinguishes two cases. In the first case, the program is executed in an atomic step and terminates; its internal transitions are not visible to the environment. This justifies the existence of the finite number of internal transitions that transform $\langle z_1, s_1\rangle$ into $\langle \epsilon, s_n\rangle$.

$$\frac{\begin{array}{l} s_1(b) = \mathsf{true} \quad z_n = \epsilon \\ \exists \langle z_1, s_1\rangle, \cdots, \langle z_n, s_n\rangle \cdot \forall 1 < k \leq n \cdot \langle z_{k\text{-}1}, s_{k\text{-}1}\rangle \xrightarrow{i} \langle z_k,\ s_k\rangle \end{array}}{\langle \mathbf{await}\ b\ \mathbf{do}\ z_1\ \mathbf{od},\ s_1\rangle \xrightarrow{i} \langle \epsilon,\ s_n\rangle}$$

In the second case, however, the program does not terminate, but deadlocks. The execution of the program enters the await-construct but never exits; there is no configuration the last configuration can evolve into.

$$\frac{\begin{array}{l} s_1(b) = \mathsf{true} \\ \exists \langle z_2, s_2\rangle, \cdots, \langle z_n, s_n\rangle \cdot \forall 1 < k \leq n \cdot \langle z_{k\text{-}1}, s_{k\text{-}1}\rangle \xrightarrow{i} \langle z_k,\ s_k\rangle \wedge \\ \forall z_{n+1} \cdot \neg \langle z_n, s_n\rangle \xrightarrow{i} \langle z_{n+1},\ s_{n+1}\rangle \end{array}}{\langle \mathbf{await}\ b\ \mathbf{do}\ z_1\ \mathbf{od},\ s_1\rangle \xrightarrow{i} \langle \mathbf{await}\ b\ \mathbf{do}\ z_1\ \mathbf{od},\ s_1\rangle}$$

There is no restriction on the body of the await-construct. In particular, it may contain an event announcement. The await-construct, therefore, limits the scope [51] of the state variables to those programs that are executed following the announcement of the event.

### 5.5.3 COMPUTATION

In addition to the state of the system that programs may read and update, they may also have local variables that are hidden such that environment transitions are not allowed to access them. We do not model this concept yet.

**Definition 6** *A configuration $c_1$ is disabled if there is no $c_2$ such that $c_1 \xrightarrow{i} c_2$.*

**Definition 7** *A computation is a possibly infinite sequence of environment and program transitions $\langle z_1, s_1\rangle \xrightarrow{l_1} \cdots \xrightarrow{l_{k-1}} \langle z_k, s_k\rangle \xrightarrow{l_k} \cdots$ such that the final configuration is disabled if the sequence is finite. A computation $\sigma$ is blocked if it is finite and the program of the last configuration is not $\epsilon$. A computation terminates iff it is finite and the program of the last configuration is $\epsilon$.*

The above operational semantics does not explicitly (but implicitly) discuss the case of events announced by the environment (including external events). The programs triggered by these events are part of the environment and their transitions are environment transitions.

### 5.5.4 NOTATION

Given a computation $\sigma$, then $Z(\sigma)$, $S(\sigma)$ and $L(\sigma)$ are the projections of $\sigma$ to sequences of programs, states and transition labels. For instance, if $\sigma = \langle z_1, s_1\rangle \xrightarrow{l_1} \langle z_2, s_2\rangle \xrightarrow{l_2} \langle z_3, s_3\rangle$, then:

- $Z(\sigma) = [z_1, z_2, z_3]$,
- $S(\sigma) = [s_1, s_2, s_3]$,
- $L(\sigma) = [l_1, l_2]$.

$Z(\sigma_k)$, $S(\sigma_k)$, $L(\sigma_k)$, and $\sigma_k$ respectively denote the k'th program, the k'th state, the k'th transition label and the k'th configuration. The number of configurations in $\sigma$ is denoted $len(\sigma)$. If $\sigma$ is infinite, then $len(\sigma) = \infty$.

The operational semantics given above shows that the behavior of a program depends upon the binding of the underlying event-based system. Consequently, the set of computations of a program that announces some event only has a meaning iff a binding is assumed. We, therefore, give the following definition:

**Definition 8** *Let $(\vartheta, \mathcal{B})$ be an event-based system and $z$ a program. The set of computations of $z$ in the binding $\mathcal{B}$ (denoted $cp[z, \mathcal{B}]$) is the set of computations $\sigma$ such that $Z(\sigma_1) = z$ and obtained by executing $z$ in the context of the event-based system $(\vartheta, \mathcal{B})$.*

## 5.6 SUMMARY

The chapter presented a formal definition of the LECAP programming language, a core programming language for programs that announce events.

The definition of this language includes three main concepts: the event-based system, the syntax of the language and its operational semantics. An event-based system was defined as a tuple of a set of variables shared by programs in the event-based system and the binding that binds programs to subscriptions. In addition to well-known constructs such as assignment, iteration, and the parallel constructs, the syntax of the LECAP programming language includes a construct for the announcement of events. The semantics of the LECAP programming language is also given. In particular, the execution of the announcement construct consists of triggering the set of subscribers and executing them concurrently with the remainder of the announcing program.

Other researchers defined a formal semantics for languages that announce events [36, 35]. In these semantics, the event-based system includes a set of events. We have rather included a sort of events in the language of the logic. This solves the issue of whether the set of events must be a finite set, a static set, or a dynamic set. In particular, typing and subtyping can be applied on this sort as on any type. These semantics are also different from ours; the semantics of the announce construct is given in terms of an event infrastructure. Finally, these semantics include neither the parallel construct nor the synchronization and mutual exclusion constructs.

# Chapter 6

# Specification of Event-Based Applications

## 6.1 Motivation

A key requirement for the verification and analysis of software systems properties is the formulation of their behaviors in a precise manner. Formal specifications are the current mean for expressing software requirements precisely. They can be unequivocally understood and acted upon by all software engineers involved in the production of a software system [5]. In fact, formal specifications are not only used for the verification of software properties but may also serve as documentation tools, as contracts, as communication means among stakeholders (e.g. clients, customers, designers, testers, implementers, maintenance engineers), as oracles for black box testing, or as basis for the derivation of test cases [134]. A formal framework for the design of software systems such as LECAP must, therefore, include a formal notation that permits the specification of these systems. The aim of this chapter is to fulfill this requirement.

A formal specification technique must include three building blocks: a syntax, a semantics, and a satisfaction relation. The latter expresses the relation between the first and the second which are called specification and specificand respectively. We discuss these concepts for event-based systems; we provide an answer to each of the following questions:

- How to specify an event-based application? In other terms, what are the constituents of the specification of an event-based application?
- Given the formal specification of an event-based application, what is its semantics, what does a specification mean?
- What are the expected uses of such specifications? In particular, can they be used for the verification of software systems properties? can they be used for the different uses that are generally recognized for formal specifications? e.g. can they be used for the top-down development of systems?

- Is the specification of event-announcement adequately supported?

These questions are justified since, as shown in Chapter 3, techniques for the formal specification of event-based applications have been proposed where many of these questions are open.

Though the aim of this chapter is to answer these questions, we do not provide a specification language such as VDM [130], Z [33], B [74], Alloy [70], or RSL [129], but merely an abstract specification language consisting of assertions that can be defined in any many-sorted language that extends the one presented in Chapter 4 by defining more sorts. For instance, one may extend this language with the sort of characters, the sort of strings, the sort of reals, or the sort of naturals. Such extensions, however, have no impact on our logic; we can, therefore, say that our framework is notation independent.

The remainder of the chapter is organized as follows. The next section explains the structure of specifications in event-based systems. Section 6.3 defines the concept of behavioral specification and discusses the related satisfaction relation. This concept is extended in Chapter 6.4 to include more kinds of behavioral specifications. Section 6.5 introduces structural specifications that are used for the specification of components in event-based applications. Section 6.6 summarizes the chapter and discusses the answer to the above questions.

## 6.2 Structure of Specifications

The aim of this section is to informally introduce behavioral and structural specifications and justify their need. This is done in the light of the software development process that we envision and that we already presented in Chapter 1. If we want to construct an application that satisfies the requirements $\phi_1, \cdots, \phi_n$, we must follow the following steps (depicted on Figure 1.2):

1. Designing an architecture that identifies the components necessary for constructing the application.

2. Developing the formal specifications $S_1, \cdots, S_m$ of these components and verifying some local properties about these specifications.

3. Composing the specification of the whole application starting with the specification of the components and verifying some global properties.

4. Refining the specifications $S_1, \cdots, S_m$ to some implementations $I_1, \cdots, I_m$.

The process clearly distinguishes the specifications of the components from those of the whole application. The first are based on an undefined binding and the second are based on bindings that reflect the desired application architecture.

LECAP is an approach in which the development of a system is done starting from that of the parts. Ideally, such parts must be independent from each other. In this case, when developing the parts (components) of a software system, the developers are not aware of the existence of other parts with which their components will interact. In the event-based terminology, this means that when announcing an event, the publisher is not aware of the existence of the subscriber. The binding of methods to events is therefore undefined when the components are developed.

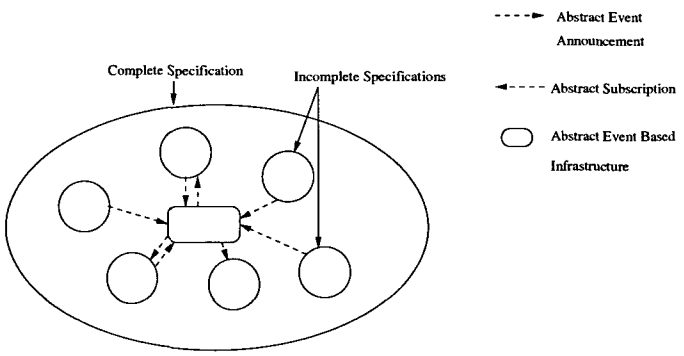


Figure 6.1: Specifications in Abstract Event-Based Systems

These concepts are captured in Figure 6.1; the specification of an application is composed of that of its components, the binding, and the abstract event-based infrastructure. On the other hand, the binding is represented in the figure through the subscription and the event-announcement arrows.

This structuring brings the compositionality and the loose coupling of the event-based architectural style to the abstract level: specifications of components may be developed individually and put together by means of the abstract binding.

## 6.3 BEHAVIORAL SPECIFICATIONS

This kind of specification is used for capturing the requirements of an event-based application. A behavioral specification is recognized by the binding which embodies the architecture of the application. Such a specification is not intended to be used for the top-down development of the application, but only for the verification of its global properties.

### 6.3.1 DEFINITION

**Definition 9** *A behavioral specification is a formula* $(\vartheta, \mathcal{B}) ::(P, R, G, E)$*, where:*

- *the* pre-condition $P$ *is a unary assertion while the* rely-condition $R$*, the* guar-condition $G$ *and the* post-condition $E$ *are binary assertions,*
- $(\vartheta, \mathcal{B})$ *is an event-based system,*
- *any free variable occurring in* $P$*,* $R$*,* $G$*, or* $E$ *is an element of* $\vartheta$*,*
- $P$ *is stable when* $R$*,*
- $E$ *is stable when* $R$*.*

A behavioral specification is essentially composed of three parts: the event-based system, the assumptions, and the commitments.

A program is not intended to work in all environments, but in those which satisfy its assumptions: the pre-condition $P$ and the rely-condition $R$. By the pre-condition, it is required that any program which satisfies this specification should only be started in states satisfying $P$. On the other hand, the rely-condition characterizes the transitions that may be done by the environment. A rely-condition is a binary assertion as it compares the current state with the previous state.

Our logic requires that any behavioral specification be such that $P$ and $E$ are stable when $R$. This simplifies the construction rules without reducing the expressiveness of the logic.

A program must guarantee something; this is captured by its commitments which are the guarantee and the post-conditions. By the guarantee condition, a program commits to perform only transitions that satisfy a certain assertion, namely its guar-condition. The program also commits to terminate in a state where the post-condition holds.

This way of specifying the behavior of systems is not new. In fact, pre- and post-conditions which are some kinds of assumption/commitment specifications were already used by Hoare [67]. Assumption/commitments for the verification of concurrent systems were also used by Francez and Pnuelli [53] and many other researchers. Jones [77] was the first to use this technique for the stepwise construction of software systems. We extend his approach to the stepwise construction of event-based applications.

We recall that this thesis is only concerned with partial correctness. That is, we want to construct programs that satisfy their post-conditions when they terminate.

### 6.3.2 External

**Definition 10** *Given a specification* $(\vartheta, \mathcal{B}) :: (P, R, G, E)$*, then,* $ext[(\vartheta, \mathcal{B}), P, R]$ *denotes the set of computations* $\sigma$ *such that the following conditions hold:*

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

The definition characterizes external computations which are computations of programs executed in environments that respect the assumptions $P$ and $R$. That is, computations such that the initial state satisfies the pre-condition and any environment transition that changes a state variable satisfies the rely-condition $R$.

Note that the environment may also announce some events that will result in the invocation of some other programs. The set of programs triggered by such events are part of the environment; their transitions are, therefore, also environment transitions, and hence, are also required to satisfy the rely-condition. On the other hand, the triggered programs are also part of the environment of each other. They are consequently also required to satisfy the rely-condition of each other.

### 6.3.3 Internal

**Definition 11** *Given a specification* $(\vartheta, \mathcal{B}) :: (P, R, G, E)$*, then,* $int[(\vartheta, \mathcal{B}), G, E]$ *denotes the set of computations* $\sigma$ *such that the following conditions hold:*

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

An internal computation is a finite computation that terminates in a state satisfying the post-condition $E$ and is such that any of its program transition (transition labelled with $i$) changing some state variable satisfies the guar-condition $G$.

**Definition 12** *A behavioral judgment is a formula* $z$ *sat* $(\vartheta, \mathcal{B}) :: (P, R, G, E)$ *where:*

- $z$ *is a program,*

- $(\vartheta, \mathcal{B}) :: (P, R, G, E)$ *is a behavioral specification,*
- *any variable accessed by* $z$ *is in the set of variables* $\vartheta$ *of the event-based system.*

Remember that we already required that for any specification $(\vartheta, \mathcal{B}) :: (P, R, G, E)$ any free variable occurring in the definition of $P$, $R$, $G$, or $E$ must be an element of $\vartheta$.

### 6.3.4 SATISFACTION

**Definition 13** *A behavioral judgment* $z$ *sat* $(\vartheta, \mathcal{B}) :: (P, R, G, E)$ *is valid iff*

$$cp[z, \mathcal{B}] \cap ext[(\vartheta, \mathcal{B}), P, R] \subseteq int[(\vartheta, \mathcal{B}), G, E].$$

*This is denoted as* $\models_\pi z$ *sat* $(\vartheta, \mathcal{B}) :: (P, R, G, E)$. *We also say that the program* $z$ *satisfies the specification* $(\vartheta, \mathcal{B}) :: (P,\ R,\ G,\ E)$.

Informally, to show that a judgment $z$ *sat* $(\vartheta, \mathcal{B}) :: (P,\ P,\ G,\ E)$ is valid, one proves that any terminating computation of $z$ that starts in a state satisfying $P$ and is executed in an environment whose interference satisfies $R$ has a final state satisfying $E$ and any of its program transitions changing the state variables satisfies $G$. A program that satisfies its specification satisfies its commitments if the environment satisfies the assumptions.

**Definition 14** *The program* $z$ *announces no event (denoted as* $events(z) = \{\}$*) iff* **announce** *is not a subprogram of* $z$.

This definition is, in fact, too strong. A program such as

**if** false **then announce**$(e)$ **else skip fi**

announces no event, but is excluded by our definition. This definition is, however, simple and easy to verify. The most important is that no program exists that really announces an event and that is not ruled out by our definition. If $z$ announces an event, then $z$ must have a subprogram of the form **announce**$(x)$.

## 6.4 Extended Behavioral specifications

We have found it indispensable to support specifications in the style of process algebra languages such as CSP and CCS. That is, we allow specifications of the form $(\vartheta, \mathcal{B})$ : $:\{S_1 \| S_2\}$, $(\vartheta, \mathcal{B})::$**if** $b$ **then** $S_1$ **else** $S_2$ **fi**, and $(\vartheta, \mathcal{B})::S_1; S_2$ where $(\vartheta, \mathcal{B})::S_1$ and $(\vartheta, \mathcal{B})::S_2$ are also behavioral specifications.

### 6.4.1 Definitions

**Definition 15** *Extended behavioral specifications are defined recursively. If we assume that the formulas $(\vartheta, \mathcal{B})::S_1$ and $(\vartheta, \mathcal{B})::S_2$ are behavioral specifications (including extended ones) on the same event-based system $(\vartheta, \mathcal{B})$ and any variable occurring free in $b$ is in $\vartheta$, then, the formulas $(\vartheta, \mathcal{B})::S_1; S_2$, $(\vartheta, \mathcal{B})::\{S_1 \| S_2\}$, and $(\vartheta, \mathcal{B})::$**if** $b$ **then** $S_1$ **else** $S_2$ **fi** are extended behavioral specifications.*

A specification of the form $(\vartheta, \mathcal{B})::$**if** $b$ **then** $S_1$ **else** $S_2$ **fi** is called an extended behavioral conditional specification. A specification of the form $(\vartheta, \mathcal{B}) : :S_1; S_2$ is called an extended behavioral sequential specification and a specification of the form $(\vartheta, \mathcal{B}) : :S_1 \| S_2$ is called an extended behavioral parallel specification.

**Definition 16** *An extended behavioral judgment is a formula $z$ sat $(\vartheta, \mathcal{B}) : :S$ consisting in a program $z$ and an extended behavioral specification $(\vartheta, \mathcal{B}) : :S$ such that any variable accessed by $z$ is an element of $\vartheta$.*

**Definition 17** *The program $z_1$ behaves as the program $z_2$ in the event-based system $(\vartheta, \mathcal{B})$ (denoted as $z_1$* **behaves as** *$z_2$* **in** *$(\vartheta, \mathcal{B})$) iff for any behavioral specification of the form $(\vartheta, \mathcal{B}) : :(P,\ R,\ G,\ E)$, if $z_2$ sat $(\vartheta, \mathcal{B}) : :(P,\ R,\ G,\ E)$ is a valid judgment, then, $z_1$ sat $(\vartheta, \mathcal{B}) : :(P,\ R,\ G,\ E)$ is also a valid judgment.*

The definition is quiet clear. A program $z_1$ behaves as the other one $z_2$ in the binding $(\vartheta, \mathcal{B})$ if the first satisfies any behavioral specification that the second satisfies.

### 6.4.2 Satisfaction

We now give meanings to extended behavioral specifications.

**Definition 18** *A program $z$ satisfies the extended behavioral specification $(\vartheta, \mathcal{B})::S_1; S_2$ iff three programs $z_1$, $z_2$ and $z_3$ exist such that the following hold:*

- $z_1$ *sat* $(\vartheta, \mathcal{B}_0) :: S_1$ *is a valid judgment,*
- $z_2$ *sat* $(\vartheta, \mathcal{B}_0) :: S_2$ *is a valid judgment,*
- $z_3 = z_1; z_2$,
- $events(z_3) = \{\}$, *and*
- $z$ **behaves as** $z_3$ **in** $(\vartheta, \mathcal{B})$.

The simplest example of a program that satisfies the specification $(\vartheta, \mathcal{B}) : S_1; S_2$ is the program $z_3$ that is the sequential composition of the programs $z_1$ and $z_2$ satisfying $(\vartheta, \mathcal{B}) : S_1$ and $(\vartheta, \mathcal{B}) :: S_2$ respectively. Any other program that behaves as $z_3$ in the event-based system $(\vartheta, \mathcal{B})$ also satisfies $(\vartheta, \mathcal{B}) :: S_1; S_2$. For instance, $z_1$; **skip**; $z_2$, $z_1$; $v := v$; $z$; **skip**.

**Definition 19** *A program $z$ satisfies the extended behavioral specification $(\vartheta, \mathcal{B}) :: \{S_1 \| S_2\}$ iff three programs $z_1$, $z_2$, and $z_3$ exist such that:*

- $z_1$ *sat* $(\vartheta, \mathcal{B}_0) :: S_1$ *is a valid judgment,*
- $z_2$ *sat* $(\vartheta, \mathcal{B}_0) :: S_2$ *is a valid judgment,*
- $z_3 = \{z_1 \| z_2\}$,
- $events(z_3) = \{\}$, *and*
- $z$ **behaves as** $z_3$ **in** $(\vartheta, \mathcal{B})$.

**Definition 20** *A program $z$ satisfies the extended behavioral specification $(\vartheta, \mathcal{B}) ::$* **if** $b$ **then** $S_1$ **else** $S_2$ **fi** *iff three programs $z_1$, $z_2$, and $z_3$ exist such that:*

- $z_1$ *sat* $(\vartheta, \mathcal{B}_0) :: S_1$ *is a valid judgment,*
- $z_2$ *sat* $(\vartheta, \mathcal{B}_0) :: S_2$ *is a valid judgment,*
- $z_3 =$ **if** $b$ **then** $z_1$ **else** $z_2$ **fi**,
- $events(z_3) = \{\}$, *and*
- $z$ **behaves as** $z_3$ **in** $(\vartheta, \mathcal{B})$.

## 6.5 Structural Specifications

The previous concept of behavioral specification does not support announcement of events. And, in fact, announcement of events is not required when verifying the properties of an application. Instead, the announcement of events is replaced with triggered subscribers. For the development of components, however, event announcements must indeed be taken into consideration. We introduce a new kind of specification called structural specification. Such specifications are characterized by the missing binding. One does not know yet with which other components a component will communicate.

Similarly to process algebra languages such as CSP [68] and CCS [89], we allow specifications to take the forms $(\vartheta, \mathcal{B}) ::\{S_1 \| S_2\}$, $(\vartheta, \mathcal{B}) ::$ **if** $b$ **then** $S_1$ **else** $S_2$ **fi**, $(\vartheta, \mathcal{B}) S_1; S_2$. As extended behavioral specifications, this kind of specifications allows specifying not only the behaviors of the specified program, but also make obvious which components this program may be composed of.

### 6.5.1 Definitions

We introduce the abstract announcement construct which allows specification of event announcements at the abstract level.

**Definition 21** *We assume a set of variables $\vartheta$. A structural specification is defined recursively.*

- *If exp is an expression that evaluates to an event, then, $\vartheta ::$* **announce**$(exp)$ *is a structural specification iff any variable occurring free in the definition of exp is in $\vartheta$.*
- *$\vartheta ::(P,\ R,\ G,\ E)$ is a structural specification iff:*
  - *$P$ is a unary assertion and $R$, $G$, and $E$ are binary assertions whose free variables are in $\vartheta$, and*
  - *$P$ and $E$ are stable when $R$.*
- *If $\vartheta ::S_1$ and $\vartheta ::S_2$ are structural specifications on the same set of variables $\vartheta$ and any variable occurring free in $b$ is in $\vartheta$, then, $\vartheta ::S_1; S_2$, $\vartheta ::\{S_1 \| S_2\}$, and $\vartheta ::$* **if** $b$ **then** $S_1$ **else** $S_2$ **fi** *are also structural specifications.*

**Definition 22** *A structural judgment is a formula $z$ sat $\vartheta ::S$ consisting in a program $z$ and a structural specification $\vartheta ::S$ such that any variable accessed by $z$ is an element of $\vartheta$.*

**Definition 23** *The program $z_1$ behaves as the program $z_2$ (denoted as $z_1$* **behaves as** *$z_2$) iff for any event-based system $(\vartheta, \mathcal{B})$, $z_1$ behaves as $z_2$ in $(\vartheta, \mathcal{B})$.*

*That is, $z_1$* **behaves as** *$z_2$* **in** *$(\vartheta, \mathcal{B})$ for any event-based system $(\vartheta, \mathcal{B})$.*

The definition is also clear. The behavior similarity is now extended to any event-based system.

### 6.5.2 SATISFACTION

We now give some meanings to structural specifications. As one may guess from the name, this satisfaction relation constrains the structure of a program. We distinguish four cases of structural specification.

**Definition 24** *The program $z$ satisfies the specification $\vartheta ::(P,\ R,\ G,\ E)$ iff:*

- *$z$ $\underline{sat}$ $(\vartheta, \mathcal{B}_0) ::(P,\ R,\ G,\ E)$ is a valid judgment and*
- *$events(z) = \{\}$.*

In the absence of synchronization, the definition is equivalent to requiring that the program satisfies the given specification if the binding is replaced with any other binding.

Note that in general, $z$ $\underline{sat}$ $\vartheta ::(P,\ R,\ G,\ E)$ can not be derived from the behavioral judgment $z$ $\underline{sat}$ $(\vartheta, \mathcal{B}) ::(P,\ R,\ G,\ E)$. In the latter judgment $z$ may announce an event and let the behavior $(P,\ R,\ G,\ E)$ achieved by a subscriber while in the first case, $z$ announces no event, hence, achieves the behavior itself.

**Definition 25** *The program $z$;* **announce**$(exp_1)$ *satisfies the structural specification $\vartheta ::(P,\ R,\ G,\ E)$;* **announce**$(exp_2)$ *iff $z$ $\underline{sat}$ $\vartheta ::(P,\ R,\ G,\ E \wedge exp_1 = exp_2)$ is a valid judgment.*

The post-condition of the program $z$ determines the state in which the announce construct is invoked and, hence, the set of events that may be announced in this state.

**Definition 26** *A program $z$ satisfies the structural specification $\vartheta ::S_1; S_2$ iff three programs $z_1$, $z_2$ and $z_3$ exist such that the following hold:*

- *$z_1$ $\underline{sat}$ $\vartheta ::S_1$ is a valid judgment,*

- $z_2$ *sat* $\vartheta :: S_2$ *is a valid judgment,*
- $z_3 = z_1; z_2$, *and*
- $z$ **behaves as** $z_3$.

The simplest example of a program that satisfies the specification $\vartheta : S_1; S_2$ is the program $z_3$ that is the sequential composition of the programs $z_1$ and $z_2$ satisfying $\vartheta :: S_1$ and $\vartheta :: S_2$ respectively. Any other program that behaves as $z_3$ also satisfies $\vartheta :: S_1; S_2$.

**Definition 27** *A program $z$ satisfies the structural specification $\vartheta :: \{S_1 \| S_2\}$ iff three programs $z_1$, $z_2$, and $z_3$ exist such that:*

- $z_1$ *sat* $\vartheta :: S_1$ *is a valid judgment,*
- $z_2$ *sat* $\vartheta :: S_2$ *is a valid judgment,*
- $z_3 = \{z_1 \| z_2\}$, *and*
- $z$ **behaves as** $z_3$.

**Definition 28** *A program $z$ satisfies the structural specification $\vartheta$::***if** $b$ **then** $S_1$ **else** $S_2$ **fi** *iff three programs $z_1$, $z_2$, and $z_3$ exist such that:*

- $z_1$ *sat* $\vartheta :: S_1$ *is a valid judgment,*
- $z_2$ *sat* $\vartheta :: S_2$ *is a valid judgment,*
- $z_3 =$ **if** $b$ **then** $z_1$ **else** $z_2$ **fi**, *and*
- $z$ **behaves as** $z_3$.

These definitions justify that the satisfaction is indeed structural; not only the behavior of the specified program is constrained, but to some extends, also its structure. For instance, a program that satisfies $\vartheta :: S_1$; **announce**(*exp*) needs to be written as the sequential composition of a program satisfying $\vartheta :: S_1$ and some event announcement. In general, the concepts of structural and extended behavioral specifications have the following advantages.

1. It might be argued that by including specifications of the forms **if** $b$ **then** $S_1$ **else** $S_2$ **fi**, $\{S_1 \| S_2\}$, or $\{S_1; S_2\}$, the concepts of structural specifications and extended behavioral specifications are not abstract enough. We, however, argue that this is not the case; structural specifications and (extended) behavioral specifications include traditional specifications that are of the form $(P, R, G, E)$. Any program specified by means of the concepts of structural and behavioral specifications can, therefore, be specified at the same level of abstraction as when using the 4-tuple $(P, R, G, E)$.

2. Structural and behavioral specifications are more intelligible than traditional specifications; they are easier to read and to understand. The reader can easily capture the (possible) structural constitution of such specifications. The parts are still existing in the whole.

3. Announcement of events is supported in structural specifications. Traditional specifications do not lend themselves to easy formulation of properties such as announcement of events.

In general, structural specifications allow designers to express their requirements at the level of abstraction they find adequate. Despite these advantages, structural specifications have the problem that they do not allow verifying properties of systems. For instance, what does it means to say that the program a program satisfying $\vartheta : :(P, R, G, E); \textbf{announce}(exp)$ terminates in a state satisfying $Q$. Therefore, a means must be found to convert such specifications into behavioral and extended specifications.

## 6.6 Summary

Any framework for the formal construction of software systems must:

- indicate how to formally describe these systems,
- define a set of specificand for assigning meaning to specifications, and
- show how to relate a specification to its semantics.

We have shown in this chapter how this can be done for event-based applications. We distinguished two kinds of specification, namely behavioral and structural specifications. We claimed that structural specifications are more intelligible while they do not allow verifying properties of systems. This lead us to the argument that a way must be proposed for converting specifications from structural to behavioral ones.

Yet, there is a number of questions that are still open. How do we know that these specifications are well-suited for the announcement of events? how to specify that a program announces an event after the fulfillment of the condition $Q$? The remainder of the thesis answers these questions. So, please hold on!

# Chapter 7

# Construction of Systems

## 7.1 Overview

The development process of an event-based application (depicted in Figure 1.2) is composed of four main steps that we listed in the previous chapters. Given that we now know what specifications are involved in this development process, we can give more information on it.

The first step of the process is concerned with the architectural design where the different components of the applications are identified. This step is clearly out of the scope of this thesis.

The second step is concerned with constructing structural specifications of components which are specifications that do not depend on any binding. In fact, structural specifications, allow announcement of events to be specified in such conditions where the binding is undefined. This step also includes the verification of local properties related to specifications of components. Given a structural specification of a component, say $\vartheta :: \{S_1 \| S_2\}$, there is a-priori no indication that the programs which satisfy $\vartheta :: S_1$ and $\vartheta :: S_2$ do not interfere with each other. Such requirements can, and need to, be discharged in isolation. For this, the empty binding can be assumed and the behavioral specifications derived.

Once the local properties concerning the specifications of components are verified, these specifications can be composed to construct the complete behavioral specification of the application. This process includes adding new subscriptions to the binding and discharging the global properties of the application. If a subscription relating the program $z$ to the event $e$ is added to the binding, the behavior of any program that announces $e$ must be reviewed for interference freedom. Finally, some global properties may be checked.

Figure 7.1 depicts this process and shows that iterations may be done between the various steps. For instance, if the global property $\psi$ does not hold, it may be because the binding is not adequately constructed, in which case we go back and adjust it.

The aim of this chapter is twofold. First, to present a set of rules for the decomposition of programs. These rules are the basis for the top-down development of components.

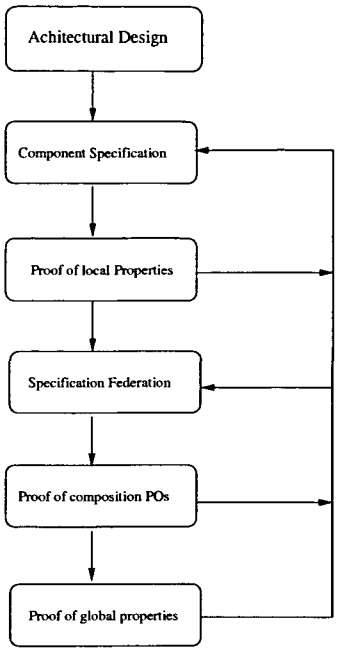


Figure 7.1: Specifications Development Process in Event-Based Systems

Next, we show how to integrate new components into an event-based application. That is, we show how the (behavioral or extended behavioral) specification of an application is composed starting with the specifications of the components. In addition, Section 7.5 discusses a symbolic example whose purpose is to clarify the complete development process of an event-based application. Section 7.6 summarizes the chapter.

## 7.2 Construction of Components

This section presents the rules for the top-down construction of programs (components). The rules are extensions and adoptions of those investigated in [77, 120, 136] and show how a specification can successively be decomposed, hence are called decomposition rules. They are of one of the following forms:

$$\frac{\begin{array}{c} \textit{premise } 1 \\ \textit{premise } 2 \\ \ldots \\ \textit{premise } n \end{array}}{\textit{conclusion.}}$$

The rule means that if each of its premises holds in the structure $\pi$, then the conclusion follows in the same structure.

The symbol $\diamond$ is used in some of the following rules as a generic operator that can be replaced with the parallel composition operator $\|$ or with the sequential composition operator ;. This replacement must, however, be the same in the same rule. That is, $\diamond$ can not be replaced with $\|$ at one place of a rule and with ; at another place of the same rule.

### 7.2.1 Consequence Rule

The consequence rule allows strengthening the assumptions while weakening the commitments in a specification. It is the basis for the refinement of specifications.

$$\frac{\begin{array}{l} E_1 \Rightarrow E_2, \\ G_1 \Rightarrow G_2, \\ R_2 \Rightarrow R_1, \\ P_2 \Rightarrow P_1, \\ z \ \underline{sat}\ \vartheta ::(P_1,\ R_1,\ G_1,\ E_1). \end{array}}{z \ \underline{sat}\ \vartheta ::(P_2,\ R_2,\ G_2,\ E_2).}$$

If $z$ is executed in an environment satisfying the assumptions $P_2$ and $R_2$, since $P_1$ and $R_1$ follow from $P_2$, and $R_2$, $z$ is in fact executed in an environment satisfying $P_1$ and $R_1$ and, therefore, guarantees $G_1$ and $E_1$ which however imply $G_2$ and $E_2$ respectively.

### 7.2.2 Composed Consequence Rule

$$\frac{\begin{array}{l} z_1 \ \underline{sat}\ \vartheta ::S \Rightarrow z_1 \ \underline{sat}\ \vartheta ::S' \\ z \ \underline{sat}\ \vartheta ::S_1 \diamond S \diamond S_2 \end{array}}{z \ \underline{sat}\ \vartheta ::S_1 \diamond S' \diamond S_2.}$$

Let us investigate the case where $\diamond$ is replaced with the sequential composition operator. If $z$ can be written as the sequence of three programs satisfying $\vartheta ::S_1$, $\vartheta ::(P_1,\ R_1,\ G_1,\ E_1)$, and $(P_1,\ R_1,\ G_1,\ E_1)::S_2$, then, since the second program also satisfies $\vartheta ::P_2,\ R_2,\ G_2,\ E_2)$ the consequence of the rule follows.

### 7.2.3 Parallel Rules

The parallel rule justifies the decomposition of $\{z_1 \| z_2\}$ into $z_1$ and $z_2$. If the programs $z_1$ and $z_2$ can coexist, then their parallel composition is a program that ends in a state that satisfies the post-condition of each of these programs. Coexistence means that the rely-condition of each follows from the guar-condition of the other.

### 7.2.4 Basic Parallel Rule

$$\frac{\begin{array}{l} G_2 \Rightarrow R_1 \\ G_1 \Rightarrow R_2 \\ z_1 \ \underline{sat}\ \vartheta::(P,\ R_1,\ G_1,\ E_1) \\ z_2 \ \underline{sat}\ \vartheta::(P,\ R_2,\ G_2,\ E_2) \end{array}}{\{z_1 \| z_2\} \ \underline{sat}\ \vartheta::(P,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ E_1 \wedge E_2).} \tag{7.1}$$

### 7.2.5 Composed Parallel Rule

$$\frac{\begin{array}{l} G_2 \Rightarrow R_1 \\ G_1 \Rightarrow R_2 \\ events(z) = \{\} \\ z \ \underline{sat}\ \vartheta::(P,\ R_1,\ G_1,\ E_1) \| (P_2,\ R_2,\ G_2,\ E_2) \end{array}}{z \ \underline{sat}\ \vartheta::(P,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ E_1 \wedge E_2).} \tag{7.2}$$

If $z$ can be written as the parallel composition of two programs satisfying $\vartheta::(P,\ R_1,\ G_1,\ E_1)$ and $\vartheta::(P_2,\ R_2,\ G_2,\ E_2)$ respectively, then by application of the basic parallel rule one deduces the conclusion of the composed parallel rule.

### 7.2.6 Sequential Rule

The rule permits the sequential composition of programs.

### 7.2.7 Basic Sequential Rule

The sequential composition is permitted if whenever started in a state satisfying its pre-condition, the first program guarantees to terminate in a state satisfying the pre-condition of the second program. In these conditions, the result of the composition terminates in a state satisfying $E_1 \mid E_2$ which is the composition of the assertions $E_1$ and $E_2$.

$$\frac{\begin{array}{l} z_1 \ \underline{sat} \ \vartheta::(P_1,\ R_1,\ G_1,\ E_1 \wedge P_2) \\ z_2 \ \underline{sat} \ \vartheta::(P_2,\ R_2,\ G_2,\ E_2) \end{array}}{z_1; z_2 \ \underline{sat} \ \vartheta::(P_1,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ E_1 \mid E_2).} \tag{7.3}$$

Obviously, any program transition of the composition $z_1; z_2$ is either a transition of $z_1$ or $z_2$, and therefore satisfies $G_1$ or $G_2$. The environment must ensures the rely-condition of each of the programs $z_1$ and $z_2$, i.e. $R_1 \wedge R_2$.

### 7.2.8 Composed Sequential Rule

$$\frac{\begin{array}{l} events(z) = \{\} \\ z \ \underline{sat} \ \vartheta::(P_1,\ R_1,\ G_1,\ E_1 \wedge P_2);(P_2,\ R_2,\ G_2,\ E_2) \end{array}}{z \ \underline{sat} \ \vartheta::(P_1,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ E_1 \mid E_2).} \tag{7.4}$$

If $z$ can be written as the sequential composition of two programs satisfying $\vartheta::(P,\ R_1,\ G_1,\ E_1)$ and $\vartheta::(P_2,\ R_2,\ G_2,\ E_2)$ respectively, then by application of the sequential rule one deduces the conclusion of the rule.

### 7.2.9 Conditional Rule

The conditional rule is probably one of the simplest rule if we take into consideration that the environment is not allowed to interfere on variables used in the boolean test.

### 7.2.10 Basic Conditional Rule

$$\frac{\begin{array}{l} z_1 \ \underline{sat}\ \vartheta ::(P \wedge b,\ R,\ G,\ E) \\ z_2 \ \underline{sat}\ \vartheta ::(P \wedge \neg b,\ R,\ G,\ E) \end{array}}{\textbf{if } b \textbf{ then } z_1 \textbf{ else } z_2 \textbf{ fi } \underline{sat}\ \vartheta ::(P,\ R,\ G,\ E).} \tag{7.5}$$

### 7.2.11 Composed Conditional Rule

$$\frac{z \ \underline{sat}\ \vartheta ::\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}; S_3}{z \ \underline{sat}\ \vartheta ::\textbf{if } b \textbf{ then } S_1; S_3 \textbf{ else } S_2; S_3 \textbf{ fi}.} \tag{7.6}$$

If the program $z$ does either $S_1$ or $S_2$ and subsequently performs $S_3$, then the program $z$ does either $S_1$ followed by $S_3$ or $S_2$ followed by $S_3$.

### 7.2.12 Global Rule

The global rule allows introduction of new shared variables in the specifications of programs. Since the new variable is assumed not to occur in $z$, $z$ does not change its value.

$$\frac{z \ \underline{sat}\ \vartheta \setminus \{v\} ::(P,\ R,\ G,\ E)}{z \ \underline{sat}\ \vartheta \cup \{v\} ::(P,\ R,\ G \wedge I_{\{v\}},\ E).} \tag{7.7}$$

### 7.2.13 Pre Rule

The pre- rule is straightforward. It allows adding more information in the post-condition of a program. If a program must end in a state satisfying $E$ when started in a state satisfying $P$, then, it ends in a state that satisfied $P$ and which was transformed such that it now satisfies $E$.

$$\frac{z \ \underline{sat}\ \vartheta ::(P,\ R,\ G,\ E)}{z \ \underline{sat}\ \vartheta ::(P,\ R,\ G,\ \overleftarrow{P} \wedge E).} \tag{7.8}$$

### 7.2.14 Post Rule

The post-rule is as straightforward as the pre-rule. It allows adding more information in the post-condition and claims that if the final state of a program satisfies $E$, then, it was reached by a finite number of guar- and rely-transitions.

$$\frac{z \ \underline{sat} \ \vartheta ::(P,\ R,\ G,\ E)}{z \ \underline{sat} \ \vartheta ::(P,\ R,\ G,\ E \wedge (R \vee G)^{+}).} \tag{7.9}$$

### 7.2.15 Skip Rule

The **skip** statement tolerates any interference that conserves the validity of $P$. Its unique program transition does nothing but terminates. Thanks to the stability of the pre-condition $P$, the program terminates in a state satisfying $P$.

$$\frac{P \textbf{ stable when } R}{\textbf{skip } \underline{sat} \ \vartheta ::(P,\ R,\ \mathsf{false},\ P).} \tag{7.10}$$

### 7.2.16 Assignment rule

A single program transition is performed, namely the assignment of the value of $r$ to the variable $v$ while all other variables are kept unchanged. The assignment is done in a state that satisfied $P$. If from these conditions we can derive that the guar- and the post-conditions will be satisfied, then the conclusion of the rule follows.

$$\frac{\begin{array}{l} P, E \textbf{ stable when } R \\ \overleftarrow{P} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \{v\}} \quad \Rightarrow \quad (G \vee I_{\vartheta}) \wedge E \end{array}}{v := r \ \underline{sat} \ \vartheta ::(P, R, G, E).} \tag{7.11}$$

The fact that the environment may interfere before and after the assignment is rendered obsolete by the stability requirement.

### 7.2.17 Iteration Rule

In this rule, the pre-condition $P$ is an invariant of the loop; each iteration must ensure not only its post-condition, but also the pre-condition of the next iteration.

$$\frac{\begin{array}{l} b \textbf{ stable when } R \\ z \ \underline{sat}\ \vartheta :: (P \wedge b,\ R,\ G,\ E \wedge P) \end{array}}{\textbf{while } b \textbf{ do } z \textbf{ od } \ \underline{sat}\ \vartheta :: (P,\ R,\ G,\ (E^{+} \vee R^{*}) \wedge \neg b)\,.} \tag{7.12}$$

## 7.3 Components Integration

A clear way of integrating components is a crucial requirement for any methodology that supports the development of event-based applications. This section discusses this issue. Integrating a component in an event-based system is trivial if this component is not subscribed to some event. Any program conserves its behavior. The formula $\mathcal{B} \cup \{z_1 \mapsto \{\}\}$ represents the binding obtained from $\mathcal{B}$ by additionally mapping the program $z_1$ to the empty set of events.

Integration Rule

$$\frac{\begin{array}{l} z_1 \notin \mathsf{dom}\ \mathcal{B} \\ \mathcal{B}_1 = \mathcal{B} \cup \{z_1 \mapsto \{\}\} \\ z \ \underline{sat}\ (\vartheta,\ \mathcal{B}) :: (P,\ R,\ G,\ E) \end{array}}{z \ \underline{sat}\ (\vartheta,\ \mathcal{B}_1) :: (P,\ R,\ G,\ E)} \tag{7.13}$$

Things are instead more interesting when subscribing a component of the event-based system to an event. There is no additional rule that tackles this issue. The process of transforming a structural specification to a behavioral specification is used for this purpose. Informally, the following steps must be followed:

- Subscription of the program $z_1$ to the event $e$ by adding the corresponding entry to the binding,
- Identification of any program $z_2$ that possibly announces the event $e$,
- Transformation of the structural specification of $z_2$ into a behavioral specification while discharging the related proof obligations.

When transforming the structural specification of $z_2$ to a behavioral specification, it is important to reuse proofs that were already discharged. For instance, it is very likely that the specification of $\|subscribers(e)$ will be computed. The general requirement for this is that any two distinct subscribers to $e$ be such that the rely-condition of one follows from the guar-condition of the other. Since this requirement was already fulfilled in the binding before the subscription, it is sufficient to prove that the rely-condition of the newly subscribed program follows from the guar- condition of any other subscribed program while the guar-condition of the newly subscribed program implies the rely-condition of any other subscribed program.

## 7.4 System Behavior Analysis

The integration of a component in the event-based system requires verifying interference freedom and validity of some global properties. This verification is based on behavioral specifications which are specifications of the composed application obtained by defining a binding that reflects the architecture of the application.

This section shows how to compose the specification of the application on the basis of the specifications of the components. Further, the section presents some rules for manipulating behavioral specifications. In fact, most of the rules presented for structural specifications are easily extended to behavioral specifications.

In particular, the consequence rule, the parallel rule, conditional rule, the global rule, the pre rule, the post rule, the skip rule, and the assignment rule are simply obtained by replacing the set of variables $\vartheta$ with the event-based system $(\vartheta, \mathcal{B})$. For instance, the behavioral consequence rule is given below. Other rules are omitted as they are trivially derived form the structural versions.

### 7.4.1 Consequence Rules

The consequence rule allows strengthening the assumptions while weakening the commitments in a specification. It is the basis for the refinement of specifications.

### 7.4.2 Basic Consequence Rule

$$\frac{\begin{array}{l} E_1 \Rightarrow E_2, \\ G_1 \Rightarrow G_2, \\ R_2 \Rightarrow R_1, \\ P_2 \Rightarrow P_1, \\ z \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P_1,\ R_1,\ G_1,\ E_1) \end{array}}{z \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P_2,\ R_2,\ G_2,\ E_2).}$$

### 7.4.3 Sequential Rule

The sequential rule is one of the rules (besides the iteration rule) whose behavioral version is not derived trivially from the structural version. The rule permits the sequential composition of programs. We consider two programs $z_1$ and $z_2$ such that $z_1$ announces no events, i.e. $events(z) = \{\}$. The sequential composition is possible if the pre-condition of the second program follows from the post-condition of the first.

### 7.4.4 Basic Sequential Rule

$$\frac{\begin{array}{l} z_1 \ \underline{sat}\ \vartheta ::(P_1,\ R_1,\ G_1,\ E_1 \wedge P_2) \\ z_2 \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P_2,\ R_2,\ G_2,\ E_2) \end{array}}{z_1; z_2 \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P_1,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ E_1 \mid E_2).} \tag{7.14}$$

The requirement that the first program announces no event is included in the structural specification $z_1 \ \underline{sat}\ \vartheta ::(P_1,\ R_1,\ G_1,\ E_1 \wedge P_2)$ and is important for the rule to hold.

### 7.4.5 Composed Sequential Rule

$$\frac{z \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P_1,\ R_1,\ G_1,\ E_1 \wedge P_2); (P_2,\ R_2,\ G_2,\ E_2)}{z \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P_1,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ E_1 \mid E_2).} \tag{7.15}$$

The composed sequential rule requires no modification as the meaning of $z \ \underline{sat}\ (\vartheta, \mathcal{B})::S_1; S_2$ requires that the first program announces no event.

### 7.4.6 Composition of Specifications: Announce Rule

The announce rule allows generating behavioral specifications based on a binding and some structural specifications. It results from the parallel rule and from the semantics of the announce construct. If $e$ is an event and $\{z_1, \cdots, z_n\}$ is the set of subscribers to $e$, then the following rule holds.

$$\frac{\begin{array}{l} subscribers(e) = \{z_1, \cdots z_n\} \\ \{z_1 \| \cdots \| z_n \| z\} \ \underline{sat} \ (\vartheta, \mathcal{B}) ::(P,\ R,\ G,\ E) \end{array}}{\textbf{announce}(e); z \ \underline{sat} \ (\vartheta, \mathcal{B}) ::(P,\ R,\ G,\ E).} \tag{7.16}$$

This rule is, however, applicable only to cases where $e$ is a constant event value and the set of subscribers well defined. In practice, however, an expression *exp* may be used whose value depends upon the state variables. In this case, $subscribers(exp)$ can not unequivocally be defined.

Let us take for instance the case where *exp* is the variable *count* which is a state variable. Further, a program $z_1$ may be subscribed to events satisfying $count > 10$ while another program $z_2$ may be subscribed to events that satisfy $count > 0$. The set of subscribers to the event *count* is $\{z_2\}$ when $count \leq 10 \wedge count > 0$ while it takes the value $\{z_1, z_2\}$ if $count > 10$ holds. We need to split the range of *count* such that in each interval, $subscriber(e)$ is defined unequivocally.

$$\frac{\begin{array}{l} events(z) = X_1 \uplus X_2 \\ \{z_{1i} \| \cdots \| z_{ni} \| z_{n+1}\} \ \underline{sat} \ (\vartheta, \mathcal{B}) ::S_i \\ \forall e \in X_i \cdot subscribers(e) = \{z_{1i}, \cdots z_{ni}\} \\ z \ \underline{sat} \ \vartheta ::(P,\ R,\ G,\ E); \textbf{announce}(exp) \end{array}}{z; z_{n+1} \ \underline{sat} \ (\vartheta, \mathcal{B}) ::(P,\ R,\ G,\ E); \textbf{if } exp \in X_1 \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}.} \tag{7.17}$$

The range of the expression *exp* in the specification $\vartheta ::(P,\ R,\ G,\ E); \textbf{announce}(exp)$ is the set of values that it may take. This set is defined as

$$events(z) = \{e : Event \cdot \exists s_1, s_2 : State \cdot (s_1, s_2) \models E \wedge exp = e\}$$

which is the set of events $e$ such that there exists two states that validate the assertion $E \wedge e = exp$.

In this rule, it is assumed that this set can be split into two disjoint subsets $X_1$ and $X_2$ such that all events in $X_1$ have the same set of subscribers and all events in $X_2$ have the same set of subscribers. The rule can easily be extended to cases where more $X_i$ are required. The result of the rule is a behavioral specification that reflects the architecture of the application.

### 7.4.7 Conditional Rule

The conditional rule in the context of behavioral specifications is slightly different from that of structural specifications. The specification of the program **if** $b$ **then** $z_1$ **else** $z_2$ **fi**; $z$ is given instead of simply giving that of **if** $b$ **then** $z_1$ **else** $z_2$ **fi**. In this way, announcement of events is supported within if constructs.

### 7.4.8 Basic Conditional Rule

$$\frac{\begin{array}{l} z_1; z \ \underline{sat}\ (\vartheta, \mathcal{B}) : : (P \wedge b,\ R,\ G,\ E) \\ z_2; z \ \underline{sat}\ (\vartheta, \mathcal{B}) : : (P \wedge \neg b,\ R,\ G,\ E) \end{array}}{\textbf{if } b \textbf{ then } z_1 \textbf{ else } z_2 \textbf{ fi}; z \ \underline{sat}\ (\vartheta, \mathcal{B}) : : (P,\ R,\ G,\ E).} \tag{7.18}$$

### 7.4.9 Composed Conditional Rule

$$\frac{z \ \underline{sat}\ (\vartheta, \mathcal{B}) : : \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}; S_3}{z \ \underline{sat}\ (\vartheta, \mathcal{B}) : : \textbf{if } b \textbf{ then } S_1; S_3 \textbf{ else } S_2; S_3 \textbf{ fi}.} \tag{7.19}$$

The composed conditional rules is similar to the case of structural specifications.

### 7.4.10 Iteration rule

The iteration rule is the second rule whose behavioral formulation differs from the sequential one. This is understandable as the semantics of the iteration construct is given in terms of the sequential composition.

In fact, we have not yet found a general enough and elegant formulation for the support of event announcement in the `while` construct.

To touch at the difficulties in the formulation of this rule, let us consider the programs $z \ \underline{sat} \ \vartheta ::(P, R, G, E).$ and $z_w \stackrel{def}{=}$ **while** $b$ **do** $z$; **announce**$(e)$ **od** where $e$ is a constant value (without occurence of variables). If the loop is executed three times, $z_w$ can be expanded into:

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

Let us also assume a binding where the only program subscribed to the event $e$ is

$$z_1 \ \underline{sat} \ \vartheta ::(P_1, R_1, G_1, E_1).$$

In case there is no interference, the result of the above iteration would be:

$$E \mid E \mid E \ \wedge \ E \mid E_1 \ \wedge \ E \mid E \mid E_1 \ \wedge \ E \mid E \mid E \mid E_1$$

This result is unfortunately not easily generalized to cases, for instance where the event $e$ is an expression that depends upon some global variables or where the announcement construct appears within the body of some if constructs.

**Note:** The set of behavioral rules is sufficient for developing event-based applications when the binding is static; no structural specifications are required. An adequate methodology for the development of event-based applications must, however, foresee the integration of new components into the application. In this respect, this set of rules is insufficient. A naive way of supporting the addition of new components into the application is to restart the verification of all components from scratch after each modification of the binding. Clearly, such a method does not scale to applications with a non trivial number of components; compositionality is required, which is provided by the announce rule.

### 7.4.11 Practical Considerations

The following extension of the skip rule was shown to be useful in practice.

$$\frac{z \ \underline{sat} \ (\vartheta, \mathcal{B}) ::(P, \ R, \ G, \ E)}{\begin{array}{l} z; \textbf{skip} \ \underline{sat} \ (\vartheta, \mathcal{B}) ::(P, \ R, \ G, \ E) \\ \textbf{skip}; z \ \underline{sat} \ (\vartheta, \mathcal{B}) ::(P, \ R, \ G, \ E) \\ z \| \textbf{skip} \ \underline{sat} \ (\vartheta, \mathcal{B}) ::(P, \ R, \ G, \ E) \end{array}} \tag{7.20}$$

The behavior of the program $z$ is not affected by any parallel execution with **skip**. This rule also holds for structural specifications.

### 7.4.12 Non Refinement of Matching Functions

This section forbids the refinement of matching functions. A matching function is a function that takes a subscription and an event and decides whether the event matches the subscription. When a subscription is modeled as a unary assertion, the matching function is simply the membership operator (i.e. a boolean function).

*As a function, one may be tempted to refine a matching function, we urge to resist this temptation.*

Let us illustrate what happens to a refined matching function. For this, we consider a matching function $m_1$ that is required to satisfy the specification $(P, Q \vee Q_1)$. By the traditional consequence rule, a program $m_2$ that satisfies $(P, Q)$ also satisfies $(P, Q \vee Q_1)$ and can be used instead of $m_1$. This implies that we can have a tuple $(e, s)$ (where $e$ is an event and $s$ a subscription) such that $m_1(e, s)$ holds while $m_2(e, s)$ does not hold. In other terms, we could have a situation where at the abstract level a program $z_1$ is subscribed to the event $e$ announced by some $z$ while at the concrete level the implementation of $z_1$ is not subscribed to $e$. The implementation of $z$ would subsequently not be a refinement of $z$.

## 7.5 A symbolic Example

This section presents a symbolic example whose intend is to illustrate the application of the proposed development methodology in a succinct and condensed manner. By symbolic example, we mean an example in which the assertions are not further defined.

### 7.5.1 Specification of Components

We assume two components $z_1$ and $z_2$ with the following structural specifications where $\mathcal{S}_0 = (\vartheta, \mathcal{B}_0)$.

$z_1$ *sat* $\vartheta ::\{(P_{11},\ R_{11},\ G_{11},\ E_{11}) \| (P_{12},\ R_{12},\ G_{12},\ E_{12})\}; \mathbf{announce}(e); (P_{13},\ R_{13},\ G_{13},\ E_{13})$.

$z_2$ *sat* $\vartheta ::(P_{21},\ R_{21},\ G_{21},\ E_{21})$.

### Verification of Local Properties

We want to show that the assertion $Q$ holds in any final state of $z_1$. This property is local as it is a property of $z_1$ and does not require a binding to be constructed. It is independent of the application in which the component $z_1$ will be integrated. To discharge this property we must transform the specification $z_1$ into a behavioral specification.

### Derivation of the Behavioral Specification

By application of the announce rule, followed by the skip rule, we derive that:

$z_1$ *sat* $(\vartheta, \mathcal{B}_0) :: \{(P_{11},\ R_{11},\ G_{11},\ E_{11}) \| (P_{12},\ R_{12},\ G_{12},\ E_{12})\}; (P_{13},\ R_{13},\ G_{13},\ E_{13})$.

The behavioral specification of $z_2$ is obtained by simply replacing $\vartheta$ with the event-based system $(\vartheta, \mathcal{B}_0)$ because $z_2$ announces no event.

The following proof obligation must be discharged for the program $z_1$ to have a predictable result. The first is required by the parallel rule and the second is required by the sequential rule.

**Proof Obligation 1** $(G_{11} \Rightarrow R_{12})$ *and* $(G_{12} \Rightarrow R_{11})$.

**Proof Obligation 2** $E_{11} \wedge E_{13} \Rightarrow P_{13}$

We assume in the remainder that these proof obligations are discharged. We can now formulate the proof obligation of interest.

**Proof Obligation 3** $(E_{11} \wedge E_{12}) \mid E_{13} \Rightarrow Q$.

The left part of the implication is the post-condition of the behavioral specification of $z$ obtained by applying the parallel rule first and the sequential rule next.

Note that due to the clarity of the specification, there is no need to explicitly write each step of the derivation of this proof obligation.

### Integration of Components

We now compose the application, i.e. the binding is constructed. We subscribe the program $z_2$ to the event $e$. Our intention is to obtain an application such that $E_{21}$ holds in the final state of $z$. The binding now looks as follows:

$$\mathcal{B}_1 = \{\ \langle \mathbf{z}_1 \rangle \ \mapsto\ \{\},\quad \langle \mathbf{z}_2 \rangle \ \mapsto\ \{e\},\quad \langle \mathbf{skip} \rangle \ \mapsto\ \{x : Event\}\}$$

and is such that $subscribers(e) = \{z_2, \mathbf{skip}\}$.

### Derivation of the Behavioral Specification

As above, we must apply the announce rule followed by the skip rule to derive the following behavioral specification.

$z_1$ *sat* $(\vartheta, \mathcal{B}_0)$::$\{(P_{11},\ R_{11},\ G_{11},\ E_{11}) \| (P_{12},\ R_{12},\ G_{12},\ E_{12})\}; \{(P_{21},\ R_{21},\ G_{21},\ E_{21}) \| (P_{13},\ R_{13},\ G_{13},\ E_{13}\})$.

In addition to the proof obligations discharged above, the following proof obligation must be discharged to ensure interference freedom. The first proof obligation is required by the parallel rule while the second is required by the sequential rule.

**Proof Obligation 4** $(G_{21} \Rightarrow R_{13}) \wedge (G_{13} \Rightarrow R_{21})$.

**Proof Obligation 5** $(E_{11} \wedge E_{12}) \Rightarrow P_{21}$.

The proof obligation for the property of interest is therefore:

**Proof Obligation 6** $(E_{11} \wedge E_{12}) \mid (E_{21} \wedge E_{13}) \Rightarrow E_{21}$.

If this property holds, the components $z_1$ and $z_2$ can now be developed independently by stepwise refinement of their structural specifications. The program $z_1$ for instance, will be developed by stepwise development of the components $z_{11}$, $z_{12}$, and $z_{13}$ that satisfy $\vartheta$::$(P_{11},\ R_{11},\ G_{11},\ E_{11})$, $\vartheta$::$(P_{12},\ R_{12},\ G_{12},\ E_{12})$, and $\vartheta$::$(P_{13},\ R_{13},\ G_{13},\ E_{13})$ respectively. Obviously, COTS components can also be used instead.

A general problem that one must be aware of when using such COTS components is that they are already implemented and nothing (except refinements) can be modified on their specifications to make them fulfill the requirements that one would like to achieve. In this example, if the composition of the application reveals that the property $E_{21}$ is not satisfied, we can modify the specification of $z_2$ to make it fit our requirements. The implementation of $z_2$ is only ordered when we are sure that its specification describes the expected behavior. If, however, $z_2$ is an "off-the-shelf" component and its specification does not allow the fulfillment of the property of interest, we must change the component and investigate another one.

## 7.6 SUMMARY

This chapter presented a set of rules for the stepwise construction of event-based applications. In the introductory section, we gave more details on the development process, the steps involved in this process, and we elicited the use of the various kinds of specifications that are involved in this process. Next, we provided some rules for the decomposition of specifications. These rules are intended for the top-down development of programs on the one hand, and for the transformation of structural specifications into behavioral specifications on the other hand. The details and the necessity of this process were also discussed. Finally, this chapter presented a symbolic example that illustrates in a succinct manner the different steps in the development process of an event-based application.

# Chapter 8

# Synchronization and Mutual Exclusion

## 8.1 Motivation

We have shown the similarity between event-based systems and concurrent systems. On the other hand, synchronization and mutual exclusion are two techniques without which the development of a large class of concurrent systems would be difficult —if possible at all. Mutual exclusion is the assurance that only one process is given access to a shared resource at any one time. We show in this chapter how synchronizing event-based programs can be constructed. Mutual exclusion is not sufficient to provide fair access to resources; deadlock may arise. To prevent this, we must be aware of methods and techniques which allow us to control the allocation of resources to processes. Stølen [120] and Xu [135] have shown how to ensure deadlock freedom in the stepwise development of state based concurrent systems. We adapt these techniques to allow the stepwise development of synchornizing deadlock-free event-based programs. In the style of Stølen [120], we extend specifications to include a new component called wait-condition which is a unary assertion. A program is allowed either to terminate or to block. However, the program may only block in a state satisfying the wait-condition. Further, it is required that no program blocks within the body of an await-statement.

The remainder of the chapter is organized as follows. The next section elaborates on the construction of synchronizing event-based applications. In particular, the impact of the `await` construct is discussed. We extend behavioral and structural specifications with wait-conditions in Section 8.3. Section 8.4 adapts the rules for the construction of event-based components to leverage the concepts of synchronization and mutual exclusion while Section 8.5 introduces auxiliary variables for the specification and verification of complex systems. Section 8.6 concludes the chapter.

## 8.2 MUTUAL EXCLUSION

The previous chapter ignored the await construct. This section shows how to construct programs that support mutual exclusion which is the assurance that no two programs access a shared resource simultaneously. The requirement in such concurrent systems is to let processes access the resources without causing data conflicts. The LECAP language provides an `await` construct that serves this purpose. In a program such as

**await** $b$ **do** z **od**,

the body of the `await` construct will be executed atomically without any interference. The specification of this program is given by the await-rule.

### 8.2.1 AWAIT RULE

The await-rule specifies the behavior of a program in an `await` construct. Let us assume that the program is executed in an environment whose transitions satisfy the rely-condition $R$. The execution of this construct is described as follows. Its body will not be executed until the blocking condition $b$ holds. During this blockage, the environment is allowed to interfere in a way that $R$ is satisfied. Therefore, the body of the `await` construct must have a pre-condition that is not affected by the interference of the environment. In other terms, this body will start in a state that is reached from a state satisfying $P$ by a finite number of $R$ transitions. This justifies the requirement on the stability of $P$. Similarly, after the execution of the construct, the environment can still perform a transition that satisfies $R$. To terminate in a state satisfying $E$, it is, therefore, important that $E$ be stable when $R$.

There are many other factors that influence the result of an `await` construct:

1. each program transition changing the state must do so in a way that the guarantee condition holds,
2. the body of the await-construct may be a **skip**, resulting in $I_\vartheta$,
3. due to the atomicity of the execution, the `await` construct only performs one program transition that is visible to the environment,
4. the final state must satisfy the post-condition $E$.

The first two of these factors imply that each program transition of the `await` construct which changes the state must satisfy the guar-condition. There is, however only one such

transition. This means that the body of the await construct must also satisfy the guar-condition if the state is changed, i.e. $(I_\vartheta \vee G)$ must hold.

$$\frac{\begin{array}{l} P \textbf{ stable when } R \\ E \textbf{ stable when } R \\ z \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P \wedge b,\ \mathsf{false}, \mathsf{true},\ (G \vee I_\vartheta) \wedge E) \end{array}}{\textbf{await } b \textbf{ do } z \textbf{ od } \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P, R, G, E)} \tag{8.1}$$

Announcement of events within the body of an await construct is not forbidden. If the body $z$ satisfies its specification the fact that it announces an event is not relevant. There is however a crucial difference between events announced inside await-constructs and other events. Any program that is subscribed to an event announced inside an await construct will be completely executed before the end of the construct. This has a direct impact on the sequential rule.

### 8.2.2 Sequential-Await Rule

The sequential-await-rule illustrates the fact that an event that is announced in an await construct has no effect on programs that are sequentially composed with this await construct. Note the absence of the requirement that the first program announces no event.

$$\frac{\begin{array}{l} z_2 \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P_2,\ R,\ G,\ E_2) \\ \textbf{await } b \textbf{ do } z_1 \textbf{ od } \underline{sat}\ (\vartheta, \mathcal{B}) ::(P_1,\ R,\ G,\ E_1 \wedge P_2) \end{array}}{\textbf{await } b \textbf{ do } z_1 \textbf{ od}; z_2 \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P_1,\ R,\ G,\ E_1 \mid E_2)} \tag{8.2}$$

## 8.3 Specification of Deadlock Free Programs

The previous section showed how to specify programs that depend on synchronization and mutual exclusion. It, however, ignored the issue of deadlock that may arise when more than one program wait for a condition to hold. This section shows an extension of the specification technique presented in Chapter 6, that supports developing deadlock free applications. In the style of Stølen [121], we extend specifications to include a new component called wait-condition which is a unary assertion. A program is allowed either to terminate or to block in a state satisfying the wait-condition. Further, it is required that no program blocks within the body of an await-statement.

### 8.3.1 SPECIFICATION

As in Chapter 6, we distinguish structural and behavioral specifications. The concept of event-based system remains the same.

### 8.3.2 DEFINITION

**Definition 29** *A behavioral specification is a formula* $(\vartheta, \mathcal{B}) ::(P, R, W, G, E)$, *where:*

- $(\vartheta, \mathcal{B})$ *is a an event-based system,*
- *the pre-condition* $P$, *and the wait-condition* $W$ *are unary assertions,*
- *the rely-condition* $R$, *the guar-condition* $G$ *and the post-condition* $E$ *are binary assertions,*
- *any variable occurring free in* $P$, $R$, $W$, $G$, *or* $E$ *is an element of* $\vartheta$,
- $P$ *is stable when* $R$,
- $E$ *is stable when* $R$.

The set $ext[(\vartheta, \mathcal{B}), P, R]$ constrains the environment and is composed of computations that are such that the first state satisfies the unary assertion $P$ while any transition labelled with $v$ is either such that the binary assertion $R$ holds or the state is kept unchanged.

**Definition 30** *Given the event-based system* $(\vartheta, \mathcal{B})$, *a unary assertion* $P$, *and a binary assertion* $R$, *then* $ext[(\vartheta, \mathcal{B}), P, R]$ *denotes the set of computations* $\sigma$ *such that the following conditions hold:*

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

On the other hand, configurations that characterize an adequate behavior of the running program are finite configurations with final states satisfying either the post-condition or the wait-condition and with program transitions that satisfy the guar-condition if they change any of the state variables.

**Definition 31** *Assuming an event-based system $(\vartheta, \mathcal{B})$, a unary assertion $W$, and two binary assertions $G$, and $E$, then $int[(\vartheta, \mathcal{B}), W, G, E]$ denotes the set of computations $\sigma$ such that the following conditions hold:*

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

These definitions implicitly take into consideration the case of a program $z_e$ triggered by an event $e$ announced by $z$; the subscriber $z_e$ is part of the running program which becomes $z_e \| z_1$ where $z_1$ is the remainder of $z$. In the parallel composition $z_e \| z_1$, $z_e$ and $z_1$ are in the environment of each other and are, therefore, required to coexist.

### 8.3.3 Extended Behavioral specifications

We introduce extended behavioral specifications as in the Chapter 6.

If we assume that the formulas $(\vartheta, \mathcal{B}) :: S_1$ and $(\vartheta, \mathcal{B}) :: S_2$ are either behavioral specifications (including extended ones) the same event-based system $(\vartheta, \mathcal{B})$, then, the formulas $(\vartheta, \mathcal{B}) :: S_1; S_2$, $(\vartheta, \mathcal{B}) :: S_1 \| S_2$, and $(\vartheta, \mathcal{B}) ::$ **if** $b$ **then** $S_1$ **else** $S_2$ **fi** are extended behavioral specifications.

In addition to these, $(\vartheta, \mathcal{B})$**await** $b$ **do** $S$ **od** is also a extended behavioral specification whose satisfaction is defined as follows.

**Definition 32** *A program $z$ satisfies the extended behavioral specification* $(\vartheta, \mathcal{B}) ::$**await** $b$ **do** $S$ **od** *iff two programs $z_1$ and $z_2$ exist such that the following hold:*

- $z_1$ *sat* $(\vartheta, \mathcal{B}_0) :: S$ *is a valid judgment,*
- $z_2 =$ **await** $b$ **do** $z_1$ **od**,
- $events(z_1) = \{\}$, *and*
- $z$ **behaves as** $z_2$ **in** $(\vartheta, \mathcal{B})$.

The simplest example of a program that satisfies this specification is of course

**await** $b$ **do** $z_1$ **od**.

### 8.3.4 Structural Specifications

Structural specifications are now introduced as of one the forms $S_1 \| S_2$, **if** $b$ **then** $S_1$ **else** $S_2$ **fi**, $S_1; S_2$, or **await** $b$ **do** $S$ **od**. These kinds of specifications allow specifying not only the behaviors of the specified program, but also make obvious which components this program is composed of.

**Definition 33** *We assume a set of variables $\vartheta$. A structural specification is defined recursively.*

- *If exp is an expression that evaluates to an event, the formula $\vartheta$ : :**announce**(exp) is a structural specification iff any variable occurring free in exp is in $\vartheta$.*
- *$\vartheta$ : :$(P,\ R,\ W,\ G,\ E)$ is a structural specification iff:*
  - *$P$, and $W$ are unary assertions and $R$, $G$, and $E$ are binary assertions whose free variables are in $\vartheta$, and*
  - *$P$ and $E$ are stable when $R$.*
- *If $\vartheta$ : :$S_1$ and $\vartheta$ : :$S_2$ are structural specifications on the same set of variables $\vartheta$, then, $\vartheta$ : :$S_1; S_2$, $\vartheta$ : :$S_1 \| S_2$, and $\vartheta$ : :**if** $b$ **then** $S_1$ **else** $S_2$ **fi**, $\vartheta$ : :**await** $b$ **do** $S$ **od** are also structural specifications.*

### 8.3.5 Satisfaction

The satisfaction relation relates a specification to a specificand. In other terms, we assign meanings to specifications.

**Definition 34** *The program $z$ satisfies $S \stackrel{def}{=} (\vartheta, \mathcal{B})$ : :$(P, R, W, G, E)$ (denoted as $\models$ $z$ sat $S$) iff $cp[z, \mathcal{B}] \cap ext[(\vartheta, \mathcal{B}), P, R] \subseteq int[(\vartheta, \mathcal{B}), W, G, E]$ holds.*

To show that $z$ *sat* $(\vartheta, \mathcal{B})$ : :$(P, R, W, G, E)$ is valid, one proves that any computation of $z$ started in a state satisfying $P$ and executed in an environment whose interference satisfies $R$ has a final state satisfying $E$ if it terminates and $W$ if it blocks while any program transition changing the state variables satisfies $G$.

**Definition 35** *We say that $(\vartheta, \mathcal{B})$ : :$(P, R, W, G, E)$ is valid iff for any program $z \in \mathcal{M}_x$, $z$ sat $(\vartheta, \mathcal{B})$ : :$(P, R, W, G, E)$ is valid.*

**Definition 36** *A program $z$ satisfies the specification $\vartheta::(P,\ R,\ W,\ G,\ E)$ iff:*

- $z$ *sat* $(\{\mathbf{skip}\}, \vartheta, \mathcal{B}_0)::(P,\ R,\ W,\ G,\ E)$ *holds and*
- $events(z) = \{\}$.

**Definition 37** *A program $z$ satisfies the specification $\vartheta::$* **await** $b$ **do** $S$ **od** *iff two programs $z_1$ and $z_2$ exist such that:*

- $z_1$ *sat* $\vartheta::S$ *holds,*
- $z_2 =$ **await** $b$ **do** $z_1$ **od**, *and*
- $z$ **behaves as** $z_2$.

Other cases of structural specifications are given semantics as in the previous chapter.

## 8.4 Construction of Systems

The construction of deadlock free applications includes the same steps presented in the previous chapter. In particular decomposition of programs must be elicited for synchronizing event-based applications.

### 8.4.1 Construction of Programs

This section presents the rules for the top-down construction of programs (components).

### 8.4.2 Parallel rule

We investigate the rule proposed by Stølen [121] first. A similar rule is proposed by Xu [135].

$$\frac{\begin{array}{l}\neg(W_1 \wedge W_2) \wedge \neg(W_2 \wedge E_1) \wedge \neg(W_1 \wedge E_2) \\ z_1 \ \underline{sat}\ \vartheta::(P, R \vee G_1,\ W \vee W_1,\ G_1,\ E_1) \\ z_2 \ \underline{sat}\ \vartheta::(P, R \vee G_2,\ W \vee W_2,\ G_2,\ E_2)\end{array}}{z_1 \| z_2 \ \underline{sat}\ \vartheta::(P, R,\ W,\ G_1 \vee G_2,\ E_1 \wedge E_2)} \tag{8.3}$$

We concentrate on the wait-conditions. The rule says that if we establish that $\neg(W_1 \wedge W_2) \wedge \neg(W_2 \wedge E_1) \wedge \neg(W_1 \wedge E_2)$ is true, then the parallel composition of $z_1$ and $z_2$ is a program that may block when $W$ holds. The intuition behind this rule is better grasped if we consider its following simplification where $W$ is false.

$$\frac{\begin{array}{l}\neg(W_1 \wedge W_2) \wedge \neg(W_2 \wedge E_1) \wedge \neg(W_1 \wedge E_2) \\ z_1 \ \underline{sat}\ \vartheta::(P, R \vee G_1, W_1, G_1, E_1) \\ z_2 \ \underline{sat}\ \vartheta::(P, R \vee G_2, W_2, G_2, E_2)\end{array}}{z_1 \| z_2 \ \underline{sat}\ \vartheta::(P, R, \mathsf{false}, G_1 \vee G_2, E_1 \wedge E_2)} \tag{8.4}$$

The programs $z_1$ and $z_2$ may block in states satisfying $W_1$ and $W_2$ respectively. Their parallel composition blocks if either one of them blocks while the other is terminated or both block. It is thus understandable that the composition of both programs does not block if we prove that neither of them blocks when the other is terminated and both programs do not simultaneously block. We reformulate these rules in a more direct and intuitive way:

$$\frac{\begin{array}{l}z_1 \ \underline{sat}\ \vartheta::(P, R \vee G_1, W_1, G_1, E_1) \\ z_2 \ \underline{sat}\ \vartheta::(P, R \vee G_2, W_2, G_2, E_2)\end{array}}{z_1 \| z_2 \ \underline{sat}\ \vartheta::(P, R, W, G_1 \vee G_2, E_1 \wedge E_2)} \tag{8.5}$$

$$\text{where } W \stackrel{def}{=} (W_1 \wedge W_2) \vee (W_2 \wedge E_1) \vee (W_1 \wedge E_2)$$

Independently of any assumption on the wait-conditions of $z_1$ and $z_2$, the rule describes the behavior of $z_1 \| z_2$. Any state in which $z_1 \| z_2$ blocks is such that either both programs $z_1$ and $z_2$ are blocked or one of them is blocked and the other terminated. The rule is a generalization of that proposed in [121, 136]. Showing that the composition of $z_1$ and $z_2$ does not deadlock consists of proving that $z_1 \| z_2$ never blocks.

$$\frac{\begin{array}{l}G_1 \Rightarrow R_2 \\ G_2 \Rightarrow R_1 \\ z_1 \ \underline{sat}\ (\vartheta, \mathcal{B})::(P,\ R_1,\ W_1,\ G_1,\ E_1) \\ z_2 \ \underline{sat}\ (\vartheta, \mathcal{B})::(P,\ R_2,\ W_2,\ G_2,\ E_2)\end{array}}{\{z_1 \| z_2\} \ \underline{sat}\ (\vartheta, \mathcal{B})::(P,\ R_1 \wedge R_2,\ W,\ G_1 \vee G_2,\ E_1 \wedge E_2)} \tag{8.6}$$

$$\text{where } W \stackrel{def}{=} (W_1 \wedge W_2) \vee (W_2 \wedge E_1) \vee (W_1 \wedge E_2).$$

The rule is also applicable if $z_1$ and $z_2$ satisfy the two structural specifications $\vartheta::(P,\ R_1,\ W_1,\ G_1,\ E_1)$ and $\vartheta::(P,\ R_2,\ W_2,\ G_2,\ E_2)$.

### 8.4.3 ANNOUNCE RULE

$$\frac{\begin{array}{l} events(z) = X_1 \uplus X_2 \\ \forall e \in X_i \cdot subscribers(e) = \{z_{1i}, \cdots z_{ni}\} \\ \{z_{1i} \| \cdots \| z_{ni} \| z_{n+1}\} \ \underline{sat}\ (\vartheta, \mathcal{B}) ::S_i \\ z \ \underline{sat}\ \vartheta ::(P,\ R,\ W,\ G,\ E); \textbf{announce}(exp) \end{array}}{z; z_{n+1} \ \underline{sat}\ \ (\vartheta, \mathcal{B}) ::(P,\ R,\ W,\ G,\ E); \textbf{if}\ exp \in X_1\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}.} \tag{8.7}$$

The domain of the expression $exp$ in the specification $\vartheta ::(P,\ R,\ G,\ E); \textbf{announce}(exp)$ is the set of values that it may take. This set is defined as

$$events(z) = \{e : Event \cdot \exists s_1, s_2 : State \cdot (s_1, s_2) \models E \wedge exp = e\}$$

which is the set of events $e$ such that there exists two states that validate the assertion $E \wedge e = exp$.

### 8.4.4 AWAIT RULE

The await rule is intended for synchronization and mutual exclusion. The await statement blocks until $b$ holds. During this blockage, the environment may interfere, yielding to the requirement that $P$ is stable when $R$. In addition to satisfying the post-condition, the final state of $z$ must satisfy the guar-condition if the state is changed (expressed as $G \vee I_\vartheta$).

$$\frac{\begin{array}{l} P\ \textbf{stable when}\ R \\ E\ \textbf{stable when}\ R \\ z \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P \wedge b,\ \mathsf{false},\ \mathsf{true},\ (G \vee I_\vartheta) \wedge E) \end{array}}{\textbf{await}\ b\ \textbf{do}\ z\ \textbf{od}\ \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P, R,\ P \wedge \neg b,\ G,\ E)} \tag{8.8}$$

### 8.4.5 SEQUENTIAL RULE

The rule permits the sequential composition of programs. We consider two programs $z_1$ and $z_2$ such that $z_1$ announces no event. The resulting program blocks if either of the composing programs blocks.

$$\frac{\begin{array}{l} z_1 \ \underline{sat}\ \vartheta : : (P_1,\ R_1,\ W_1,\ G_1,\ E_1) \\ z_2 \ \underline{sat}\ (\vartheta, \mathcal{B}) : : (P_2,\ R_2,\ W_2,\ G_2,\ E_2) \end{array}}{z_1; z_2 \ \underline{sat}\ (\vartheta, \mathcal{B}) : : (P_1,\ R_1 \wedge R_2,\ W_1 \vee W_2,\ G_1 \vee G_2,\ E_1 \mid E_2).} \tag{8.9}$$

### 8.4.6 Sequential-Await Rule

Unlike the previous rule that relies on the first program not announcing an event, this rule claims that the sequential composition may be done if the first program that possibly announces some events is embedded in an await construct.

$$\frac{\begin{array}{l} z_2 \ \underline{sat}\ (\vartheta, \mathcal{B}) : : (P_2,\ R,\ W_1,\ G,\ E_2) \\ \textbf{await}\ b\ \textbf{do}\ z_1\ \textbf{od}\ \underline{sat}\ (\vartheta, \mathcal{B}) : : (P_1,\ R,\ W_2,\ G,\ E_1 \wedge P_2) \end{array}}{\textbf{await}\ b\ \textbf{do}\ z_1\ \textbf{od}; z_2 \ \underline{sat}\ (\vartheta, \mathcal{B}) : : (P_1,\ R,\ W_1 \vee W_2,\ G,\ E_1 \mid E_2)} \tag{8.10}$$

### 8.4.7 Consequence Rule

The consequence rule allows strengthening the assumptions while weakening the commitments. It is the basis for the refinement of specifications. If $z$ blocks in states satisfying $W_1$, then $z$ blocks in states satisfying $W_2$ if $W_2$ follows from $W_1$.

$$\frac{\begin{array}{l} E_1 \Rightarrow E_2, \\ G_1 \Rightarrow G_2, \\ R_2 \Rightarrow R_1 \\ W_1 \Rightarrow W_2, \\ P_2 \Rightarrow P_1 \\ z \ \underline{sat}\ (\vartheta, \mathcal{B}) : : (P_1,\ R_1,\ W_1,\ G_1,\ E_1) \end{array}}{z \ \underline{sat}\ (\vartheta, \mathcal{B}) : : (P_2,\ R_2,\ W_2,\ G_2,\ E_2)} \tag{8.11}$$

Other rules can be derived in a straightforward manner from the rules in Chapter 7.

## 8.5 Auxiliary Variables

Auxiliary variables have been used as a tool not only for the verification of systems [136, 118], but also for their specification [121]. We extend Stolen's auxiliary variables rules

[121] and apply it to the event-based paradigm.

We extend the definition of an event-based system to include a set of auxiliary variables $\alpha$. The event-based system $\mathcal{S}$ is now defined as $(\vartheta, \alpha, \mathcal{B})$ where $\alpha \cap \vartheta = \{\}$. An auxiliary variable is a variable that may be used for the specification of programs although it does not belong to $\vartheta$.

There are some further restrictions on auxiliary variables:

- Auxiliary variables are not allowed to occur in tests of **if**, **while** and **await** statements. This restriction ensures that such variables have no influence on the implemented algorithm and correspondingly on the result of this implementation.
- Auxiliary variables are not allowed to appear on the right hand side of assignments unless the variable on the left hand side is also an auxiliary variable.
- Auxiliary variables should not depend on each other. In this way, it is possible to remove some auxiliary variables from the program without need of removing all auxiliary variables.

### 8.5.1 Specification

The basic structural specification is now a formula of the form $(\vartheta, \alpha) : :(P, R, G, W, E)$ where $\vartheta$ is the set of state variables and $\alpha$ is the set of auxiliary variables such that $\vartheta \cap \alpha = \{\}$. As in the previous chapters, any variable occurring free in $P$, $W$, $R$, $G$, or $E$ must be in $\vartheta \cup \alpha$. The behavioral specification $(\vartheta, \alpha, \mathcal{B}) : :(P, R, G, W, E)$ is defined similarly. Other kinds of structural specifications are defined as in the previous chapters.

### 8.5.2 Satisfaction

Based on these assumptions, any assignment $a: = u$ where $a$ is an auxiliary variable and $u$ is an expression is such that any variable occurring in $u$ is an element of $\vartheta \cup \{a\}$. Such an assignment is called a well defined assignment to an auxiliary variable. A sequence of well defined assignments to auxiliary variables is denoted $l_{(\vartheta,\alpha)}$. We define the concept of *program augmentation.*

**Definition 38** *Given the event-based system $(\vartheta, \alpha, \mathcal{B})$, a program $z_2$ is an augmentation of the program $z_1$ (denoted $z_1 \hookrightarrow z_2$) iff $z_2$ can be obtained from $z_1$ by the following substitutions:*

- *any assignment* $v := r$ *with* **await** *true* **do** $v := r; l_{(\vartheta,\alpha)}$ **od**, *where* $v := r$ *does not occur in an await statement and* $l_{(\vartheta,\alpha)}$ *is a sequence of well defined assignments to auxiliary variables;*

- *any statement of the form* **await** $b$ **do** $z$ **od** *with* **await** $b$ **do** $z'; l_{(\vartheta,\alpha)}$ **od**, *where* $z \hookrightarrow z'$ *holds and* $l_{(\vartheta,\alpha)}$ *is a sequence of well defined assignments to auxiliary variables.*

**Definition 39** *The formula* $z_1$ *sat* $(\vartheta, \alpha, \mathcal{B}) : :(P, R, W, G, E)$ *is valid iff there exists a program* $z_2$ *such that* $z_1 \hookrightarrow z_2$ *and* $cp[z_2, \mathcal{B}] \cap ext[(\vartheta, \alpha, \mathcal{B}), P, R] \Rightarrow int[(\vartheta, \alpha, \mathcal{B}), W, G, E]$.

**Definition 40** *The formula* $z_1$ *sat* $(\vartheta, \alpha) : :(P, R, W, G, E)$ *is valid iff:*

- $z_1$ *sat* $(\{\mathbf{skip}\}, \vartheta, \alpha, \mathcal{B}_0) : :(P, R, W, G, E)$ *is valid and*

- $events(z_1) = \{\}$.

To show that a program $z_1$ satisfies a specification, one constructs an augmentation of $z_1$ that satisfies this specification.

### 8.5.3 Deduction rules

Assignment rule.

The rule expresses the ability to replace an assignment $v := r$ with the sequence of assignments $v := r; a := u$ without changing the results of the program. One can notice that the conclusion of the rule remains the same as for the assignment rule without auxiliary variables.

$$\frac{\begin{array}{l} P \textbf{ stable when } R \\ E \textbf{ stable when } R \\ \overleftarrow{P} \wedge v = \overleftarrow{r} \wedge a = \overleftarrow{u} \wedge I_{(\vartheta \cup \alpha) \setminus \{v,a\}} \Rightarrow (G \vee I_{\vartheta \cup \alpha}) \wedge E \end{array}}{v := r \text{ sat } (\vartheta, \alpha) : :(P, R, \mathsf{false}, G, E).} \tag{8.12}$$

AWAIT RULE

The rule is obtained by a combination of the first await rule, the sequential rule, and the above assignment rule. By the sequential rule one deduces the validity of

$$\{z;\, a{:=}\, u\}\ \underline{sat}\ (P \wedge b,\ \mathsf{false},\ \mathsf{false},\ \mathsf{true},\ E \mid (I_{\vartheta \cup \alpha \setminus \{a\}} \wedge a = \overleftarrow{u})).$$

By the await-rule this program is embedded in an await-construct.

$$\frac{\begin{array}{l} P \textbf{ stable when } R \\ E_2 \textbf{ stable when } R \\ E_1 \mid (I_{\vartheta \cup \alpha \setminus \{a\}} \wedge a = \overleftarrow{u}) \ \Rightarrow\ (G \vee I_{\vartheta \cup \alpha}) \wedge E_2 \\ z\ \underline{sat}\ (\vartheta, \alpha, \mathcal{B}) ::(P \wedge b,\ \mathsf{false},\ \mathsf{true}, (G \vee I_\vartheta) \wedge E_1) \end{array}}{\textbf{await } b \textbf{ do } z \textbf{ od}\ \ \underline{sat}\ (\vartheta, \alpha, \mathcal{B}) ::(P, R, P \wedge \neg b, G, E_2)} \tag{8.13}$$

ELIMINATION RULE

The elimination rule allows removing some auxiliary variables from a specification. The variable to be eliminated should occur neither in $W$, $G$, nor in $E$.

$$\frac{z\ \ \underline{sat}\ (\vartheta, \alpha, \mathcal{B}) ::(P, R, W, G, E)}{z\ \ \underline{sat}\ (\vartheta, \alpha \setminus \{x\}, \mathcal{B}) ::(\exists x : P, \forall\ \overleftarrow{x} : \exists x : R, W, G, E)} \tag{8.14}$$

## 8.6 SUMMARY

Due to the similarities between event-based systems and concurrent systems, we experienced the necessity of eliciting the development process of deadlock free event-based applications. This chapter started by illustrating the use of the await construct in a context where deadlock is not likely to occur; only mutual exclusion is required and specified. Next, the chapter showed how to specify a component that possibly blocks. The specification

of a program is extended with a fifth component called wait-condition which characterizes the state in which a program is allowed to block.

Auxiliary variables were shown to be necessary for the specification of some concurrent systems. This chapter also discussed the combination of auxiliary variables and event-based systems.

We are convinced that many developers are not aware of the concurrency implied by the event-based paradigm. We suspect that most event-based applications do not really need strong concurrency. Some kind of coarse grained concurrency may be accepted by developers if the effort for developing such applications is less than it would be otherwise the case. Chapter 11 alleviates the difficulties in the development of event-based applications by imposing the constraint that for any two distinct subscribers to an event $e$, none of them modifies a variable that the other writes to or reads. The rely- and guar-conditions disappear under such a requirement giving rise to a specification technique that resembles the specification of sequential programs.

# Chapter 9

# Stack-Counter Example

We consider an example resembling that of Dingel et al. [35, 36]. The goal is to develop a system including a stack and a counter. Whenever an element is pushed on the stack, the counter must be incremented. Similarly, the counter must be decremented whenever an element is removed from the (top of the) stack. Although this seems to be a very simple example, it represents an important class of applications. A program executes its body and announces an event at the end of this execution as a means to notifying other components of this successful execution.

To make the development process resemble that of real event-based applications we start with a system that includes only a stack and the related operations. The counter is added as an evolution of the system.

In this example, and in most examples in this thesis, we do not perform the top-down development of components. It does not differ much from the traditional top-down development of rely/guarantee programs. We rather concentrate on illustrating how event-based applications can be composed at the abstract level, which is new. Specifications of components are constructed, local properties are checked in isolation, then the specifications are put together, and the global properties are checked.

## 9.1 Component Specification

### 9.1.1 Data Modeling

We define the datatypes needed for the construction of our application.

$$
\begin{array}{lcl}
\mathit{Element} & = & \mathit{token}; \\
\mathit{Stack} & = & \mathit{Element}*; \\
\mathit{Event} & :: & \mathit{name} : \mathit{EventName},\ \mathit{elt} :\ \mathit{Element}; \\
\mathit{Subscription} & = & \mathit{Event}\text{-set}; \\
\mathit{Prog} & = & \langle \mathit{impl\text{-}increment} \rangle \mid \langle \mathit{impl\text{-}push} \rangle \mid \langle \mathit{impl\text{-}decrement} \rangle \mid \langle \mathit{impl\text{-}pop} \rangle \mid \langle \mathit{skip} \rangle; \\
\mathit{EventName} & = & \langle \mathit{pushaction} \rangle \mid \langle \mathit{popaction} \rangle; \\
\mathit{Binding} & = & \mathit{Prog} \xrightarrow{m} \mathit{Subscription}\text{-set};
\end{array}
$$

An event is defined as a VDM composite type (record) which includes the name of the event and an element which is not further defined (declared as *token*). We introduce the enumeration type *Prog* for referring to operations defined in this specification and that may be elements of the EB system's set of programs. Further, a subscription is a set of events that a subscriber is interested in. Finally, a binding associates each program (element of type *Prog*) to its set of subscriptions. The state of the EB system is composed of the variables *stack*, *count*, and *elt*. The binding is not yet defined since the structural specifications of the components are constructed separately.

**System**

$$
\begin{array}{lcl}
\mathit{stack} & : & \mathit{Stack}; \\
\mathit{counter} & : & \mathbb{N}; \\
\mathit{elt} & : & \mathit{Element};
\end{array}
$$

### 9.1.2 Specification of Components

Next, we propose the specifications *push* and *pop* for the operations *impl-push* and *impl-pop* for adding and removing elements to and from the stack. Two points indicate that they are structural specifications. First the binding is not constructed. Next, the specification of *push* is composed of that of *simple-push* and an announcement construct.

**simple-push**() $\triangleq$
pre true
rely $\mathit{stack} = \overleftarrow{\mathit{stack}} \wedge \mathit{elt} = \overleftarrow{\mathit{elt}}$
guar $\mathit{stack} = [\mathit{elt}] \frown \overleftarrow{\mathit{stack}} \wedge I_{\vartheta \setminus \{\mathit{stack}\}}$
post $\mathit{stack} = [\mathit{elt}] \frown \overleftarrow{\mathit{stack}} \wedge \mathit{elt} = \overleftarrow{\mathit{elt}}$

**push**() $\triangleq$
*simple-push*();
**announce**(*mk-Event*($\langle \mathit{pushaction} \rangle, \mathit{elt}$));

These specifications are not difficult to understand. Push for instance, does *simple-push* followed by the announcement of an event whose name is $\langle \mathit{pushaction} \rangle$ and whose payload is

the pushed element. On the other hand, *simple-push* specifies the addition of an element to the stack. The post-condition ensures that the new stack consists of the old stack augmented with a new element. The rely-condition requires that the stack is not changed by the environment.

Analogously, *pop* does *simple-pop* and subsequently announces an event named $\langle popaction \rangle$ with the removed element as payload.

**simple-pop**() $\triangleq$
  pre $stack \neq []$
  rely $stack = \overleftarrow{stack} \wedge elt = \overleftarrow{elt}$
  guar $stack = \mathsf{tl}\ \overleftarrow{stack} \wedge I_{\vartheta \setminus \{stack, elt\}} \wedge elt = \mathsf{hd}\ \overleftarrow{stack}$
  post $stack = \mathsf{tl}\ \overleftarrow{stack} \wedge elt = \mathsf{hd}\ \overleftarrow{stack}$

**pop**() $\triangleq$
  *simple-pop*;
  **announce**(*mk-Event*($\langle popaction \rangle$, *elt*));

## 9.2 Verification of Local Properties

As local property, we would like to show for instance that *impl-pop* reverses *impl-push*. That is, if we add an element to the stack and apply *impl-pop* next, the resulting stack is equal to the initial stack.

### 9.2.1 Derivation of Behavioral specifications

We saw that structural specifications are not suitable for checking properties of systems. For instance it is not obvious what is the post-condition of *impl-push*. The specifications, therefore, need to be transformed to behavioral specifications. For the purpose of verifying local properties, we assume the empty binding $\mathcal{B}_0$, i.e. the binding that maps any program different from **skip** to the empty set of events but **skip** to all events. Formally,

$$\mathcal{B}_0 \triangleq \{\langle \textbf{impl-push} \rangle \mapsto \{\}, \langle \textbf{impl-pop} \rangle \mapsto \{\}, \langle \textbf{skip} \rangle \mapsto \{x : Event\}\}$$

From this binding, it is clear that for any event $e$,

$$subscribers(e) = \{\textbf{skip}\}.$$

And, applying the announce rule followed by the skip rule, the following specifications are derived:

$$\textbf{behavioral-push}() \triangleq \textit{simple-push}(); \qquad \textbf{behavioral-pop}() \triangleq \textit{simple-pop}();$$

Our goal is, therefore, to show that the program *impl-pushpop* is the identity operation.

$$\textit{impl-pushpop} \triangleq \textit{impl-push}(); \textit{impl-pop}()$$

We call *pushpop* the specification of *impl-pushpop*. By the definition of structural specifications, is it clear that *impl-pushpop* satisfies the following structural specification:

$$\textit{pushpop} \triangleq \textit{behavioral-push}(); \textit{behavioral-pop}()$$

which we naturally expand to:

$$\textit{pushpop} \triangleq \textit{simple-push}(); \textit{simple-pop}()$$

The sequential rule requires discharging the following proof obligation which trivially holds due to the validity of its conclusion.

**Proof Obligation 7** **post**-*simple-push* $\Rightarrow$ **pre**-*simple-pop*

The proof obligation of interest is, therefore, formulated as:

**Proof Obligation 8** **post**-*simple-push*() | **post**-*simple-pop*() $\Rightarrow$ $\textit{stack} = \overleftarrow{\textit{stack}}$.

To discharge this proof obligation, we first simplify it and transform it to:

**Proof Obligation 9** $\textit{stack} = [\textit{elt}] \frown \overleftarrow{\textit{stack}} \mid \textit{stack} = \textit{tl}\ \overleftarrow{\textit{stack}} \Rightarrow \textit{stack} = \overleftarrow{\textit{stack}}$

Which trivially results from the application of the definition of the assertion composition operator |.

To summarize, we extracted the local properties of the components *impl-push* and *impl-pop*. We showed that in the empty binding, *impl-pop* reverses *impl-push*.

## 9.3 Application Composition

This step consists of subscribing components to events. Such components may be newly developed and inserted in the system or existing and already subscribed to other events. In the first case, this may be viewed as an evolution/maintenance operation. This also corresponds to the composition of the system by the integration of the developed components. In the second case this can be either the reconfiguration of an existing application or simply the construction of an application starting with some "off-the-shelf" components.

All these cases are captured in our development process through the subscription of components to events. We want to include the operations for incrementing and decrementing the counter into our event-based application.

### 9.3.1 Specification of new components

We propose the following specifications for the operations *impl-increment* and *impl-decrement* for incrementing and decremeting the counter.

**decrement**() $\triangleq$
pre $counter > 0$
rely $counter = \overleftarrow{counter}$
guar $counter = \overleftarrow{counter}\text{-}1 \wedge I_{\vartheta \setminus \{counter\}}$
post $counter = \overleftarrow{counter}\text{-}1$

**increment**() $\triangleq$
pre true
rely $counter = \overleftarrow{counter}$
guar $counter = \overleftarrow{counter} + 1 \wedge I_{\vartheta \setminus \{counter\}}$
post $counter = \overleftarrow{counter} + 1$

Fortunately, these specifications are so trivial that there is no local properties to be checked about them. We would have assumed the empty binding again, transformed the specifications into behavioral specifications, and discharged these properties.

By the integration rule we can insert these operations into the binding which becomes:

$$\mathcal{B}_1 \triangleq \{\langle \textbf{impl-push} \rangle \mapsto \{\}, \langle \textbf{impl-decrement} \rangle \mapsto \{\}, \langle \textbf{impl-increment} \rangle \mapsto \{\}, \langle \textbf{impl-pop} \rangle \mapsto \{\}, \langle \textbf{skip} \rangle \mapsto \{x : Event \mid \textsf{true}\}\}$$

Following Section 7.3 that defines the process of integrating and subscribing a component, three points must be investigated next:

- subscribing the component to the event of interest,
- identifying programs that possibly announce this event,
- transforming the structural specifications of these components into behavioral specifications while discharging the required interference freedom POs.

### 9.3.2 Subscription of new components

We want to achieve the effect of counting the elements of the stack using the counter. For this, we subscribe the component *impl-increment* to events named $\langle pushaction \rangle$. The expected effect is the increment of the counter whenever an element is pushed on the stack. Similarly, we want to achieve the decrement of the stack whenever an element is removed from the stack. We, therefore, subscribe the program *impl-decrement* to the events with name $\langle popaction \rangle$.

The binding now looks as follows:

$$\begin{aligned} \mathcal{B}_2 \quad \triangleq \quad \{ \quad & \langle \textbf{impl-push} \rangle \mapsto \{\}, \langle \textbf{impl-decrement} \rangle \mapsto \{e : Event \cdot e.name = \langle popaction \rangle\}, \\ & \langle \textbf{impl-increment} \rangle \mapsto \{e : Event \cdot e.name = \langle pushaction \rangle\}, \\ & \langle \textbf{impl-pop} \rangle \mapsto \{\}, \langle \textbf{skip} \rangle \mapsto \{x : Event\}\} \end{aligned}$$

It results that for any events $e_1$ and $e_2$ such that $e_1.name = \langle pushaction \rangle$ and $e_2.name = \langle popaction \rangle$, the function *subscribers* that defines the subscribers of an event is defined as follows:

$$\begin{aligned} subscribers(e_1) &= \{\langle \textbf{impl-increment} \rangle, \langle \textbf{skip} \rangle\} \\ subscribers(e_2) &= \{\langle \textbf{impl-decrement} \rangle, \langle \textbf{skip} \rangle\} \end{aligned}$$

Since, however, by the skip rule, $skip \| z$ behaves as $z$ we can reduce the above definition to:

$$\begin{aligned} subscribers(e_1) &= \{\langle \textbf{impl-increment} \rangle\} \\ subscribers(e_2) &= \{\langle \textbf{impl-decrement} \rangle\} \end{aligned}$$

### 9.3.3 Identification of affected Components

We identify the components whose behaviors may be affected by the subscription of the programs *impl-increment* and *impl-decrement* to the events named $\langle pushaction \rangle$ and $\langle popaction \rangle$.

Trivially, the program *impl-push* is affected by the subscription of *impl-increment* while *impl-pop* is affected by subscription of *impl-decrement*.

Although this step is easy in the case of this application, this is not the case in general. Consider a component with the structural specification $\vartheta \cup \{v\} :: (P, R, G, E)$; **announce**$(v)$. The events announced by this program are determined by the first part which satisfies $\vartheta \cup \{v\} :: (P, R, G, E)$. The set of events possibly announced by this program is, therefore, the values that the variable $v$ may take in states satisfying the post-condition $E$ (see Chapter 13 for instance).

### 9.3.4 Derivation of Behavioral specifications

New subscriptions were added to the binding. We, therefore, need to show that this does not result in interference. In particular, the behavior of components that may be affected by these subscriptions must be revised. This is done by transforming the structural specifications of these components into behavioral specifications.

We consider the case of *push* first. For any event $e_1$ such that $e_1.name = \langle pushaction \rangle$ , it was shown in the previous section that

$$subscribers(e_1) = \{\langle \textbf{impl-increment} \rangle\}$$

By the announce rule one deduces the following behavioral specification of *impl-push*:

$$\textbf{behavioral-push}() \triangleq \; simple\text{-}push(); increment();$$

The sequential rule requires discharging the proof obligation:

**Proof Obligation 10** *post-simple-push* $\Rightarrow$ *pre-increment*

which indeed holds since pre-increment is true.

Next, we revise the behavior of *impl-pop*; we determine its behavioral specification first. For any event $e_2$ such that $e_2.name = \langle popaction \rangle$, the following equality holds.

$$subscribers(e_2) = \{\langle \textbf{impl-decrement} \rangle\}$$

By the announce rule one deduces the following behavioral specification:

$$\textbf{behavioral-pop}() \triangleq \textit{simple-pop}(); \textit{decrement}()$$

The sequential rule, however requires that the pre-condition of *decrement* follows from the *post-condition* of *simple-pop* which we formulate as:

**Proof Obligation 11** $\textbf{post-}\textit{simple-pop}() \Rightarrow \textbf{pre-}\textit{decrement}()$

It is, however, easy to see that this proof obligation does not hold. If a program satisfying *simple-pop* is started in a state where $counter < 0$, it also terminates in such a state (if the environment does not modify *counter*) which does not result in the pre-condition of *impl-decrement*. We, therefore, need to refine the specification of *simple-pop*. Further, we add more information into the post-condition by means of the pre- and post- rules as follows:

$$\begin{aligned}
&\textbf{simple-pop1}() \triangleq \\
&\quad \textsf{pre} \;\; stack \neq [] \wedge counter > 0 \\
&\quad \textsf{rely} \;\; stack = \overleftarrow{stack} \wedge elt = \overleftarrow{elt} \wedge counter > 0 \\
&\quad \textsf{guar} \;\; stack = \textsf{tl}\ \overleftarrow{stack} \wedge I_{\vartheta \setminus \{stack, elt\}} \wedge elt = \textsf{hd}\ \overleftarrow{stack} \\
&\quad \textsf{post} \;\; stack = \textsf{tl}\ \overleftarrow{stack} \wedge elt = \textsf{hd}\ \overleftarrow{stack} \wedge (\textbf{rely} \vee \textbf{guar})^+ \wedge \overleftarrow{\textbf{pre}}
\end{aligned}$$

and the behavioral specification of *impl-pop* correspondingly evolves into:

$$\textbf{pop1}() \triangleq \quad \textit{simple-pop1}; \textit{decrement}();$$

And the proof obligation becomes:

**Proof Obligation 12** $stack = \textit{tl}\ \overleftarrow{stack} \wedge elt = \textit{hd}\ \overleftarrow{stack} \wedge (\textbf{rely} \vee \textbf{guar})^+ \wedge \overleftarrow{\textbf{pre}} \Rightarrow counter > 0$

in which **pre**-*simple-pop*1(), **rely**-*simple-pop*1(), and **guar**-*simple-pop*1() are abbreviated as **pre**, **rely**, and **guar** respectively.

To discharge this proof obligation, we simplify it by

- removing any conjunct that does not concern the counter (using the fact that $A \Rightarrow C$ results into $A \wedge B \Rightarrow C$).
- observing that $counter = \overleftarrow{counter} \vee counter > 0$ follows from $(\mathbf{rely} \vee \mathbf{guar})^+$

The proof obligation is now given as:

**Proof Obligation 13** $\overleftarrow{counter} > 0 \wedge (counter = \overleftarrow{counter} \vee counter > 0) \Rightarrow counter > 0.$

which also trivially holds.

## 9.4 Verification of Global Properties

We have completed the specification of the application in a way that interference freedom is ensured. We now want to check some global properties; those properties that motivated the subscription of some components to some events. In our case, we wanted to ensure the counting of the elements of the stack.

The property of interest can be divided into two requirements:

- pushing an element results in a situation where the counter reflects the length of the stack,
- removing an element results in a situation where the counter reflects the length of the stack,

These requirements are formulated as:

**Proof Obligation 14** $\overleftarrow{pre\text{-}push} \wedge post\text{-}push \Rightarrow counter = \mathsf{len}\ (stack).$

**Proof Obligation 15** $\overleftarrow{pre\text{-}pop} \wedge post\text{-}pop \Rightarrow counter = \mathsf{len}\ (stack).$

Let us investigate the first of these POs that we attempt to discharge based on the behavioral specification *behavioral-push*. It is, however, easy to see that this PO does not hold; the program may be started in a state where the value of the counter is different from the size of the stack. We, therefore, need to further strengthen the assumptions on the environment by refining *simple-push*. We also apply the pre- and post- rules to add more information into the post-conditions. The specification now looks as follows:

**behavioral-push()**

pre len $(stack) = counter$

rely $stack = \overleftarrow{stack} \wedge counter = \overleftarrow{counter}$

guar $G \vee G'$

post $stack = [elt] \frown \overleftarrow{stack} \wedge (I_\vartheta \vee G)^+ \mid counter = \overleftarrow{counter} + 1 \wedge (I_\vartheta \vee G')^+$

To discharge the PO, we observe that:

- $counter = \overleftarrow{counter} + 1 \wedge (I_\vartheta \vee G')^+$ results in $counter = \overleftarrow{counter} + 1 \wedge I_{\vartheta\{counter\}}$ and
- $stack = [elt] \frown \overleftarrow{stack} \wedge (I_\vartheta \vee G)^+$ results in $stack = [elt] \frown \overleftarrow{stack} \wedge I_{\vartheta\{stack\}}$

This observation leads to rewriting the PO as:

**Proof Obligation 16** $(\overleftarrow{counter} = len\ (\overleftarrow{stack}) \wedge stack = [elt] \frown \overleftarrow{stack} \wedge counter = \overleftarrow{counter}) \mid counter = \overleftarrow{counter} + 1 \wedge stack = \overleftarrow{stack} \Rightarrow counter = len\ (stack)$

The validity of the PO is obtained by applying the definition of the composition operator | and observing that adding an element to the stack results in increasing its length by 1.

The proof of the second PO follows the same steps and is omitted.

## 9.5 Component Implementation

We propose the following implementations for the above programs. We assume that each assignment is atomic.

impl-simple-push $\triangleq$ stack:=[elt]$\frown$ stack

impl-push $\triangleq$ impl-simple-push; **announce**(mk-Event(⟨*pushaction*⟩,elt))

impl-increment $\triangleq$ counter:= counter+1

impl-simple-pop $\triangleq$ elt:=hd stack; stack:=tl stack

impl-pop $\triangleq$ impl-simple-pop; **announce**(mk-Event(⟨*popaction*⟩,elt))

impl-decrement $\triangleq$ counter:= counter-1

We need to show that *impl-push* satisfies the specification *push*, that *impl-increment* satisfies *increment*, that *impl-pop* satisfies *pop* and that *impl-decrement* satisfies *decrement*.

The proof that *impl-increment* satisfies *increment* is done as in traditional rely/guarantee approaches [34, 77, 94, 121, 136]. Similar are the proofs that *impl-decrement* satisfies *decrement*, that *impl-simple-pop* satisfies *simple-pop* and that *impl-simple-push* satisfies *simple-push*. We omit them.

By application of the definition of $z$ *sat* $\vartheta :: S_1; S_2$ it results that *impl-push* satisfies *push* and that *impl-pop* satisfies *pop*.

## 9.6 Summary

This chapter presented a first basic example in the design, analysis and implementation of a correct event-based application. Although this is a simple example many argue that it represents an important class of systems in event-based applications [56]. In particular, it represents the class of applications where a component modifies its state and subsequently notifies interested subscribers of the successful completion of the operation. The structural specification of such components is typically of the form $\vartheta :: (P, R, G, E); \mathbf{announce}(e)$. The steps required for developing such applications are the same that we followed in this chapter.

We showed how to discharge the local properties of components by assuming an empty binding. Then, we showed how to add new components in the event-based system, hence, composing the specification of the application starting from those of the components. The subscription of such components may lead to interference in the system. We discussed how to detect the required proof obligations and how to discharge them. The subscription of a component to some event is often certainly motivated by some effect that must be achieved. This effect is captured as global property of the system; we showed how to tackle such issues. Finally, we showed that the stepwise development of components that announce events is not very different from that of traditional rely/guarantee components.

Although it is based on rely/guarantee conditions, the example presented in this chapter allows no program to run concurrently. This may be seen as abusing rely-/guarantee conditions. In the next chapter, we discuss a variation of this system that is more tolerant in the sense that it allows more applications to be executed concurrently.

# Chapter 10

# Mutual Exclusion in the Stack-Counter Example

The version of the stack-counter example presented in the previous chapter requires that any program running concurrently should modify neither the stack, the element, nor the counter. This restriction may be unacceptable for some settings. This chapter shows how this requirement can be alleviated through the use of the await construct. Programs are now executed atomically such that internal transitions are not visible to the environment which may interfere either after or before the execution of some program.

## 10.1 Component Specification

The goal remains that of designing a system composed of a stack and a counter. The program *impl-push* is intended to add an element to the stack while *impl-pop* removes the element on the top of the stack. Unlike in the previous model, we want to increase the number of applications that may be executed in parallel with *impl-push* and *impl-pop*. The above requirement that no other application should concurrently write to the stack is too strong. The solution proposal consists of embedding the specifications presented in the last chapter in some await-constructs. We propose the following specifications:

**simple-push**() $\triangleq$
  pre true
  rely false
  guar true
  post $stack = [elt] \frown \overleftarrow{stack} \wedge elt = \overleftarrow{elt}$

**push**() $\triangleq$
  **await** true **do**
    *simple-push*();
    **announce**(*mk-Event*($\langle pushaction \rangle$, *elt*))
  **od**

**simple-pop**() $\triangleq$
pre $stack \neq []$
rely false
guar true
post $stack = \mathsf{tl}\ \overleftarrow{stack} \wedge elt = \mathsf{hd}\ \overleftarrow{stack}$

**pop**() $\triangleq$
**await** true **do**
*simple-pop*;
**announce**(*mk-Event*($\langle$*popaction*$\rangle$, *elt*))
**od**

The specifications are still easy to understand; any program that implements *push* performs *simple-push* and announces the event *mk-Event*($\langle$*pushaction*$\rangle$, *elt*); however, all in an atomic step. Any implementation of *simple-push* adds an element to the stack provided there is no interference; and, the await construct indeed ensures that there will be no interference.

Analogously, we can change the specification of *impl-increment* and *impl-decrement* such that they rely on nothing to be done in parallel. Since the execution of the await construct includes those of subscribers, *impl-increment* and *impl-decrement* will be executed as part of the execution of the announcement constructs of *impl-push* and *impl-pop* respectively. It is, therefore, realistic to assume false from the environment and to guarantee true.

**decrement**() $\triangleq$
pre $counter > 0$
rely false
guar true
post $counter = \overleftarrow{counter}\text{-}1$

**increment**() $\triangleq$
pre true
rely false
guar true
post $counter = \overleftarrow{counter} + 1$

## 10.2 Verification of Local Properties

Our local property of interest remains checking that *impl-pop* indeed reverses *impl-push*. Verifying local properties is done by assuming the empty binding that relates the program **skip** to all events and any event to the unique program **skip**.

For any event $e$, $subscribers(e) = \{\mathbf{skip}\}$, therefore, holds; resulting by the skip rule in reducing the behavior of *simple-push*; **announce**(*mk-Event*($\langle$*pushaction*$\rangle$, *elt*)) to that of *simple-push*. Similarly, *simple-pop*; **announce**(*mk-Event*($\langle$*popaction*$\rangle$, *elt*)) is reduced to *simple-pop*.

The behavioral specifications of *push* and *pop* are, therefore:

**behavioral-push**() $\triangleq$
  **await** true **do**
    *simple-push*;
  **od**

**behavioral-pop**() $\triangleq$
  **await** true **do**
    *simple-pop*;
  **od**

An application of the definition of await structural specifications followed by the application of the await rule with the parameters given below allows rewriting this specification as:

**behavioral-push**() $\triangleq$
  pre true
  rely $I_\vartheta$
  guar true
  post $stack = [elt] \frown \overleftarrow{stack} \wedge elt = \overleftarrow{elt}$

**behavioral-pop**() $\triangleq$
  pre $stack \neq []$
  rely $I_\vartheta$
  guar true
  post $stack = \mathsf{tl}\ \overleftarrow{stack} \wedge elt = \mathsf{hd}\ \overleftarrow{stack}$

- $R \stackrel{def}{=} I_\vartheta$,
- $G \stackrel{def}{=}$ true,
- $b =$ true,
- $P \stackrel{def}{=}$ true for the case of *push* and $P \stackrel{def}{=} stack \neq []$ for the case of *pop*.

The proof obligation, therefore, consists of showing that the following requirement holds:

**Proof Obligation 17** $stack = [elt] \frown \overleftarrow{stack} \mid stack = \mathsf{tl}\ \overleftarrow{stack} \wedge elt = \mathsf{hd}\ \overleftarrow{stack} \Rightarrow stack = \overleftarrow{stack}$.

This, however, directly results from the application of the definition of $\mid$ and tl .

## 10.3 Application Composition

The step consists of subscribing programs to events, deriving behavioral specifications, and discharging the required proof obligations.

### 10.3.1 SUBSCRIPTIONS

As in the previous chapter, we subscribe the program *impl-increment* to events named $\langle pushaction \rangle$ and *impl-decerement* to events named $\langle popaction \rangle$. The resulting binding is the following:

$$\begin{array}{lll} \mathcal{B}_2 & \triangleq & \{ \ \langle \textbf{impl-push} \rangle \mapsto \{\}, \langle \textbf{impl-decrement} \rangle \mapsto \{e : Event \cdot e.name = \langle popaction \rangle\}, \\ & & \ \ \langle \textbf{impl-increment} \rangle \mapsto \{e : Event \cdot e.name = \langle pushaction \rangle\}, \\ & & \ \ \langle \textbf{impl-pop} \rangle \mapsto \{\}, \langle \textbf{skip} \rangle \mapsto \{x : Event \mid \textsf{true}\}\} \end{array}$$

leading to the following equalities where $e_1$ is an event such that $e_1.name = \langle pushaction \rangle$ and $e_2$ is an event such that $e_2.name = \langle popaction \rangle$.

$$\begin{array}{l} subscribers(e_1) = \{\langle \textbf{increment} \rangle\} \\ subscribers(e_2) = \{\langle \textbf{decrement} \rangle\} \end{array}$$

Intuitively, the program *impl-increment* is interested in events announced by *impl-push* while *impl-decrement* is interested in events announced by *impl-pop*. The resulting behavior should be the increment of the counter whenever an element is added on the stack and its decrement whenever an element is removed from the stack.

### 10.3.2 IDENTIFICATION OF AFFECTED COMPONENTS

Subscribing *impl-increment* and *impl-decrement* to events with names $\langle pushaction \rangle$ and $\langle popaction \rangle$ leads to a modification of the behavior of *impl-push* and *impl-pop* which possibly announce these events.

Their behaviors must, therefore, be revised and the necessary proof obligations discharged.

### 10.3.3 DERIVATION OF BEHAVIORAL SPECIFICATIONS

Let us investigate the changes on the behavior of *impl-push*. Since the set of subscribers to the event it announces can be reduced to *increment*, the announce rule can be applied to yield the following behavioral specification.

```
behavioral-push() ≜
    await true do
            simple-push(); increment();
    od
```

By application of the definition of structural sequential specifications, the sequential rule, the definition of structural await specifications, and the await rule (with the parameters $P$, $R$, $G$, and $b$ given below), we derive the following behavioral specifications:

**behavioral-push**() $\triangleq$
pre $\mathsf{len}\ (stack) = counter$
rely $\mathsf{len}\ (stack) = counter$
guar $stack = [elt] \frown \overleftarrow{stack} \wedge I_{\vartheta \setminus \{stack\}} \mid counter = \overleftarrow{counter} + 1 \wedge I_{\vartheta \setminus \{counter\}}$
post $\overleftarrow{\mathbf{pre}} \wedge \mathbf{rely}^+ \mid stack = [elt] \frown \overleftarrow{stack} \wedge I_{\vartheta \setminus \{stack\}} \mid counter = \overleftarrow{counter} + 1 \wedge I_{\vartheta \setminus \{counter\}} \mid \mathbf{rely}^+$

- $R \stackrel{def}{=} \mathsf{len}\ (stack) = counter$,
- $G \stackrel{def}{=} stack = [elt] \frown \overleftarrow{stack} \wedge I_{\vartheta \setminus \{stack\}} \mid \wedge counter = \overleftarrow{counter} + 1 \wedge I_{\vartheta \setminus \{counter\}}$,
- $b = \mathsf{true}$,
- $P \stackrel{def}{=} \mathsf{len}\ (stack) = counter$.

The proof obligation for the application of the sequential rule was ignored since; it holds because of the validity of *pre-increment*.

A similar process leads the following behavioral specification of *impl-pop*:

**behavioral-pop**() $\triangleq$
pre $\mathsf{len}\ (stack) = counter \wedge counter > 0 \wedge stack \neq []$
rely $\mathsf{len}\ (stack) = counter \wedge counter > 0 \wedge stack \neq []$
guar $stack = \mathsf{tl}\ \overleftarrow{stack} \wedge I_{\vartheta \setminus \{stack, elt\}} \wedge elt = \mathsf{hd}\ \overleftarrow{stack} \mid counter = \overleftarrow{counter}\text{-}1 \wedge I_{\vartheta \setminus \{counter\}}$
post $\overleftarrow{\mathbf{pre}} \wedge \mathbf{rely}^+ \mid stack = \mathsf{tl}\ \overleftarrow{stack} \wedge elt = \mathsf{hd}\ \overleftarrow{stack} \wedge I_{\vartheta \setminus \{stack, elt\}} \mid$
$counter = \overleftarrow{counter} + 1 \wedge I_{\vartheta \setminus \{counter\}} \mid \mathbf{rely}^+$

Concerning the parallel execution of these programs, any number of programs satisfying *push* can indeed be executed in parallel: **guar**-*push* $\Rightarrow$ **rely**-*push* holds. Further, any program satisfying *pop* can also be executed in parallel with any number of programs satisfying *push*. However, only one program satisfying *pop* can be executed at any one time. The rely-condition of *pop* does not follow from its guarantee condition.

## 10.4 Global System Behavior

Having completed the specification of our application, we can now verify some of its properties. In particular, we want to show that the counter indicates the size of the stack.

This requirement is, however, easily verified as resulting from the post-condition of the above behavioral specifications. The proof is done by observing that:

- **rely** | **rely** $\Rightarrow$ **rely**,
- applying the definition of |, and
- observing that:
  - adding an element on the stack results in increasing its length by 1, i.e. $stack = [elt] \frown \overleftarrow{stack} \Rightarrow \mathsf{len}\ (stack) = \mathsf{len}\ (\overleftarrow{stack}) + 1$;
  - removing an element from the stack results in decreasing its length by 1, i.e. $stack = \mathsf{tl}\ \overleftarrow{stack} \Rightarrow \mathsf{len}\ (stack) = \mathsf{len}\ (\overleftarrow{stack})\text{-}1$;

Although this specification satisfies our global property of interest, there is, however, a subtlety. The element added on the stack may not be the intended element. Let us consider the following specification:

**malicious**()
pre true
rely true
guar $elt = Element_0 \wedge stack = [] \wedge counter = 0$
post true

that characterizes a component supposed to assign the value $Element_0$ to *elt*, empty the stack, and set the counter to 0. If a program satisfying this specification, say *impl-malicious*, is executed concurrently with the program *impl-push*, it is possible that *impl-malicious* may be executed before *impl-push*; leading to the addition of $Element_0$ to the stack. This is to support the argument that one has to be careful about the intended behavior of a component, its real behavior, and the behavior of the environment.

## 10.5 Component Implementation

We propose the following implementations for the above programs.

impl-simple-push $\triangleq$ stack:=[elt] $\curvearrowright$ stack

impl-push $\triangleq$ **await true do** impl-simple-push; **announce**(mk-Event(⟨*pushaction*⟩,elt)) **od**

impl-increment $\triangleq$ counter:= counter+1

impl-simple-pop $\triangleq$ elt:=hd stack; stack:=tl stack

impl-pop $\triangleq$ **await true do** impl-simple-pop; **announce**(mk-Event(⟨*popaction*⟩,elt))**od**

impl-decrement $\triangleq$ counter:= counter-1

By successively applying the definition of the validity of $z$ *sat* **await** $b$ **do** $S_1$ **od** and that of $z$ *sat* $S_1; S_2$ the proof that *impl-push* satisfies *push* is done by showing that *impl-simple-push* satisfies the specification *push*. This is, however, done in the traditional rely/guarantee manner and can be omitted.

## 10.6 Summary

This chapter presented an improvement of the stack-counter example designed in the last chapter. The await construct was used to allow the atomic execution of *push* (respectively *pop*) and its subscribers, resulting in programs that allow more programs to be executed concurrently.

The chapter also contributed to arguing on the systematic nature of the methodology. The same steps that were applied for the last version of the example were also applied:

- independent specification of components,
- verification of local properties,
- composition of the application; including transformation of structural specifications into behavioral specifications and verification of related requirements,
- verification of global properties,
- implementation/refinement of the components.

The example presented in this chapter suffers from an important limitation. As we have shown through the *malicious* example, a program running is parallel could satisfy the rely-condition while modifying the intention of the specified program. In this particular example, this was possible because programs satisfying *malicious* have access to the global

variable *elt* used as input variable. This directly touches the issue of method that is crucial for event-based systems.

In fact, a subscriber must be a method that is invoked by the event-based system. The subscriber may, therefore, specify in which variable (called formal parameter) it wants the passed value to be stored. Such a variable may be a local variable (call-by-value) on which only the invoked method has access.

Chapter 12 presents a basic solution to the issue of method invocation in LECAP in general and in SEATY in particular.

# Chapter 11

# Sequential Event-Based Applications

## 11.1 Overview

The operational semantics of the LECAP programming language presented in Chapter 5 shows that event-based systems share some substantial properties with concurrent systems. It is, therefore, not surprising that constructing event-based applications is not a trivial task. To alleviate these difficulties, we propose to simplify the construction of event-based applications by identifying relevant architectural types. An architectural type [13] is obtained from an architectural style by fixing some of its parameters which may be components, connectors, or behaviors. An architectural type is at an intermediary level of abstraction between the architectural style and an architecture.

In this chapter we introduce an architectural type called SEATY (**S**equential **E**vent-based **A**rchitectural **TY**pe), which is obtained from the event-based architectural style by requiring that no two distinct subscribers share some variable. This is a coarse interference freedom requirement that allows designing applications without explicit need for the rely and guarantee conditions. We believe that this architectural type includes an important class of event-based applications. As one may expect, the development of this kind of applications is fairly simpler than pure rely/guarantee applications.

The name sequential event-based architectural type should not mislead, the SEATY type does exclude neither concurrency nor distribution. The set of variables that two programs access may be distributed on different computers as well as two programs may run in parallel, but accessing different set of variables. We refer to this type as sequential simply because of the similarities between the specification techniques of SEATY programs and that of sequential programs.

In fact, concurrent systems with such restrictions have been widely studied under the designation of action systems [115, 8, 9, 116]. "An action system is a parallel or distributed program where parallel activity is described in terms of events, so called actions. The actions are atomic: if an action is chosen for execution, it is executed to completion without

any interference from the other actions in the system. Several actions can be executed in parallel, as long as the actions do not share any variables [8]". The argument for such systems is that their use permits the design of systems to be separated from the issue of how the systems is to be implemented; using shared variables or not.

The remainder of the chapter is organized as follows. The next section describes the differences between SEATY specifications and pure LECAP specifications. In section 11.3, we present the rules for the top-down development of SEATY components. These rules are simplifications of general LECAP rules. Section 11.4 discusses the composition of SEATY structural specifications and the manipulation of behavioral specifications while Section 11.5 illustrates the SEATY development process by means of a symbolic example. Finally, 11.6 summarizes and concludes the chapter.

## 11.2 SPECIFICATION OF SEATY PROGRAMS

Informally, specifications of SEATY programs differ from that of pure LECAP programs by the missing rely- and guar-conditions. In particular, we still have structural and behavioral specifications.

### 11.2.1 BEHAVIORAL SEATY SPECIFICATIONS

**Definition 41** *A behavioral* SEATY *specification is a formula* $(\vartheta, \mathcal{B}) ::(P,\ E)$ *where:*

- $P$ *is a unary assertion and* $E$ *is a binary assertion,*
- *any variable occurring free in* $P$ *or* $E$ *is in* $\vartheta$.

The set $\vartheta$ represents the set of variables that programs satisfying this specification are allowed to access.

An event-based system remains defined as a tuple $(\vartheta,\ \mathcal{B})$ where $\vartheta$ is a set of variables that programs access, and $\mathcal{B}$ is a binding of programs to events. In the definition of a SEATY specification, the tuple $(\vartheta_1, \mathcal{B})$ implicitly defines any event-based system of the form $(\vartheta \cup \vartheta_1,\ \mathcal{B})$ where the set of variables accessed by programs in $\mathcal{B}$ is a subset of $\vartheta \cup \vartheta_1$; that is, $\bigcup_{z \in \mathsf{dom}\ \mathcal{B}} \vartheta_z \subseteq \vartheta \cup \vartheta_1$.

**Definition 42** *We say that the set of variables* $\vartheta$ *covers the binding* $\mathcal{B}$ *iff for any subscriber* $z$ *in* $\mathcal{B}$*, the set of variables that it accesses is a subset of* $\vartheta$*. That is,* $\bigcup_{z \in \mathsf{dom}\ \mathcal{B}} \vartheta_z \subseteq \vartheta$.

A SEATY specification such as $(\vartheta_1, \mathcal{B}) : :(P,\ E)$ is a reformulation of the pure LECAP specification $(\vartheta \cup \vartheta_1, \mathcal{B}) : :(P,\ I_{\vartheta_1},\ I_{\vartheta \setminus \vartheta_1},\ E)$ iff $\vartheta \cup \vartheta_1$ covers $\mathcal{B}$ and $\vartheta \cap \vartheta_1 = \{\}$. Similarly, $\vartheta_1 : :(P,\ E)$ is a reformulation of the pure LECAP specification $\vartheta \cup \vartheta_1 : :(P,\ I_{\vartheta_1},\ I_{\vartheta \setminus \vartheta_1},\ E)$ if $\vartheta \cap \vartheta_1 = \{\}$. A program that satisfies this specification is a program that relies on that the environment will not change the variables that it has access to. On the other hand, such a program guarantees to change none of the variable it is not allowed to access.

### 11.2.2 EXTENDED BEHAVIORAL SPECIFICATIONS

A extended behavioral SEATY specification may be of one of the forms $(\vartheta, \mathcal{B}) : :S_1 \| S_2$, $(\vartheta, \mathcal{B}) : :S_1; S_2$, $(\vartheta, \mathcal{B}) : :$**await** $b$ **do** $S_1$ **od**, and $(\vartheta, \mathcal{B}) : :$**if** $b$ **then** $S_1$ **else** $S_2$ **fi** if $(\vartheta, \mathcal{B}) : :S_1$ and $(\vartheta, \mathcal{B}) : :S_2$ are behavioral (including extended ones) SEATY specifications.

Semantics is given to behavioral SEATY specifications by converting them to LECAP specifications.

### 11.2.3 STRUCTURAL SPECIFICATION

A structural SEATY specification may be of the forms $\vartheta : :$**announce**$(exp)$ or $\vartheta : :(P,\ E)$ if any variable occurring free in $exp$, $P$, or $E$ is an element of $\vartheta$.

Further, if $\vartheta : :S_1$ and $\vartheta : :S_2$ are two structural SEATY specifications and $b$ is boolean formula whose free variables are all in $\vartheta$, then, $\vartheta ::$**await** $b$ **do** $S_1$ **od**, $\vartheta ::\{S_1 \| S_2\}$, $\vartheta ::S_1; S_2$, and $\vartheta : :$**if** $b$ **then** $S_1$ **else** $S_2$ **fi** are also structural SEATY specifications.

Structural SEATY specifications are also given semantics by converting them to LECAP specifications.

## 11.3 CONSTRUCTION OF COMPONENTS

We provide a set of rules for the top-down development of SEATY programs. The soundness of these rules can be justified by making rely-/ and guar-conditions explicit. The rules for the construction of SEATY programs are interesting because they are based on the well-known pre- and post-condition style of specifying software systems.

### 11.3.1 Consequence Rule

The consequence rule allows strengthening the assumptions while weakening the commitments.

$$\frac{\begin{array}{l} E_1 \Rightarrow E_2, \\ P_2 \Rightarrow P_1, \\ z \ \underline{sat}\ \vartheta ::(P_1,\ E_1) \end{array}}{z \ \underline{sat}\ \vartheta ::(P_2,\ E_2).}$$

### 11.3.2 Conditional Rule

#### Basic Conditional Rule

The requirement made in LECAP specifications that the environment is not allowed to interfere with variables used in the boolean test becomes obsolete since it is a general requirement in the context of the SEATY type.

$$\frac{\begin{array}{l} z_1 \ \underline{sat}\ \vartheta ::(P \wedge b,\ E) \\ z_2 \ \underline{sat}\ \vartheta ::(P \wedge \neg b,\ E) \end{array}}{\textbf{if } b \textbf{ then } z_1 \textbf{ else } z_2 \textbf{ fi } \underline{sat}\ \vartheta ::(P,\ E).} \tag{11.1}$$

#### Composed Conditional Rule

$$\frac{z \ \underline{sat}\ \vartheta ::\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}; S_3}{z \ \underline{sat}\ \vartheta ::\textbf{if } b \textbf{ then } S_1; S_3 \textbf{ else } S_2; S_3 \textbf{ fi}.} \tag{11.2}$$

### 11.3.3 Parallel Rule

Two SEATY programs are composed in parallel. The traditional requirement for non-interference is expressed in terms of the set of variables that each of these programs accesses.

### 11.3.4 BASIC PARALLEL RULE

$$\frac{\begin{array}{l}\vartheta_{11} \cap \vartheta_{22} = \{\} \\ z_1 \ \underline{sat}\ \vartheta_1 : : (P_1,\ E_1) \\ z_2 \ \underline{sat}\ \vartheta_2 : : (P_2,\ E_2)\end{array}}{\{z_1 \| z_2\}\ \underline{sat}\ \vartheta_1 \cup \vartheta_2 : : (P_1 \wedge P_2,\ E_1 \wedge E_2).} \tag{11.3}$$

The general requirement in SEATY is that two programs running in parallel be such that none of them modifies a variable that the other has access to. This requirement may be too strong in situations were concurrency is needed. We provide the following rule that allows paralleling two programs even if they share some variables. Each of the programs is executed atomically. The parallel execution of $z_1$ and $z_2$ will, therefore, be either $z_1; z_2$ or $z_2; z_1$ (see [9, 115, 8, 116]).

$$\frac{\begin{array}{l}P_1,\ A \textbf{ stable when } E_2 \wedge B \\ P_2,\ B \textbf{ stable when } E_1 \wedge A \\ z_1 \ \underline{sat}\ \vartheta : : \textbf{await true do } (P_1,\ E_1 \wedge A) \textbf{ od} \\ z_2 \ \underline{sat}\ \vartheta : : \textbf{await true do } (P_2,\ E_2 \wedge B) \textbf{ od}\end{array}}{\{z_1 \| z_2\}\ \underline{sat}\ \vartheta : : (P_1 \wedge P_2,\ A \wedge B).} \tag{11.4}$$

COMPOSED PARALLEL RULE

$$\frac{\begin{array}{l}P_1,\ A \textbf{ stable when } E_2 \wedge B \\ P_2,\ B \textbf{ stable when } E_1 \wedge A \\ z \ \underline{sat}\ \vartheta : : (P_1,\ E_1 \wedge A) \| (P_2,\ E_2 \wedge B)\end{array}}{z \ \underline{sat}\ \vartheta : : (P_1 \wedge P_2,\ A \wedge B).} \tag{11.5}$$

If $z$ can be written as two programs $z_1$ and $z_2$ that coexist with each other, then the result of $z$ results from the application of the basic parallel to $z_1$ and $z_2$.

### 11.3.5 ITERATION RULE

In this rule, the pre-condition $P$ is an invariant of the loop; each iteration must ensure that its post-condition implies the pre-condition of the next iteration. The final state is

such that either the body of the loop was executed a finite number of time or it was not executed at all. In the first case the state satisfies $E^+$. In the second case, the identity assertion $I_\vartheta$ holds.

$$\frac{z \ \underline{sat} \ \vartheta::(P \wedge b, \ E \wedge P)}{\textbf{while } b \textbf{ do } z \textbf{ od} \ \ \underline{sat} \ \vartheta::(P, \ (E^+ \vee I_\vartheta) \wedge \neg b).} \tag{11.6}$$

### 11.3.6 Sequential Rule

The rule permits the sequential composition of programs.

#### Basic Sequential Rule

The composition is similar to the traditional sequential rule due to the missing announcement of events.

$$\frac{\begin{array}{l} z_1 \ \ \underline{sat} \ \ \vartheta::(P_1, \ E_1 \wedge P_2) \\ z_2 \ \ \underline{sat} \ \ \vartheta::(P_2, \ E_2) \end{array}}{z_1; z_2 \ \underline{sat} \ \vartheta::(P_1, \ E_1 \mid E_2).} \tag{11.7}$$

#### Composed Sequential Rule

$$\frac{z \ \ \underline{sat} \ \ \vartheta::(P_1, \ E_1 \wedge P_2); (P_2, \ E_2)}{z \ \underline{sat} \ \vartheta::(P_1, \ E_1 \mid E_2).} \tag{11.8}$$

If $z$ can be written as the sequential composition of two programs satisfying $\vartheta::(P_1, E_1 \wedge P_2)$ and $\vartheta::(P_2, E_2)$ respectively, then the conclusion of the rule is obtained by application of the basic sequential rule.

### 11.3.7 Skip Rule

The **skip** statement does nothing but terminates. The set of variables that the program is allowed to change is kept unchanged.

$$\textbf{skip}\ \underline{sat}\ \vartheta ::(P,\ I_\vartheta). \tag{11.9}$$

### 11.3.8 Assignment rule

A single program transition is performed, namely the assignment of the value of $r$ to the variable $v$ while all other variables are kept unchanged.

$$\frac{\overleftarrow{P} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \{v\}} \quad \Rightarrow \quad E}{v := r\ \underline{sat}\ \vartheta \cup \{v\} ::(P,\ E).} \tag{11.10}$$

### 11.3.9 Pre Rule

The pre-rule is straightforward; If a program terminates in a state satisfying $E$ when started in a state satisfying $P$, then, it does so relatively to the initial state (which satisfies the pre-condition $P$).

$$\frac{z\ \underline{sat}\ \vartheta ::(P,\ E)}{z\ \underline{sat}\ \vartheta ::(P,\ \overleftarrow{P} \wedge E).} \tag{11.11}$$

### 11.3.10 Global Rule

The global rule allows introduction of a new variable in the specification of a program. The program terminates in a state where the value of the new variable is the same as in the initial state.

$$\frac{z\ \underline{sat}\ (\vartheta \setminus \{v\},\ \mathcal{B}) ::(P,\ E)}{z\ \underline{sat}\ (\vartheta \cup \{v\},\ \mathcal{B}) ::(P,\ E \wedge I_{\{v\}}).} \tag{11.12}$$

## 11.4 System Behavior Analysis

After the specification of the components, the development of an event-based application recommends composing the specifications of this application starting with those of the components. This composition is done by means of the announce rule that we present below.

In addition to composing the specification of the application, proof obligations must be discharged that ensure that there is no interference in the system. Further, required global properties need to be checked. Such verifications are done on behavioral specifications that result from the announce rule.

The manipulation of behavioral specifications is done by means of some rules that resemble those for the decomposition of software systems. The consequence rule, the parallel rule, the pre rule, the assignment rule, the global rule, and the skip rule are obtained from those of the previous section by replacing $\vartheta$ with the event-based system $(\vartheta, \mathcal{B})$. For instance, the consequence rule for behavioral specifications is formulated as follows.

### 11.4.1 Consequence Rule

The consequence rule allows strengthening the assumptions while weakening the commitments.

$$\frac{\begin{array}{l} E_1 \Rightarrow E_2, \\ P_2 \Rightarrow P_1, \\ z \ \underline{sat}\ (\vartheta, \mathcal{B}) : : (P_1,\ E_1) \end{array}}{z \ \underline{sat}\ (\vartheta, \mathcal{B}) : : (P_2,\ E_2).}$$

The program $z$ may now announce an event. The behavior of the subscribers to these events is included in the behavior of $z$ captured by the behavioral specification $(\vartheta, \mathcal{B}) : : (P_1,\ E_1)$.

### 11.4.2 Composition of Applications: Announce Rule

Event-based applications are composed by subscribing programs to events. The resulting specifications are given by the announce rule which is based on the parallel rule and the semantics of the announce construct.

$$\frac{\begin{array}{l} subscribers(e) = \{z_1, \cdots z_n\} \\ \{z_1 \| \cdots \| z_n \| z\} \ \underline{sat}\ (\vartheta,\ \mathcal{B}) ::(P,\ E) \end{array}}{\textbf{announce}(e); z \ \underline{sat}\ \vartheta ::(P,\ E).} \tag{11.13}$$

The following rule is more general and allows announcing an event specified by means of an expression *exp*.

$$\frac{\begin{array}{l} events(z) = X_1 \uplus X_2 \\ \forall e \in X_i \cdot subscribers(e) = \{z_{1i}, \cdots, z_{ni}\} \\ \{z_{1i} \| \cdots \| z_{ni} \| z_{n+1}\} \ \underline{sat}\ (\vartheta, \mathcal{B}) ::S_i \\ z \ \underline{sat}\ \vartheta ::(P,\ E); \textbf{announce}(exp) \end{array}}{z; z_{n+1} \ \underline{sat}\ \ (\vartheta, \mathcal{B}) ::(P,\ E); \textbf{if } exp \in X_1 \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}.} \tag{11.14}$$

The set $events(z)$ determines the set of events that may be announced by the program $z$and is defined as

$$events(z) = \{e : Event \cdot \exists s_1, s_2 : State \cdot (s_1, s_2) \models E \wedge exp = e\}$$

The rule results in extended behavioral specifications on which other rules can be applied for the verification of global properties.

### 11.4.3 Conditional Rule

The conditional rule in the context of behavioral specifications is slightly different from that of structural specifications. The specification of the program **if** $b$ **then** $z_1$ **else** $z_2$ **fi**; $z$ is given instead of simply giving that of **if** $b$ **then** $z_1$ **else** $z_2$ **fi**. In this way, announcement of events is supported within if constructs.

### 11.4.4 Basic Conditional Rule

$$\frac{\begin{array}{l} z_1; z \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P \wedge b,\ E) \\ z_2; z \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P \wedge \neg b,\ E) \end{array}}{\textbf{if } b \textbf{ then } z_1 \textbf{ else } z_2 \textbf{ fi}; z \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P,\ E).} \tag{11.15}$$

### 11.4.5 Composed Conditional Rule

$$\frac{z \ \underline{sat} \ (\vartheta, \mathcal{B}) : : \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}; S_3}{z \ \underline{sat} \ (\vartheta, \mathcal{B}) : : \textbf{if } b \textbf{ then } S_1; S_3 \textbf{ else } S_2; S_3 \textbf{ fi}.} \tag{11.16}$$

The composed conditional rules is similar to the case of structural specifications.

### 11.4.6 Sequential Rule

The rule permits the sequential composition of programs. We consider two programs $z_1$ and $z_2$ such that $z_1$ announces no event, i.e. $events(z) = \{\}$. The sequential composition is possible if the pre-condition of the second program follows from the post-condition of the first.

$$\frac{\begin{array}{l} z_1 \ \ \underline{sat} \ \ \vartheta : : (P_1, \ E_1 \wedge P_2) \\ z_2 \ \ \underline{sat} \ \ (\vartheta, \ \mathcal{B}) : : (P_2, \ E_2) \end{array}}{z_1; z_2 \ \underline{sat} \ (\vartheta \ \mathcal{B}) : : (P_1, \ E_1 \mid E_2).} \tag{11.17}$$

### 11.4.7 Sequential-await Rule

The requirement that the first program announces no event can be removed if the first program is embedded in an await construct.

$$\frac{\begin{array}{l} z_2 \ \ \underline{sat} \ \ (\vartheta, \ \mathcal{B}) : : (P_2, \ E_2) \\ \textbf{await } b \textbf{ do } z_1 \textbf{ od } \underline{sat} \ \ (\vartheta, \ \mathcal{B}) : : (P_1, \ E_1 \wedge P_2) \end{array}}{\textbf{await } b \textbf{ do } z_1 \textbf{ od}; z_2 \ \underline{sat} \ (\vartheta, \ \mathcal{B}) : : (P_1, \ E_1 \mid E_2).} \tag{11.18}$$

## 11.5 A symbolic Example

We present a symbolic example that illustrates the development process of SeaTy applications. The example is similar to that of the previous chapter, except that the rely- and guar-conditions are replaced with the set of variables that a program accesses.

### 11.5.1 Specification of Components

The system includes two components $z_1$ and $z_2$ satisfying the following structural specifications.

$z_1$ *sat* $\vartheta_1 ::\{(P_{11},\ E_{11})\|(P_{12},\ E_{12})\}; \mathbf{announce}(e); (P_{13},\ E_{13})$.

$z_2$ *sat* $\vartheta_2 ::(P_{21},\ E_{21})$.

### 11.5.2 Verification of Local Properties

The local property of interest is the satisfaction of $Q$ in the final states of $z_1$. The empty binding is assumed for the verification of local properties.

By application of the announce rule followed by the skip rule, we deduce:

$z_1$ *sat* $(\vartheta_1, \mathcal{B}_0) ::\{(P_{11},\ E_{11})\|(P_{12},\ E_{12})\}; (P_{13},\ E_{13})$.

$z_2$ *sat* $(\vartheta_2, \mathcal{B}_0) ::(P_{21},\ E_{21})$.

The behavioral specification of $z_2$ only differs from its structural specification by the presence of the binding $\mathcal{B}_0$.

To determine the post-condition of the program $z_1$ we must discharge the following properties that are required by the parallel rule and the sequential rule respectively.

**Proof Obligation 18** $P_{11},\ E_{11}$ **stable when** $E_{12}$ *and* $P_{12},\ E_{12}$ **stable when** $E_{11}$

**Proof Obligation 19** $E_{11} \wedge E_{12} \Rightarrow P_{13}$

After discharging these proof obligations, the requirement of interest can be formulated as:

**Proof Obligation 20** $(E_{11} \wedge E_{12}) \mid E_{13} \Rightarrow Q$.

Assuming these requirements hold, we can now define a binding that reflects the architecture of our application.

### 11.5.3 Composition of the Application

The next step in the development process is to construct the behavioral specification of our application starting with the specifications of the components $z_1$ and $z_2$. For this, we

construct a binding by subscribing $z_2$ to the event $e$. The expected effect is the satisfaction of the assertion $Q$ in the final states of $z_1$. The resulting binding looks as follows:

$$\mathcal{B} \stackrel{def}{=} \{ \langle \mathbf{z}_1 \rangle \mapsto \{\}, \quad \langle \mathbf{z}_2 \rangle \mapsto \{e\}, \quad \langle \mathbf{skip} \rangle \mapsto \{x : Event\}\}.$$

Given this binding, we re-deduce the behavioral specification of $z_1$ which is the basis for the verification of the global properties.

### Derivation of the Behavioral Specification

By the announce rule followed by the skip rule, we have:

$z_1$ *sat* $(\vartheta_1, \mathcal{B}) :: \{(P_{11},\ E_{11}) \| (P_{12},\ E_{12})\}; \{(P_{21}, E_{21}) \| (P_{13},\ E_{13})\}$.

The requirement for a well-behaved applications are given by the parallel rule and by the sequential rule:

**Proof Obligation 21** $P_{21},\ E_{21}$ **stable when** $E_{13}$ *and* $P_{13},\ E_{13}$ **stable when** $E_{21}$

**Proof Obligation 22** $E_{11} \wedge E_{12} \Rightarrow P_{21}$

The proof obligation for the property of interest is therefore:

**Proof Obligation 23** $(E_{11} \wedge E_{12}) \mid (E_{21} \wedge E_{13}) \Rightarrow E_{21}$.

If this property holds, the components $z_1$ and $z_2$ can now be developed independently by stepwise refinement of their structural specifications.

## 11.6 Summary

Due to the concurrent execution of subscribers, event-based systems indeed share some properties with concurrent systems among which are the hardness in ensuring their reliability. This chapter introduced an architectural type called SeaTy (Sequential Event-Based Architectural Type). Applications in this family are such that any two distinct subscribers to the same event are constrained not to change the value of any variable that the other writes to or reads. By this constraint, the interference freedom is restricted to checking the set of variables that applications write to and read. The effort in constructing

SEATY applications is, therefore, expected to be less than for constructing pure LECAP applications.

The constraint on SEATY applications may be too strong for some applications. We, therefore, weakened this requirement by allowing programs executed concurrently to do so in atomic steps.

The chapter also presented a symbolic example intended to present the development of process of SEATY applications in a succinct manner.

# Chapter 12

# Method Invocation

## 12.1 Motivation

Methods (also called procedures or subroutines) are the basic and most accepted means for ensuring modularity in software systems. And, in fact, event-based systems rely on the concept of methods. The subscribers are not only invoked and executed in parallel, but the announced event is also passed to them. This chapter discusses the notion of value passing in method invocation. That is, we apply the notions of method declaration, method specification and method invocation to event-based systems.

The chapter is organized as follows. The next chapter presents an extension of the LECAP language that takes the notion of method into consideration. The extensions are both syntactic and semantic. In Section 12.3, we define the concept of method specifications such that it brings to specifications the kind of modularity that methods bring to programs. Section 12.4 discusses method proof rules; in particular, we investigate the specification of a method invocation. Section 12.5 concludes the chapter.

## 12.2 Extending the LECAP Language

We extend the syntax and the semantics of the LECAP programming language to include methods declaration and invocation.

### 12.2.1 Syntax

The syntax of the LECAP core programming language is now defined in the following manner. The program $p(e_1, \cdots, e_n)$ is added to denote the invocation of the method $p$ with the *actual parameters* $e_1, \cdots, e_n$ where $e_1, \cdots e_n$ are some expressions.

$$P ::= x := e \mid P_1; P_2 \mid \textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2 \textbf{ fi} \mid \textbf{while } b \textbf{ do } P \textbf{ od} \mid \{P_1 \| P_2\} \mid \textbf{announce}(x) \mid \textbf{skip} \mid \textbf{await } b \textbf{ do } P \textbf{ od} \mid p(e_1, \cdots, e_n).$$

$$\textbf{proc} \quad p(\textbf{val } v_1, \cdots v_n) \triangleq P \mid p() \triangleq P.$$

A method declaration **proc** is assumed where each method invoked in a program is declared. In the method declaration $p(\textbf{val } v_1, \cdots, v_n) \triangleq P$, the left hand side of the definition is called the *header* of the method $p$ and the right hand side is called the *body* of the method. The variables $v_i$ are called *formal parameters.*

We assume two functions **header** and **body** on method declarations defined in the following natural manner:

$$\textbf{header}(p(\textbf{val } v) \triangleq P) \stackrel{def}{=} p(\textbf{val } v)$$

$$\textbf{body}(p(\textbf{val } v) \triangleq P) \stackrel{def}{=} P.$$

The parameter passing mechanism supported is *call-by-value.* The keyword **val** is used for the declaration of these parameters. In a method invocation such as $p(e_1, \cdots, e_n)$, the value of any call-by-value actual parameter $e_i$ is computed and assigned to the formal parameter $v_i$ before execution of the body $P$ [8, 61, 62, 116]. Parameters are not mandatory in the definition of a method.

### 12.2.2 Restrictions

To simplify the deduction rules, it is required that:

1. each invoked method is declared once and only once;
2. for any method invocation $p(e_1, \cdots, e_n)$ of a method declared as $p(\textbf{val } v_1, \cdots, v_n) \triangleq P$, the types of any actual parameter $e_i$ must be the same as that of the corresponding formal parameter $v_i$;
3. we assume that methods are not recursive;
4. there is no assignment to formal parameters in the body of methods;
5. any declaration $p(\textbf{val } v_1, \cdots, v_n) \triangleq P$ is such that the $v_i$ are distinct identifiers;
6. any method $m$ in the domain of the binding (i.e. any subscriber) has a declaration of the form $p(\textbf{val } x : \mathit{Event}) \triangleq P$; i.e. each subscriber has one and only one parameter and this parameter is of the event type.

In the practice of event-based systems, there is no explicit requirement that the parameter of the subscribers be a call-by-value parameter. However, if we consider a programming language such as Java, most middleware (e.g. JEDI [31], Siena [23], PeerWare [101]) do accept a call-by-reference parameter, that they, internally clone such that copies are sent to subscribers; which indeed corresponds to our restriction.

**Definition 43** *Let us consider the method* $\mathbf{mskip}(x : Event) \triangleq \mathbf{skip}$ *that takes an event and does* **skip**. *In the remainder, we will assume that for any binding* $\mathcal{B}$ *and for any event* $e$, *the method* **mskip** *is subscribed to* $e$, *i.e.* $\mathbf{mskip} \in subscribers_{\mathcal{B}}(e)$. *The binding that has only the method* **mskip** *subscribed to any event is called empty binding and denoted* $\mathcal{B}_0$ *(this replaces Definition 5).*

This restriction on bindings allows us to present uniform rules without need to distinguish the cases of events with no subscribed method.

### 12.2.3 Semantics

#### Semantics of Event Announcement

The set of programs that subscribe to the event $e$ are invoked (triggered) and executed in parallel. The first case presents the case where the announcement is the last construct in the program. The announced event is passed to the subscribers.

$$\frac{s(x) = e, \quad subscribers(e) = \{z_1, \cdots, z_n\}}{\langle \mathbf{announce}(x),\ s\rangle \xrightarrow{i} \langle \{z_1(x) \| \cdots \| z_n(x)\},\ s\rangle}$$

If however, the announcement construct is not the last construct, the subscribers are executed in parallel with the remainder of the announcing program.

$$\frac{s(x) = e, m, n \geq 0, \quad subscribers(e) = \{z_1, \cdots, z_n\}}{\langle \{^n\mathbf{announce}(x); z\}^m,\ s\rangle \xrightarrow{i} \langle \{\{z_1(x) \| \cdots \| z_n(x)\} \| \{^n z\}^m\},\ s\rangle}$$

$$\frac{s(x) = e, \quad m, n \geq 1, \quad subscribers(e) = \{z_1, \cdots, z_n\}}{\langle \{^n\mathbf{announce}(x) \| z\}^m,\ s\rangle \xrightarrow{i} \langle \{\{z_1(x) \| \cdots \| z_n(x)\} \| \{^n z\}^m\},\ s\rangle}$$

In this formula, $\{^n$ denotes a sequence of $n$ left braces $\{$. Similarly $\}^m$ denotes a sequence of $m$ right braces. It is important that the number of braces be taken into consideration since omitting some of them results in a malformed program.

### Semantics of Methods Invocation

Call-by-value formal parameters are assigned the values of the corresponding actual parameters before the body $P$ of the method is executed.

$$\frac{p(\mathbf{val}\ v_1, \cdots, v_n) \triangleq P}{\langle p(e_1, \cdots, e_n),\ s\rangle \xrightarrow{i} \langle v_1 := e_1; \cdots; v_n := e_n; P,\ s\rangle}$$

It is also possible to write methods without parameters.

$$\frac{p() \triangleq P}{\langle p(),\ s\rangle \xrightarrow{i} \langle P,\ s\rangle}$$

## 12.3 Specification of Methods

This section defines the concept of specification of methods and investigates what it means for a method to satisfy a specification. We only consider SeaTy specifications. General Lecap specifications and resulting construction rules are still to be formulated.

### 12.3.1 Specification

Some arguments in this investigation are motivated by current practices in modelling software systems. For instance, considering a method declaration of the form $m(\mathbf{val}\ x) \triangleq M$, it is clear that the header $m(\mathbf{val}\ x)$ is not a program. It is, therefore, not well-formed to say that it satisfies a specification. Since this is widely done it practice, we legalize this statement by giving it a precise meaning. We introduce the concept of method specification.

**Definition 44** *A method specification is a formula of one of the forms*

$$
\begin{array}{l}
s() \triangleq \vartheta ::S \\
s() \triangleq (\vartheta, \mathcal{B}) ::S \\
s(\textbf{val } v_1, \cdots, v_n) \triangleq \vartheta ::S \\
s(\textbf{val } v_1, \cdots, v_n) \triangleq (\vartheta, \mathcal{B}) ::S
\end{array}
$$

*such that:*

- $\vartheta ::S$, *and* $(\vartheta, \mathcal{B}) ::S$ *are structural and behavioral program specifications respectively; and*
- *any variable that occurs free in* $S$ *is in* $\{v_1, \cdots, v_n\} \cup \vartheta$.

*The left hand side of a method specification is called the header of the specification while its right hand side is called its body.*

*A method specification is structural (respectively behavioral) iff its body is a structural (respectively behavioral) specification.*

Method specifications bring to specifications the kind of modularity that methods bring to programs. The functions **header** and **body** are naturally extended to method specifications. Method specification is for instance possible in VDM [75, 128]; an example is given below.

$$
\begin{array}{l}
push : Event \xrightarrow{o} () \\
push\,(evt) \\
\textsf{pre } \textsf{true} \\
\textsf{post } stack = evt.elt \curvearrowright \overleftarrow{stack}
\end{array}
$$

**Definition 45** *Two headers* $p$ *and* $s$ *are compatible iff they differ only by their names. That is, any variable in one header occurs in the other header at the same position and with the same type. We extend this definition and say that a method declaration is compatible to a method specification if their headers are compatible.*

Next, we introduce method judgments.

**Definition 46** *Let* $m(\textbf{val } v_1, \cdots, v_n) \triangleq M$ *and* $s(\textbf{val } v_1, \cdots, v_n) \triangleq \vartheta ::S$ *be a method declaration and a method specification respectively such that they are compatible. The formula* $m$ $\underline{sat}$ $s$ *is a method judgment iff any variable accessed by* $M$ *is in* $\vartheta \cup \{v_1, \cdots, v_n\}$.

This means that before talking about a method declaration satisfying a method specification, one must make sure that their headers are compatible. Next, we investigate what it means for a method to satisfy its specification.

**Definition 47** *Let* $m(\textbf{val}\ v_1, \cdots, v_n) \triangleq M$ *and* $s(\textbf{val}\ v_1, \cdots, v_n) \triangleq \vartheta ::S$ *be a method declaration and a method specification respectively such that they are compatible. The following statements are equivalent:*

- $m(\textbf{val}\ v_1, \cdots, v_n) \triangleq M \quad \underline{sat} \quad s(\textbf{val}\ v_1, \cdots, v_n) \triangleq \vartheta ::S$
- $m(\textbf{val}\ v_1, \cdots, v_n) \quad \underline{sat} \quad \vartheta ::s(\textbf{val}\ v_1, \cdots, v_n)$,
- $M \quad \underline{sat} \quad \{v_1, \cdots, v_n\} \cup \vartheta ::S$,
- $m \quad \underline{sat} \quad \vartheta ::s$.

Informally, writing that a method declaration satisfies a method specification must be interpreted as saying that their headers are compatible and the body of the method declaration satisfies the body of the method specification. Obviously, a method judgment may be structural or behavioral depending on whether the body of the method specfication is structural or behavioral.

### 12.3.2 Composition

To take advantage of the modularity that method specifications bring, it must be possible to compose these specifications. The composition operators for this purpose are the same as those presented for program specification.

Note that composing specifications is also possible in other specification techniques. In the example of (IFAD) VDM, one would formulate the following composition.

$$pushcount : Event \times \mathbb{N} \xrightarrow{o} ()$$

$$pushcount\,(e, count) \triangleq$$
push(e);
increment(count);

Although this looks more like a program, this is indeed a specification since *push* and *increment* are specifications.

**Definition 48** *Given a method specification* $s(\textbf{val}\ v_1, \cdots v_n) \triangleq \vartheta::S$ *(respectively* $s(\textbf{val}\ v_1, \cdots, v_n) \triangleq (\vartheta, \mathcal{B})::S$*), and a sequence of expressions* $e_i, \cdots, e_n$ *such that* $e_i$ *has the same type as* $v_i$*, then,* $\vartheta::s(e_1, \cdots, e_n)$ *( respectively* $(\vartheta, \mathcal{B})::s(e_1, \cdots, e_n)$*) is a structural specification (respectively behavioral specification) iff the set of variables occuring free in* $e_1, \cdots, e_n$ *are all in* $\vartheta$*.* $\vartheta::s(e_1, \cdots, e_n)$ *and* $(\vartheta, \mathcal{B})::s(e_1, \cdots, e_n)$ *are called an invocations of the specification* $s$*.*

Given that an invocation of a method specification is a program specification, it is clear that if $s_1$ and $s_2$ are two specification invocations then, $\vartheta::\textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}$, $\vartheta::\{s_1 \| s_2\}$, and $\vartheta::s_1; s_2$ are program specifications. In particular, it is possible to define a method specification based on some specification invocation. This is what happens in the above VDM example; $push(e)$ and $increment(count)$ are two specification invocations that are sequentially composed.

To be complete we need to say what it means for a program to satisfy a specification invocation!

**Definition 49** *Let* $s(\textbf{val}\ v_1, \cdots v_n) \triangleq \vartheta::S$ *be a method specification,* $z$ *be a program, and* $\vartheta::s(e_1, \cdots, e_n)$ *be an invocation of* $s$ *where the set of free variables in* $e_1, \cdots, e_n$ *are all in* $\vartheta$*.*

*The judgment* $z$ *sat* $\vartheta::s(e_1, \cdots, e_n)$ *is valid iff there exists a method declartion* $z_1(\textbf{val}\ v_1, \cdots v_n) \triangleq Z_1$ *such that:*

- $Z_1$ *sat* $\vartheta \cup \{v_1, \cdots, v_n\}::S$ *is a valid judgment and*
- $z$ *behaves as* $z_1(e_1, \cdots, e_n)$*;*

*The judgment* $z$ *sat* $(\vartheta, \mathcal{B})::s(e_1, \cdots, e_n)$ *is valid iff there exists a method declartion* $z_1(\textbf{val}\ v_1, \cdots v_n) \triangleq Z_1$ *such that:*

- $Z_1$ *sat* $(\vartheta \cup \{v_1, \cdots, v_n\}, \mathcal{B})::S$ *is a valid judgment and*
- $z$ *behaves as* $z_1(e_1, \cdots, e_n)$ *in* $(\vartheta, \mathcal{B})$*;*

Informally, we can distinguish two cases for a program satisfying a specification invocation:

In the first case, the invocation of a method with the same actual parameters as that of the specification invocation satisfies this specification invocation if the body of the invoked method satisfies that of the invoked specification.

In the second case, a program satisfies a specification invocation if it behaves as a method invocation that satisfies this specification invocation.

## 12.4 Method Invocation Rules

The invocation of a method is indeed a program whose specification we give in this section. The rule is a simplistic application of Gries's rules for procedure call [61, 62].

To facilitate the rule for the invocation of methods, we impose the following restrictions:

- Only names are accepted in the construction of actual arguments; in particular, references to array elements, and record fields are not accepted in actual arguments.
- The actual parameters should not be affected by assignments to global variables. That is, the variables occurring in the actual parameters should not be modified by the body of the method.

An invocation that satisfies these criteria is called a valid SEATY invocation.

If these assumptions hold, then, a specification of a method invocation $m(a_1, \cdots, a_n)$ is the formula $\vartheta::S[a_1/v_1, \cdots, a_n/v_n]$ obtained by simultaneously replacing the formal parameters $v_1, \cdots, v_n$ with the actual parameters $a_1, \cdots, a_n$ in the specification $\vartheta::S$ of the body of the method $m$.

$$\frac{\begin{array}{l} m \ \underline{sat}\ s \\ m(\mathbf{val}\ v_1, \cdots, v_n) \triangleq M \\ s(\mathbf{val}\ v_1, \cdots, v_n) \triangleq \vartheta::S \end{array}}{m(a_1, \cdots, a_n)\ \underline{sat}\ \vartheta::S[a_1/v_1, \cdots, a_n/v_n]} \tag{12.1}$$

Prior replacement of bound variables can be applied to avoid conflicts. The rule is not valid for LECAP in general, but for SEATY specifications.

It is important not to confuse the header $m(\mathbf{val}\ v_1, \cdots, v_n)$ and the program $m(e_1, \cdots, e_n)$. Saying that the first satisfies a specification means that the body of $m$ satisfies this specification while saying that the second satisfies a specification means that the program $v_1 := e_1; \cdots; v_n := e_n; M$ satisfies this specification. Both concepts coincide when the method is declared without parameters.

Gries's rules [61, 62] are more general in that they additionally take call-by-result, call-by-reference, and aliasing into consideration. These paradigms are not directly needed in this framework; ignoring them allows concentrating on those aspects that are specific to event announcements.

### 12.4.1 ANNOUNCE RULE

We now update the announcement rule to take method invocation into consideration.

$$\frac{\begin{array}{l} events(z) = X_1 \uplus X_2 \\ z \ \underline{sat}\ \vartheta ::(P,\ E); \mathbf{announce}(exp) \\ \forall e \in X_i \cdot subscribers(e) = \{z_{1i}, \cdots, z_{ni}\} \\ \{z_{1i}(exp) \| \cdots \| z_{ni}(exp) \| z_{n+1}\}\ \underline{sat}\ (\vartheta, \mathcal{B}) ::S_i \end{array}}{z; z_{n+1}\ \underline{sat}\ \ (\vartheta, \mathcal{B}) ::(P,\ E); \mathbf{if}\ exp \in X_1\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}.} \tag{12.2}$$

The execution of the announce construct results in the parallel execution of subscribers with the remainder of the announcing program $z$. The announced event is passed to the invoked method ma be defined by an expression *exp* whose value depends upon the state and input variables. In this case, the behavior of the announcing program also depends upon the input variable and upon the state variables $s_k$ in the point of announcement where *exp* is evaluated.

The set $events(z)$ determines the set of events that may be announced by the program $z$ and is defined as

$$events(z) = \{e : Event \cdot \exists s_1, s_2 : State \cdot (s_1, s_2) \models E \wedge exp = e\}$$

The rule results in a behavioral specification on which other rules can be applied to to allow the verification of global properties.

## 12.5 SUMMARY

Methods are not only the basic means for modularization of software systems, but are also essential in the event-based paradigm. This chapter discussed how to support method invocation with call-by-value parameters in the LECAP language. In addition, the chapter discussed the notion of method specification in the sequential event-based architectural type; an architectural type where any two subscribers are constrained not to share variables. Gries's proof-rule for method invocation was also adapted for the SEATY type.

# Chapter 13

# A Stock Quote Service for Mobile Users

Notification and User Awareness have been recognized as important values for collaborative environments and are now standard application domains for the event-based paradigm. Khronika [85], Interlocus [95], Awareness@work [114], Nessie [107], MOTION [80, 109], OPELIX [63, 64], the IBM stock service [14] are examples of applications that enable various levels of awareness based on the event-based paradigm. Typically, a user subscribes for some information and wants to be notified whenever some data are available in the system that match her subscription criteria. A multitude of applications and devices can be used by such end-users for receiving their messages: mobile phones, pagers, e-mail clients, fax, printers, etc.

On the other hand, event-based systems allow subscription of components but not of end-users. That is, an end-user can, for instance, not say to an event-based middleware: "this is my mobile phone number, please notify me whenever a new expert in formal software design logs in". A common problem is, therefore, to bridge the gap between user-level subscriptions and component-level subscriptions. In the light of a stock service, we show how to solve this problem at the abstract level where the behavioral and architectural properties of the application can be investigated as proposed by the LECAP methodology. This stock service also serves the purpose of illustrating the application of LECAP in a real life standard case-study. The model provided in this chapter is inspired both by the MOTION messaging system [80] and the IBM stock web service [14].

The remainder of the chapter is organized as follows. The next section gives an overview of the architecture of our stock service for mobile users (SMU). In Section 13.2, we provide a formal specification of the components of our stock service. Section 13.3 discusses the verification of local properties of these components while Section 13.4 shows how to compose a stock service starting with the proposed set of components. Section 13.5 investigates the verification of some global properties while Section 15.5 concludes the chapter.

## 13.1 Architectural Overview

The architecture of our stock quote service is presented by presenting a use case scenario first that justifies the need of each component. In the second step, the we informally discuss each of component.

### 13.1.1 Use Case Scenarios

The stock service we want to construct is intended to give users the possibility to subscribe to some stock information and be notified when this information is available. To support mobility, the system takes advantage of the various devices that users may have. When a user is in his office, it is indeed comfortable for her to receive her notifications as e-mails. When, however, she is on the road, receiving her notifications as GMS short messages (SMS) may be better. Five basic usage scenarios are targeted by the Smu system.

1. Define the kind of information one is interested in;
2. specify one's reachability criteria, (e.g. SMS, E-mail, Fax, etc.);
3. publish some information;
4. add/remove a notification medium;
5. add/remove an information producer.

While the requirement for the first three usage scenarios is obvious, the two last scenarios probably need some justification. The system is intended to be flexible such that possible notification mechanisms and information producers can be integrated or simply replaced in the future.

### 13.1.2 Architecture

Smu is intended to be a highly flexible system. It is, therefore, not surprising that the proposed architecture is based on the event-based paradigm. We distinguish four kinds of components: publishers (also called producers or brokers), communication components that embody notification media, user callbacks, and a subscription manager. The sequence diagram in Figure 13.1 shows how these components collaborate to achieve the intended functionalities.

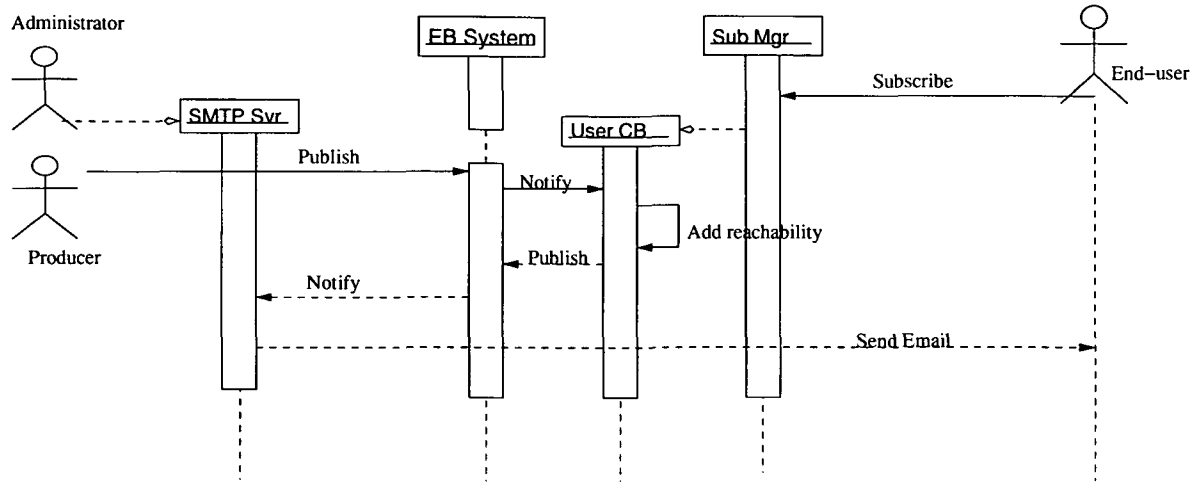


Figure 13.1: The SMU Sequential Diagram

The administrator is responsible for adding new communication components in the system (e.g. an SMTP Server as in the case of Figure 13.1). On the other hand, the end-user, is allowed to subscribe for information. The first time she subscribes, a user callback (called User CB in the figure) is subscribed for handling her subscriptions and notifications. Next, producers can publish information. If such an information matches the subscription of an end-user, her callback is invoked. The latter extends the event it received with the reachability criteria of the users and publishes it. Interested communication components are subsequently invoked by the middleware and they can deliver the message to the end-user.

A user callback is a component that handles subscriptions of a specific user. Whenever the user wishes to perform a subscription, she informs her callback. This callback mediates between the underlying event-based system and the user. It receives the user's subscriptions and subscribes on her behalf. Once an event occurs that satisfies one of the user's subscription criteria, the corresponding callback is informed.

Let us take an example. The end-user is called *Alice*, the producer is the *Wall Street Stock Info Service*. *Alice* wants to be informed by e-mail when the stock *DAX* reaches 100. The first time *Alice* subscribes, her user callback is created and subscribed on her behave (remember that event-based middleware do not know the concept of user subscription).

The *Wall Street Stock Info Service* publishes stock information whenever there is a change; and, it happens that on June 24th at 12 PM, the *DAX* reaches 100. The subscription of *Alice* is matched (in fact, the subscription of her callback). *Alice*'s callback is subsequently invoked. Note that the SMTP server could not be directly invoked since the event published by the *Wall Street Stock Info Service* does not contain the address of *Alice*. *Alice*'s callback which knows this address now adds it to the received event and publishes a message intended to the corresponding communication medium. This new publication matches

the subscription of the SMTP server, which is invoked by the event-based middleware. The e-mail is subsequently sent to Alice.

## 13.2 Components Specification

### 13.2.1 Data Modeling

We define the datatypes needed for the construction of our application.

$$
\begin{array}{lcl}
\mathit{Medium} & = & \langle \mathit{Email} \rangle \mid \langle \mathit{Pager} \rangle \mid \langle \mathit{SMS} \rangle \mid \langle \mathit{Fax} \rangle; \\
\mathit{Address} & = & \mathit{token}; \\
\mathit{StockName} & = & \mathit{token};
\end{array}
$$

We admit four communication media in the current system. The data types *Address* and *StockName* are declared as *token* because there is no need to give them a detailed structure at this level of abstraction.

$$
\begin{array}{lll}
\mathit{StockEvent} & :: & \\
 & \mathit{name} & : \mathit{StockName}, \\
 & \mathit{price} & : \mathbb{N};
\end{array}
$$

A *StockEvent* represents the kind of event that info brokers (or producers) can publish. It is obvious that this data type would contain more fields (e.g. date, broker's name) at the concrete level. We ignore them at this level.

$$
\begin{array}{lll}
\mathit{Message} & :: & \\
 & \mathit{medium} & : \mathit{Medium}, \\
 & \mathit{body} & : \mathit{StockEvent}, \\
 & \mathit{receiver} & : \mathit{Address};
\end{array}
$$

A message represents the kind of information that may be sent to end-users.

$$
\mathit{Event} = \mathit{StockEvent} \mid \mathit{Message};
$$

An event is either a stock event or a message. Messages are also events because they are a communication means between user callbacks and communication media.

$$
\begin{array}{lcl}
\mathit{Subscription} & = & \mathit{Event}\text{-set}; \\
\mathit{Callback} & = & \langle \mathit{impl\text{-}uscallback}_1 \rangle \mid \cdots \mid \langle \mathit{impl\text{-}uscallback}_n \rangle; \\
\mathit{Prog} & = & \mathit{Callback} \mid \langle \mathit{impl\text{-}smtpserver} \rangle \mid \langle \mathit{impl\text{-}smsserver} \rangle \mid \langle \mathit{impl\text{-}pagerserver} \rangle \mid \\
 & & \langle \mathit{impl\text{-}faxserver} \rangle \mid \langle \mathit{mskip} \rangle; \\
\mathit{Binding} & = & \mathit{Prog} \xrightarrow{m} \mathit{Subscription}\text{-set};
\end{array}
$$

A subscription is a set of events. We use the data type *Prog* to list methods defined in this specification and that are parts of the event-based system. We assume a finite number of users in our system such that each of them is associated to a user callback (uscallback). Next, a method is defined that represents each communication medium; the method that this communication medium uses for delivering messages.

$$
\begin{array}{lll}
\mathit{User} & :: & \\
 & \mathit{id} : \mathbb{N}, & \\
 & \mathit{callback} : \mathit{Callback}, & \\
 & \mathit{reachability} & : \mathit{Medium}, \\
 & \mathit{addresses} & : \mathit{Medium} \xrightarrow{m} \mathit{Address}, \\
 & \mathit{mailboxes} & : \mathit{Medium} \xrightarrow{m} \mathit{Message}*
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{inv} & \triangleq & \\
 & \mathit{inv}1 = & \forall \mathit{med} \in \mathsf{dom}\ \mathit{mailboxes},\ \mathit{mge} \in \mathsf{elems}\ \mathit{mailboxes}(\mathit{med}) \cdot \\
 & & \mathit{mge}.\mathit{medium} = \mathit{med} \wedge \mathit{mge}.\mathit{address} = \mathit{addresses}(\mathit{med}), \\
 & \mathit{inv}2 = & \mathit{reachability} \in \mathsf{dom}\ \mathit{addresses}, \\
 & \mathit{inv}3 = & \forall u1, u2 : \mathit{User} \cdot u1.\mathit{id} = u2.\mathit{id} \iff u1.\mathit{callback} = u2.\mathit{callback} \\
 & \mathit{inv}4 = & \forall u1, u2 : \mathit{User} \cdot u1.\mathit{id} \neq u2.\mathit{id} \Rightarrow \mathsf{rng}\ u1.\mathit{addresses} \cap \mathsf{rng}\ u2.\mathit{addresses} = \{\} \\
\mathsf{in} & & \mathit{inv}1 \wedge \mathit{inv}2 \wedge \mathit{inv}3 \wedge \mathit{inv}4;
\end{array}
$$

A user has a set of addresses; one address corresponding to each medium. The model can easily be extended to support more than one (or zero) addresses per medium. A mailbox is a sequence of messages; and, a user has a mailbox corresponding to each medium.

The first invariant requires that any message be put in the corresponding mailbox. That is, there should for instance be no e-mail in the SMS box. It also requires that the address of any received message corresponds to the user's address. The second invariant requires that a user must always be reachable; an address must be specified for the reachability medium. The third invariant requires that two users with different identifiers have two different callbacks. Finally, the fourth invariant requires that no two different users share the same address.

**System**

$$
\begin{array}{lll}
\mathit{users} & : & \mathbb{N} \xrightarrow{m} \mathit{user}, \\
v_1 & : & \mathit{StockEvent}, \\
v_2 & : & \mathit{Message}, \\
\mathcal{B} & : & \mathit{Binding} \\
\mathsf{inv} & \triangleq & \\
 & \mathit{inv}1 = & \forall\ \mathit{id} \in \mathsf{dom}\ \mathit{users} \cdot \mathit{users}(\mathit{id}).\mathit{id} = \mathit{id}, \\
 & \mathit{inv}2 = & \forall\ u \in \mathsf{rng}\ \mathit{users} \cdot \mathsf{inv}\text{-}\mathit{User}(u) \\
\mathsf{in} & & \mathit{inv}1 \wedge \mathit{inv}2;
\end{array}
$$

The state of our system consists of a sequence of users interested in being notified and two variables $v_1$ and $v_2$ used as actual argument for the announcement of events. For instance, instead of writing **announce**(*mk-StokEvent*(*name*, *price*)) we will write *mkStockEvent*(*name*, *price*); **announce**($v_2$). The method *mkStokEvent* is responsible in creating an appropriate event and storing it in the variable $v_2$. In this way, invocation of methods with record fields as arguments is avoided; this is one of the restrictions imposed in Chapter 12. The invariant ensures that each identifier *id* is mapped to a user with the same identifier. It is required that the number of users is equal to the number of user callbacks.

### 13.2.2 Specification of Components

We specified the methods (components) of our application. We distinguish information producers, user callbacks, a subscription manager, and communication media.

#### Information Producer

An information producer includes a single method *impl-broker* that satisfies *broker* for publishing stock information. The method constructs and announces an event with the given stock name and price. We omit the pre-conditions when they are true.

$$\textbf{broker}(name{:}StockName,\ price{:}\mathbb{N}) \triangleq\ mkStockEvent(name,\ price); \textbf{announce}(v_1);$$

$$\begin{array}{l}\textbf{mkStockEvent}(name : StockName,\ price : \mathbb{N}) \triangleq \\ \quad \textsf{wr} \quad v_1 \\ \quad \textsf{post} \quad v_1.name = name \wedge v_1.price = price\end{array}$$

#### User Callback

Each specification $uscallback_i$ corresponds to the method $impl\text{-}uscallback_i$. It constructs a message and subsequently announces it.

$$\begin{array}{l}\textbf{mkMessage}(sevt : StockEvent, i : \mathbb{N}) \triangleq \\ \quad \textsf{wr} \quad v_2, users \\ \quad \textsf{post} \quad v_2.medium = users(i).reachability \wedge v2.body = sevt \wedge \\ \qquad v_2.address = users(i).addresses(users(i).reachability) \wedge users = \overleftarrow{users}\end{array}$$

$$\textbf{uscallback}_1(sevt : StockEvent) \triangleq\ mkMessage(sevt, 1); \textbf{announce}(v_2);$$

$\vdots$

$$\textbf{uscallback}_\textbf{n}(sevt : StockEvent) \triangleq mkMessage(sevt, n); \textbf{announce}(v_2);$$

A number, $n$, of callbacks are defined for representing each user. Given an element $\langle impl\text{-}uscallback_i \rangle$ of type *Callback* the event-based system is responsible for invoking the corresponding callback. Note how user identifiers are used as constants in the definition of these callbacks.

An ideal way of modeling such dynamic systems would be to use an object-oriented language. In this context, we would define a class *user* that includes the fields specified in the composite structure *user* defined above. The method *impl-uscallback* would be an instance method such that each instantiation of the class *user* would have its own callback method. A subscriber would, hence, be of the form *u.impl-callback* where $u$ is an instance of *user*. Neither our logic nor the LECAP language allows using such concepts.

### COMMUNICATION MEDIA

The specification of the communication media is provided. The method *impl-smtpserver* that is supposed to satisfy *smtpserver* for instance is used by the SMTP server for notifying users.

$$\begin{array}{l}
\textbf{simple-smtpserver}(m : Message) \triangleq \\
\quad \textsf{wr} \quad users \\
\quad \textsf{pre} \quad m.medium = \langle Email \rangle \\
\quad \textsf{post} \quad (\forall id \in \textsf{dom}\ users \cdot m.address \notin \textsf{rng}\ users(id).addresses \Rightarrow users(id) = \overleftarrow{users}(id)) \wedge \\
\qquad (\forall u \in \textsf{rng}\ users \cdot u.addresses(\langle Email \rangle) = m.address \Rightarrow \\
\qquad \{m\} \cup \textsf{rng}\ \overleftarrow{users}(u.id).mailboxes(\langle Email \rangle) = users(u.id).mailboxes(\langle Email \rangle)) \wedge \\
\qquad (\forall m : Medium \cdot m \neq \langle Email \rangle \Rightarrow \overleftarrow{users}(u.id).mailboxes(m) = u.mailboxes(m))
\end{array}$$

$$\begin{array}{l}
\textbf{smtpserver}(mge : Message) \triangleq \\
\quad \textbf{await true do } simple\text{-}smtpserver(mge) \textbf{ od}
\end{array}$$

$$\begin{array}{l}
\textbf{simple-smsserver}(m : Message) \triangleq \\
\quad \textsf{wr} \quad users \\
\quad \textsf{pre} \quad m.medium = \langle SMS \rangle \\
\quad \textsf{post} \quad (\forall id \in \textsf{dom}\ \cdot m.address \notin \textsf{rng}\ users(id).addresses \Rightarrow users(id) = \overleftarrow{users}(id)) \wedge \\
\qquad (\forall u \in \textsf{rng}\ users \cdot u.addresses(\langle SMS \rangle) = m.address \Rightarrow \\
\qquad \{m\} \cup \textsf{rng}\ \overleftarrow{users}(u.id).mailboxes(\langle SMS \rangle) = users(u.id).mailboxes(\langle SMS \rangle)) \wedge \\
\qquad (\forall m : Medium \cdot m \neq \langle SMS \rangle \Rightarrow \overleftarrow{users}(u.id).mailboxes(m) = u.mailboxes(m))
\end{array}$$

**smsserver**($mge : Message$) $\triangleq$
**await true do** *simple-smsserver*($mge$) **od**

A method is defined that embodies the behavior of each communication medium. In the case of the SMTP server, this behavior consists of adding the received message in the Email mailbox of all the user whose address is specified in this message. This is an simplified view of the simple mail transfer protocol.

SUBSCRIPTION MANAGER

**subscribeUser**($user : User,\ subscription : Subscription$) $\triangleq$
wr $\mathcal{B},\ users$
post $user.id \in \mathsf{dom}\ users \implies \mathcal{B}(user.callback) = \overleftarrow{\mathcal{B}}(user.callback) \cup subscription \wedge$
$user.id \notin \mathsf{dom}\ users \implies \mathcal{B}(user.callback) = subscription \wedge$
$\forall m \in \mathsf{dom}\ \mathcal{B} \setminus \{user.callback\} \cdot \mathcal{B}(m) = \overleftarrow{\mathcal{B}}(m) \wedge\ users(user.id) = user$
$\forall id \in \mathsf{dom}\ users \setminus \{user.id\} \cdot users(id) = \overleftarrow{users}(id).$

The method *impl-subscribeUser* (satisfying *subscribeUser*) is a method of the subscription manager that allows the end-user to specify the kind of messages she is interested in. If the user to subscribe is already registered in the map *users* of all known users, then the set of events she is interested in is simply updated. If this is, however, not the case, then, the user is added to the map first.

Two methods implementing *subscribeUser* can not be executed concurrently. Since the formula **post**-*subscriberUser* **stable when post**-*subscriberUser* does not hold, the concurrent execution of two such methods is not guaranteed to terminate in a state satisfying **post**-*subscriberUser*. In fact, there is no urgent need for the concurrent execution of methods implementing *subscribeUser*. We can live with a "sequential" execution of *impl-subscribeUser* as this is not an activity that happens very frequently (like delivering of messages to users).

Next, we construct the specification *setReliability* of the method *impl-setRechability* that allows end-users to configure their reliability status.

**setReliability**($id : \mathbb{N},\ m : Medium$) $\triangleq$
wr $users$
pre $id \in \mathsf{dom}\ users \wedge m \in \mathsf{dom}\ users(id).addresses$
post $users(id) = \mu(users(id), reachability \mapsto m) \wedge$
$\forall uid \in \mathsf{dom}\ users \setminus \{id\} \cdot users(uid) = \overleftarrow{users}(uid)$

A reachability criteria can be set only if an address is defined for this medium. In this case, the corresponding user is updated while all other users are kept unchanged.

## 13.3 Verification of Local Properties

### 13.3.1 Property I

The first local property we investigate is whether subscribing a user results in a system whose invariant holds. In the tradition of LECAP, to discharge some local properties, one must transform the structural specifications into behavioral specifications first. In the case of *subscribeUser*, there is nothing that needs to be done; the structural specification coincide with the behavioral specification in any binding. We can, therefore, directly formulate our proof obligation.

**Proof Obligation 24** $\forall u : User,\ s : Subscription \cdot \textbf{post-}subscribeUser(u, s) \Rightarrow inv\text{-}System$

It is not difficult to see that this invariant does not hold. The invariant of the user to subscribe and that of the system must hold before invoking this method. We, therefore, strengthen the pre-condition of *subscribeUser* and subsequently apply the pre-rule to add more information in the post-condition, resulting in the following specification.

$$
\begin{array}{l}
\textbf{subscribeUser}(user : User,\ subscription : Subscription) \triangleq \\
\quad \textsf{pre} \quad \textsf{inv-}User(user) \wedge \textsf{inv-}System \\
\quad \textsf{post} \quad user.id \in \textsf{dom}\ users \implies \mathcal{B}(user.callback) = \overleftarrow{\mathcal{B}}(user.callback) \cup subscription \wedge \\
\qquad\qquad user.id \notin \textsf{dom}\ users \implies \mathcal{B}(user.callback) = subscription \wedge \\
\qquad\qquad \forall m \in \textsf{dom}\ \mathcal{B} \setminus \{user.callback\} \cdot \mathcal{B}(m) = \overleftarrow{\mathcal{B}}(m) \wedge\ users[user.id] = user \\
\qquad\qquad \forall id \in \textsf{dom}\ users \setminus \{user.id\} \cdot users(id) = \overleftarrow{users}(id) \wedge \\
\qquad\qquad \overleftarrow{\textsf{inv-}User}(user) \wedge \overleftarrow{\textsf{inv-}System}
\end{array}
$$

To show that the invariant of the system holds after the execution of a method which satisfies this specification, one must show that:

- the invariant of any user in the range of *users* is valid,
- any identifier in the domain of the *users* is mapped to an element with the same identifier.

The proof is simple. Let us take a user $u$ that is to be subscribed to some information using the subscription $s$. The map *users* is transformed such that $users(u.id) = u$ and any other entry is kept unchanged. That is, the identifier $u.id$ is indeed mapped to an element with the same identifier and such that its invariant holds. On the other hand, since the invariant of the system was satisfied in the initial state and any other entry is kept unchanged, the invariant of the system follows.

### 13.3.2 Property II

Next, we want to prove that users will always be reachable. In other terms, an address must be specified for the reachability medium of any user. The method *impl-setReachability* is used for modifying the reachability of a user. We show that it terminates in a state where this property holds.

The proof obligation is formulated as follows:

**Proof Obligation 25** $\forall i : \mathbb{N}, m : Medium \cdot \overleftarrow{\textbf{pre-}setReliabilty} \land \textbf{post-}setReliabilty \Rightarrow users(i).reachability \in \textsf{dom}\ users(i).addresses$

To discharge this proof obligation, we must also require that the invariant of the system holds in the initial state.

$$
\begin{array}{l}
\textbf{setReliability}(id : \mathbb{N},\ m : Medium) \triangleq \\
\quad \begin{array}{ll}
\textsf{wr} & users \\
\textsf{pre} & id \in \textsf{dom}\ users \land m \in \textsf{dom}\ users(id).addresses \land \textsf{inv-}System \\
\textsf{post} & users(id) = \mu(users(id), reachability \mapsto m) \land \\
 & \forall u \in \textsf{dom}\ users \setminus \{id\} \cdot users(id) = \overleftarrow{users}(id)
\end{array}
\end{array}
$$

The proof is straightforward; one observes that the pre-condition requires that the given medium be already mapped to some address before invoking the method. The result follows from the application of the definition of $\mu$ and the fact that all other entries in *users* are kept unchanged from a state where inv-*System* held.

## 13.4 Application Composition

This step consists of subscribing components to events. We construct a specification of the Smu application starting with the specification of the components that we presented in the previous sections.

In particular, we attach some communication media to the system, show how users can subscribe to stock information, show how to set reachability criteria, and verify some global properties of the application.

### 13.4.1 Attaching Brokers

A broker is an information source capable of publishing stock information in the system. Brokers interact with the Smu system by invoking the operation *impl-brocker*. The internal

structure of such brokers is not of interest to us. In particular, they can come and leave as they will.

In practice, however, it is necessary to be able to identify brokers that are allowed to publish information such that authorization can be checked and accuracy of information guaranteed.

### 13.4.2 Attaching Communication Media

We show how a communication medium can be attached to the system. We consider the case of an SMTP server as shown in Figure 13.1. Starting with an empty binding, we apply the integration rule to insert the method *impl-smtpserver* in the binding. Next, we subscribe this method to events of type *Message* where *medium* has the value $\langle Email \rangle$. The resulting effect should be the invocation of the SMTP server whenever a message is published in the system with *medium* set to $\langle Email \rangle$.

The binding subsequently looks as follows:

$$\begin{array}{lll} \mathcal{B}_1 \triangleq \{ & \langle \textbf{impl-smtpserver} \rangle & \mapsto \{e : Message \cdot e.medium = \langle Email \rangle\}, \\ & \langle \textbf{mskip} \rangle & \mapsto \{x : Event\}\} \end{array}$$

Other communication media can be inserted in the same way by subscribing them to the corresponding type of message. Note that there is nothing that prevents the administrator from subscribing the SMTP server to messages intended to the SMS server (i.e. with $m.medium = \langle SMS \rangle$). The pre-condition of this method will, however, not be satisfied and this will result in the violation of the first invariant of the data type *user* which requires that messages of type *Email* be indeed inserted only in the corresponding mailbox. An adequate integration of an SMS server yields the following binding:

$$\begin{array}{lll} \mathcal{B}_1 \triangleq \{ & \langle \textbf{impl-smtpserver} \rangle & \mapsto \{e : Message \cdot e.medium = \langle Email \rangle\}, \\ & \langle \textbf{impl-smsserver} \rangle & \mapsto \{e : Message \cdot e.medium = \langle SMS \rangle\}, \\ & \langle \textbf{mskip} \rangle & \mapsto \{x : Event\}\} \end{array}$$

### 13.4.3 Subscribing End-users

Since we provide the primitive *impl-broker* for publishing information, we can assume that some broker exists. We have also already integrated some communication media in the system. We proceed to subscribing end-users such that they can receive stock information.

By the integration rule, we insert a callback called $impl\text{-}uscallback_\alpha$ into the binding of the event-based system. The binding now looks as follows:

$$\mathcal{B}_2 \triangleq \{ \begin{array}{ll} \langle \textbf{impl-smtpserver} \rangle & \mapsto \{e : \mathit{Message} \cdot e.\mathit{medium} = \langle \mathit{Email} \rangle\}, \\ \langle \textbf{impl-smsserver} \rangle & \mapsto \{e : \mathit{Message} \cdot e.\mathit{medium} = \langle \mathit{SMS} \rangle\}, \\ \langle \textbf{impl-uscallback}_\alpha \rangle & \mapsto \{\}, \\ \langle \textbf{mskip} \rangle & \mapsto \{x : \mathit{Event}\} \end{array}$$

The second invariant of the user type requires that any user in the database is always reachable. Inserting the callbacks into the event-based system does not violate this requirement as no user is inserted to, removed from, or updated in the map of all known users.

Given this binding, we can subscribe the user with identifier $\alpha$ to some stock information. In particular, we subscribe her to any stock information where price is in the interval $[\mathit{min}_\alpha, \mathit{max}_\alpha]$. The corresponding subscription is:

$$\mathit{subscription} = \{e : \mathit{StockEvent} \cdot e.\mathit{price} \leq \mathit{max}_\alpha \wedge e.\mathit{price} \geq \mathit{min}_\alpha\}.$$

We also need to construct the user $u$ that we want to subscribe to these events. For this, we assume the existence of a value $\langle \mathit{Address}_\alpha \rangle$ of type *Address*. We choose $u$ to be such that

$$\begin{array}{l} u.\mathit{id} = \alpha \\ u.\mathit{callback} = \langle \mathit{uscallback}_\alpha \rangle \\ u.\mathit{reachability} = \langle \mathit{Email} \rangle \\ u.\mathit{addresses} = \{\langle \mathit{Email} \rangle \mapsto \langle \mathit{Address}_\alpha \rangle\} \\ u.\mathit{mailboxes} = \{\langle \mathit{Email} \rangle \mapsto \{\}, \langle \mathit{SMS} \rangle \mapsto \{\}\} \end{array}$$

It can easily be shown that the invariant of this user holds and we can, therefore, apply *impl-subscribeUser* resulting in the following binding

$$\mathcal{B}_2 \triangleq \{ \begin{array}{ll} \langle \textbf{impl-smtpserver} \rangle & \mapsto \{e : \mathit{Message} \cdot e.\mathit{medium} = \langle \mathit{Email} \rangle\}, \\ \langle \textbf{impl-smsserver} \rangle & \mapsto \{e : \mathit{Message} \cdot e.\mathit{medium} = \langle \mathit{SMS} \rangle\}, \\ \langle \textbf{impl-uscallback}_\alpha \rangle & \mapsto \{e : \mathit{StockEvent} \cdot e.\mathit{price} \in [\mathit{min}_\alpha, \mathit{max}_\alpha]\}, \\ \langle \textbf{mskip} \rangle & \mapsto \{x : \mathit{Event}\} \end{array}$$

and the following users configuration: $\mathit{users} = [\alpha \mapsto u]$.

### 13.4.4 Scenario Execution

To illustrate the behavior of the application, let us investigate two typical scenarios. In the first scenario, a stock event is published with some price $\theta < \mathit{min}_\alpha$. In the second scenario, the price $\theta$ is such that $\theta \in [\mathit{min}_\alpha, \mathit{max}_\beta]$.

SCENARIO I

1. The method *impl-broker* is invoked with some stock name and the natural $\theta < min_\alpha$.
2. A stock event is constructed by this method and announced.
3. The matching is performed by the event-based infrastructure which, however, notices based on the binding that nobody is interested in this event. The scenario terminates.

SCENARIO II

1. The method *impl-broker* is invoked with some stock name and the natural $\theta \in [min_\alpha, max_\beta]$.
2. A stock event *se* is constructed by this method and announced.
3. The matching is performed by the event-based infrastructure which notices based on the binding that $impl\text{-}uscallback_\alpha$ is interested in this event.
4. The method $impl\text{-}uscallback_\alpha$ is invoked by the event-based infrastructure with the corresponding event *se*.
5. The method $impl\text{-}uscallback_\alpha$ reads the reachability medium *med* and the address *adr* of the user at the position $\alpha$ of the map *users*. A message $m$ is constructed such that $m.medium = med$, $m.address = adr$, and $m.body = se$.
6. The message $m$ is announced with $m.medium = med$, $m.address = adr$, and $m.body = se$.
7. The event-based infrastructure detects based on the binding that *impl-smtpserver* is interested in events with medium equal to $\langle Email \rangle$ (since $med = \langle Email \rangle$).
8. The method *impl-smtpserver* is invoked with the argument $m$.
9. The message $m$ is inserted in the SMTP mailbox of the user with address *adr*.

### 13.4.5 IDENTIFICATION OF AFFECTED COMPONENTS

We identify the components whose behaviors may be affected by the subscription of end-users.

The first program affected by these subscriptions is *impl-broker*: a user subscription is followed by the subscription of a method of the form $impl\text{-}uscallback_i$ which is subscribed to some stock events announced by *impl-broker*. The execution of the latter, therefore, results in the execution of the first.

### 13.4.6 Derivation of Behavioral specifications

We proceed with the derivation of the behavioral specifications of the methods $impl\text{-}uscallback_i$ and *impl-broker*. We assume the binding $\mathcal{B}_3$ and discuss the definition of the function *subscriber*.

The value of this function, in fact depends on the values of $min_\alpha$, and $max_\alpha$.

- $\forall e \in \{e : StockEvent \cdot e.price \in [min_\alpha, max_\alpha]\} \cdot subscribers(e) = \{impl\text{-}uscallback_\alpha, mskip\}$
- $\forall e \in \{e : StockEvent \cdot e.price \notin [min_\alpha, max_\alpha]\} \cdot subscribers(e) = \{mskip\}$

On the other hand, for events of type *Message*, the function *subscribers* is defined as follows.

- $\forall e \in \{e : Message \cdot e.medium = \langle Email \rangle\} \cdot subscribers(e) = \{impl\text{-}smtpserver, mskip\}$
- $\forall e \in \{e : Message \cdot e.medium = \langle SMS \rangle\} \cdot subscribers(e) = \{impl\text{-}smsserver, mskip\}$
- $\forall e \in \{e : Message \cdot e.medium \notin \{\langle Email \rangle, \langle SMS \rangle\}\} \cdot subscribers(e) = \{mskip\}$

By the announce rule, one derives the following behavioral specifications of $impl\text{-}uscallback_\alpha$ and *impl-broker* respectively.

$behavioral\text{-}uscallback_\alpha(sevt : StockEvent) \triangleq$
  $mkMessage(sevt,\ \alpha)$;
  **if** $v_2.medium = \langle Email \rangle$ **then** $smtpserver(v_2) \| mskip(v_2)$
  **else if** $v_2.medium = \langle SMS \rangle$ **then** $smsserver(v_2) \| mskip(v_2)$
  **else** $mskip(v_2)$ **fi fi**

$behavioral\text{-}broker(name : StockName, price : \mathbb{N}) \triangleq$
  $mkStockEvent(name,\ price)$;
  **if** $v_1.price \in [min_\alpha, max_\alpha]$ **then** $uscallback_\alpha(v_1) \| mskip(v_1)$
  **else** $mskip(v_1)$ **fi fi**

By the application of the skip rule we simplify these specifications into the following formulas.

$behavioral\text{-}uscallback_\alpha(sevt : StockEvent) \triangleq$
  $mkMessage(sevt,\ \alpha)$;
  **if** $v_2.medium = \langle Email \rangle$ **then** $smtpserver(v_2)$
  **else if** $v_2.medium = \langle SMS \rangle$ **then** $smsserver(v_2)$
  **else** $mskip(v_2)$ **fi fi**

$behavioral\text{-}broker(name : StockName, price : \mathbb{N}) \triangleq$
  $mkStockEvent(name,\ price)$;
  **if** $v_1.price \in [min_\alpha, max_\alpha]$ **then** $uscallback_\alpha(v_1)$
  **else** $mskip(v_1)$ **fi fi**

The proof obligation to be discharged for a well-defined behavior of *impl-broker* and $impl\text{-}uscallback_\alpha$ are respectively:

**Proof Obligation 26**
$\forall s : StockEvent, i : \mathbb{N} \cdot \textbf{post-}mkMessage(s, i) \wedge v_2.medium = \langle Email \rangle \Rightarrow \textbf{pre-}smtpserver(v_2)$

**Proof Obligation 27**
$\forall s : StockEvent, i : \mathbb{N} \cdot \textbf{post-}mkMessage(s, i) \wedge v_2.medium = \langle SMS \rangle \Rightarrow \textbf{pre-}smsserver(v_2)$

**Proof Obligation 28**
$\forall n : StockName, p : \mathbb{N} \cdot \textbf{post-}mkStockEvent(n, p) \wedge v_1.price \in [min_\alpha, max_\alpha] \Rightarrow \textbf{pre-}uscallback_\alpha(v_1)$

These proof obligations are trivially discharged by replacing the various assertion names with their definitions: $\textbf{pre-}smtpserver(v_2)$ with $v_2.medium = \langle Email \rangle$, $\textbf{pre-}smsserver(v_2)$ with $v_2.medium = \langle SMS \rangle$, and $\textbf{pre-}uscallback_\alpha$ with true.

## 13.5 Verification of Global Properties

We have completed the specification of the application such that interference freedom is ensured. We proceed to checking some global properties, namely that:

- users indeed receive the messages of interest,
- messages are sent to users using the desired communication medium.

### 13.5.1 Property I

A user receives a message iff this message is stored in one of her mailboxes. The messages in which a user is interested is the set of messages to which her callback is subscribed to. The requirement is formulated as:

**Proof Obligation 29** $\forall n : StockName, p \in [min_\alpha, max_\alpha] \cdot \textbf{post-}broker(n, p) \Rightarrow$
$\exists m \in users(\alpha).mailboxes(users(\alpha).reachability) \cdot m.body.name = n \wedge m.body.price = p.$

The proof is by natural deduction. In the first case, it is assumed that the user is reachable through email.

**from** $post\text{-}broker(n,p) \wedge p \in [min_\alpha, max_\alpha] \wedge \overleftarrow{users}(\alpha).reachability = \langle Email \rangle$

**from** $p \in [min_\alpha, max_\alpha] \wedge post\text{-}mkStockEvent(n,p)$
**infer** $v_1.price \in [min_\alpha, max_\alpha]$

$post\text{-}mkStockEvent(n,p) \mid post\text{-}uscallback_\alpha(v_1)$

**from** $post\text{-}uscallback_\alpha(sevt) \wedge \overleftarrow{users}(\alpha).reachability = \langle Email \rangle$
**infer** $mkMessage(sevt, \alpha) \mid post\text{-}smtpserver(v_2)$

**from** $post\text{-}mkStockEvent(n,p) \wedge p \in [min_\alpha, max_\alpha]$
**infer** $v_1.name = n \wedge v_1.price = p \wedge v_1.price \in [min_\alpha, max_\alpha]$

$v_1.name = n \wedge v_1.price = p \mid post\text{-}uscallback_\alpha(v_1)$
$v_1.name = n \wedge v_1.price = p \wedge post\text{-}uscallback_\alpha(v_1)$

**from** $post\text{-}mkMessage(v_1, \alpha) \wedge users(\alpha).reachability = \langle Email \rangle$
**infer** $v_2.body.name = n \wedge v_2.body.price = p \wedge v_2.reachability = \langle Email \rangle \wedge$
$v_2.address = users(\alpha).adresses(\langle Email \rangle)$

**from** $post\text{-}smtpserver(v_2) \wedge v_2.body.name = n \wedge v_2.price = p \wedge$
$users(\alpha).addresses(\langle Email \rangle) = v_2.address$
**infer** $v_2 \in users(\alpha).maiboxes(\langle Email \rangle)$

**infer** $\exists m \in users(\alpha).maiboxes(\langle Email \rangle) \cdot m.body.name = n \wedge m.body.price = p$

In the second case, the user is reachable through the SMS communication medium.

**from** $post\text{-}broker(n,p) \wedge p \in [min_\alpha, max_\alpha] \wedge \overleftarrow{users}(\alpha).reachability = \langle SMS \rangle$

**from** $p \in [min_\alpha, max_\alpha] \wedge post\text{-}mkStockEvent(n,p)$
**infer** $v_1.price \in [min_\alpha, max_\alpha]$

$post\text{-}mkStockEvent(n,p) \mid post\text{-}uscallback_\alpha(v_1)$

**from** $post\text{-}uscallback_\alpha(sevt) \wedge \overleftarrow{users}(\alpha).reachability = \langle SMS \rangle$
**infer** $mkMessage(sevt, \alpha) \mid post\text{-}smsserver(v_2)$

**from** $post\text{-}mkStockEvent(n,p) \wedge p \in [min_\alpha, max_\alpha]$
**infer** $v_1.name = n \wedge v_1.price = p \wedge v_1.price \in [min_\alpha, max_\alpha]$

$v_1.name = n \wedge v_1.price = p \mid post\text{-}uscallback_\alpha(v_1)$
$v_1.name = n \wedge v_1.price = p \wedge post\text{-}uscallback_\alpha(v_1)$

**from** $post\text{-}mkMessage(v_1, \alpha) \wedge users(\alpha).reachability = \langle SMS \rangle$
**infer** $v_2.body.name = n \wedge v_2.body.price = p \wedge v_2.reachability = \langle SMS \rangle \wedge$
$v_2.address = users(\alpha).adresses(\langle SMS \rangle)$

$$\begin{array}{ll} \textbf{from} & post\text{-}smsserver(v_2) \wedge v_2.body.name = n \wedge v_2.price = p \wedge \\ & users(\alpha).addresses(\langle SMS \rangle) = v_2.address \\ \textbf{infer} & v_2 \in users(\alpha).maiboxes(\langle SMS \rangle) \end{array}$$

$$\textbf{infer} \quad \exists m \in users(\alpha).maiboxes(\langle SMS \rangle) \cdot m.body.name = n \wedge m.body.price = p$$

This proves the validity of the first property for the case where there is only one user in the system. Thanks to the composability supported in our framework, the case, with more users is easily derived. Let us assusme another user $u_1$ with the identifier $\beta \neq \alpha$ such that:

$$\begin{array}{lcl} u_1.id & = & \beta, \\ u_1.reachability & = & \langle Email \rangle, \\ u_1.mailboxes & = & \{\langle Email \rangle \mapsto \{\}, \langle SMS \rangle \mapsto \{\}\}, \\ u_1.adresses & = & \{\langle Email \rangle \mapsto Address_{\beta_1}, \langle SMS \rangle \mapsto Address_{\beta_2}\}, \\ u_1.callback & = & \langle uscallback_\beta \rangle. \end{array}$$

It can also be checked that this user's invariant holds. We subscribe this user to the same set of events as the previous user, resulting in the following binding:

$$\begin{array}{lll} \mathcal{B}_3 \triangleq \{ & \langle \textbf{impl-smtpserver} \rangle & \mapsto \{e : Message \cdot e.medium = \langle Email \rangle\}, \\ & \langle \textbf{impl-smsserver} \rangle & \mapsto \{e : Message \cdot e.medium = \langle SMS \rangle\}, \\ & \langle \textbf{impl-uscallback}_\beta \rangle & \mapsto \{e : StockEvent \cdot e.price \leq max_\alpha \wedge e.price \geq min_\alpha\}, \\ & \langle \textbf{impl-uscallback}_\alpha \rangle & \mapsto \{e : StockEvent \cdot e.price \leq max_\alpha \wedge e.price \geq min_\alpha\}, \\ & \langle \textbf{mskip} \rangle & \mapsto \{x : Event\}\} \end{array}$$

The behavioral specification of *impl-broker* becomes:

$$\begin{array}{l} behavioral\text{-}broker(name : StockName, price : \mathbb{N}) \triangleq \\ \quad mkStockEvent(name,\ price); \\ \quad \textbf{if}\ \ v_1.price \in [min_\alpha, max_\alpha]\ \ \textbf{then}\ \ uscallback_\alpha(v_1) \| uscallback_\beta(v_1) \\ \quad \textbf{else}\ \ mskip(v_1)\ \ \textbf{fi}\ \ \textbf{fi} \end{array}$$

The requirement for non-interference is formulated as:

**Proof Obligation 30** *pre-uscallback*$_\alpha$, *post-uscallback*$_\alpha$ **stable when** *post-uscallback*$_\beta$

**Proof Obligation 31** *pre-uscallback*$_\beta$, *post-uscallback*$_\beta$ **stable when** *post-uscallback*$_\alpha$

To discharge these proof obligations, we write *uscallback*$_\alpha$ and *uscallback*$_\beta$ in a form where their pre- and post-conditions are obvious. This is done by successively applying the definition of structural specifications, the conditional rule and the sequential rule, resulting in the following behavioral specifications:

**behavioral-uscallback**$_\alpha$($sevt$ : *StockEvent*) $\triangleq$

wr $users, v_2$

post $v_2.body = sevt \wedge v_2.medium = users(\alpha).reachability \wedge$
$v_2.address = users(\alpha).addresses(v_2.medium) \mid$
$(v_2.medium = \langle Email \rangle \wedge post\text{-}smtpserver(v_2)) \vee$
$(v_2.medium = \langle SMS \rangle \wedge post\text{-}smserver(v_2)) \vee$
$(v_2.medium \notin \{\langle Email \rangle, \langle SMS \rangle\} \wedge post\text{-}mskip(v_2))$

**behavioral-uscallback**$_\beta$($sevt$ : *StockEvent*) $\triangleq$

wr $users, v_2$

post $v_2.body = sevt \wedge v_2.medium = users(\beta).reachability \wedge$
$v_2.address = users(\beta).addresses(v_2.medium) \mid$
$(v_2.medium = \langle Email \rangle \wedge post\text{-}smtpserver(v_2)) \vee$
$(v_2.medium = \langle SMS \rangle \wedge post\text{-}smserver(v_2)) \vee$
$(v_2.medium \notin \{\langle Email \rangle, \langle SMS \rangle\} \wedge post\text{-}mskip(v_2))$

The pre-conditions $pre\text{-}uscallback_\alpha$ and $pre\text{-}uscallback_\beta$ are true and, therefore, stable when $Q$ for any assertion $Q$. On the other hand,

$$post\text{-}uscallback_\alpha \textbf{ stable when } post\text{-}uscallback_\beta$$

is not valid. To see why, assume that $users(\beta).reachability$ and $users(\alpha).reachability$ have the values $\langle SMS \rangle$ and $\langle Email \rangle$ respectively. From this, it results that the following assertion holds:

$$post\text{-}uscallback_\alpha(v_1) \mid post\text{-}uscallback_\beta(v_1) \Rightarrow v_2.medium = \langle SMS \rangle$$

and $v_2.medium = \langle SMS \rangle$ can not result in the validity of $post\text{-}uscallback_\alpha$.

It turns out that our non-interference requirement is too strong. We refine the above behavioral specification by weakening the post-conditions:

**behavioral-uscallback**$_\alpha$($sevt$ : *StockEvent*) $\triangleq$

wr $users, v_2$

post $A \wedge E_1$

**behavioral-uscallback**$_\beta$($sevt$ : *StockEvent*) $\triangleq$

wr $users, v_2$

post $B \wedge E_2$

where:

- $A \stackrel{def}{=} \exists m \in \mathsf{rng}\ users(\alpha).mailboxes(users(\alpha).reachability) \cdot m.body = sevt \wedge$
  $\mathsf{rng}\ \overleftarrow{users}(\alpha).mailboxes(m.medium) \subseteq \mathsf{rng}\ users(\alpha).mailboxes(m.medium)$

- $E_1 \stackrel{def}{=} \forall id \in \mathsf{dom}\ users \cdot id \neq \alpha \Rightarrow \overleftarrow{users}(id) = users(id)$

- $B \stackrel{def}{=} \exists m \in \mathsf{rng}\ users(\beta).mailboxes(users(\beta).reachability) \cdot m.body = sevt \wedge$
  $\mathsf{rng}\ \overleftarrow{users}(\beta).mailboxes(m.medium) \subseteq \mathsf{rng}\ users(\beta).mailboxes(m.medium)$

- $E_2 \stackrel{def}{=} \forall id \in \mathsf{dom}\ users \cdot id \neq \beta \Rightarrow \overleftarrow{users}(id) = users(id)$

The proof obligations for these refinements are:

**Proof Obligation 32** $post\text{-}uscallback_{\alpha}(sevt) \Rightarrow A \wedge E_1$

**Proof Obligation 33** $post\text{-}uscallback_{\beta}(sevt) \Rightarrow B \wedge E_2$

These proof obligations are discharged by natural deduction as above. We distinguish the case where the value of $users(\alpha).reachability$ is $\langle Email \rangle$ from the case where it is $\langle SMS \rangle$.

**from** $post\text{-}uscallback_\alpha(sevt) \wedge users(\alpha).reachability = \langle Email \rangle \wedge \overleftarrow{\mathsf{inv}\text{-}System}$

$v_2.body = sevt \wedge v_2.medium = \langle Email \rangle \wedge v_2.address = users(\alpha).addresses(\langle Email \rangle) \mid$
$v_2.medium = \langle Email \rangle \wedge post\text{-}smtpserver(v_2) \wedge v_2 = \overleftarrow{v_2}$

**from** $post\text{-}smtpserver(v_2) \wedge users(\alpha).addresses(\langle Email \rangle) = v_2.address \wedge v_2.body = sevt$
$v_2 \in users(\alpha).mailboxes(\langle Email \rangle))$
**infer** $\exists m \in \mathsf{rng}\ users(\alpha).mailboxes(users(\alpha).reachability) \cdot m.body = sevt$

**from** $post\text{-}smtpserver(v_2) \wedge users(\alpha).addresses(\langle Email \rangle) = v_2.address \wedge v_2.body = sevt$
**infer** $\forall m : Medium \cdot m \neq \langle Email \rangle \Rightarrow \overleftarrow{users}(\alpha).mailboxes(m) = users(\alpha).mailboxes(m)$

$A$

**from** $\overleftarrow{\mathsf{inv}\text{-}System} \wedge post\text{-}smtpserver(v_2)$
**infer** $\mathsf{inv}\text{-}System$

**from** $\overleftarrow{inv\text{-}System} \wedge post\text{-}mkMessage(sevt)$
**infer** $\mathsf{inv}\text{-}System$

$\mathsf{inv}\text{-}System$
$\forall u_1, u_2 \in \mathsf{rng}\ users \cdot u_1 \neq u_2 \Rightarrow \mathsf{rng}\ u_1.addresses \cap \mathsf{rng}\ u_2.addresses = \{\}$
$\forall id \in \mathsf{dom}\ users \cdot id \neq \alpha \Rightarrow users(\alpha).addresses(\langle Email \rangle) \notin \mathsf{rng}\ users(id).addresses$

**from** $post\text{-}smtpserver(v_2) \wedge v_2.address = users(\alpha).addresses(\langle Email \rangle)$

**from** $id \neq \alpha \wedge id \in \mathsf{dom}\ users \wedge v_2.address \notin \mathsf{rng}\ users(id).addresses$
**infer** $\overleftarrow{users}(id) = users(id)$

$\forall id \in \mathsf{dom}\ users \cdot v_2.address \notin \mathsf{rng}\ users(id).addresses \Rightarrow users(id) = \overleftarrow{users}(id)$
**infer** $\forall id \in \mathsf{dom}\ users \cdot id \neq \alpha \Rightarrow \overleftarrow{users}(id) = users(id)$

$E_1$

**infer** $A \wedge E_1$

The second part of the proof is obtained by simply replacing $\langle Email \rangle$ with $\langle SMS \rangle$ and *post-smtpserver* with *post-smsserver*. The proof that $B \wedge E_2$ follows from $post\text{-}uscallback_\beta$ is also done very similarly.

We now proceed to showing that $A$ is stable when $B \wedge E_2$, that is $A \mid B \wedge E_2 \Rightarrow A$. The proof is also by natural deduction.

**from** $A \mid B \wedge E_2$
$A \mid E_2$
**from** $E_2 \wedge \alpha \neq \beta$
**infer** $users(\alpha) = \overleftarrow{users}(\alpha)$
$A \mid users(\alpha) = \overleftarrow{users}(\alpha)$
**infer** $A$

The stability of $B$ when $A \wedge E_1$ is constructed similarly.

*We have, hence, shown that the reception of messages by one user is not affected if another user subscribes to the same messages.*

Let us now suppose that the user $\beta$ is rather interested in some messages that the user $\alpha$ is not interested in. A subscription of the user $\beta$ would be such that:

$$subscription_\beta = \{x : StockEvent \cdot s.price \in [min_\beta, max_\beta]\} \text{ where } max_\beta < min_\alpha.$$

From this, a new binding is derived:

$$\begin{array}{lll} \mathcal{B}_3 \triangleq \{ & \langle \textbf{impl-smtpserver} \rangle & \mapsto \{e : Message \cdot e.medium = \langle Email \rangle\}, \\ & \langle \textbf{impl-smsserver} \rangle & \mapsto \{e : Message \cdot e.medium = \langle SMS \rangle\}, \\ & \langle \textbf{impl-uscallback}_\beta \rangle & \mapsto \{e : StockEvent \cdot e.price \leq max_\beta \wedge e.price \geq min_\beta\}, \\ & \langle \textbf{impl-uscallback}_\alpha \rangle & \mapsto \{e : StockEvent \cdot e.price \leq max_\alpha \wedge e.price \geq min_\alpha\}, \\ & \langle \textbf{mskip} \rangle & \mapsto \{x : Event\}\} \end{array}$$

The behavioral specification of *impl-broker* becomes:

$$\begin{array}{l} behavioral\text{-}broker(name : StockName, price : \mathbb{N}) \triangleq \\ \quad mkStockEvent(name,\ price); \\ \quad \textbf{if}\ \ v_1.price \in [min_\alpha, max_\alpha]\ \ \textbf{then}\ \ uscallback_\alpha(v_1) \\ \quad \textbf{else if}\ \ v_1.price \in [min_\beta, max_\beta]\ \ \textbf{then}\ \ uscallback_\beta(v_1) \\ \quad \textbf{else}\ \ mskip(v_1)\ \ \textbf{fi}\ \ \textbf{fi} \end{array}$$

and the proof developed for discharging proof obligation 32 remains valid. Subsequently, the user with identifier $\alpha$ still receives all messages she is interested in.

*We, therefore, conclude that the reception of messages by one user is not affected by the reception of messages by the other user; we can subscribe as many users as we want, each of them will receive all messages she is interested in.*

Note, however, that the case of the parallel announcement of two messages by brokers (i.e. the concurrent execution of two brokers) have not been investigated. In fact, executing for instance $impl\text{-}broker(a, b)$ concurrently with $impl\text{-}broker(d, e)$ results in interference on the variable $v_1$. To solve this issue the body of the method *impl-broker* can be embedded in an await-construct.

### 13.5.2 PROPERTY II

This property requires showing that users always receive messages through the medium that they specify. This property is important because otherwise users would not be able to rely on such a system.

In fact, the proof of the first property also includes the proof of this property. It is enough to observe in the proofs that:

- if the communication medium is $\langle Email \rangle$ in the hypothesis, then, the conclusion ensures the existence of the indicated message in the Email mailbox;
- if the communication medium is $\langle SMS \rangle$ in the hypothesis, then, the conclusion ensures the existence of the indicated message in the SMS mailbox.

Obviously, many other global properties can be investigated, some of them requiring no modification of the specifications, others requiring either refinement or complete modification of some specifications. After the verification of these properties based on the behavioral specifications, the implementation of the components can be carried out starting with the structural specifications and following a stepwise development process, that does not differ much from the traditional top-down development processes. We omit this step.

## 13.6 SUMMARY

This chapter presented a first real life case study in the design and analysis of correct event-based applications, a stock service for mobile users (SMU). This case study is motivated by the increasing requirement for supporting user awareness in distributed multi-user applications. More precisely, SMU is inspired by the MOTION messaging system and the IBM stock service application. The design presented in this chapter is superior to that of these systems because it supports many communication media, allowing end-users to specify the medium through which they would like to be notified. This system takes advantage of the loose coupling facility provided by the event-based paradigm such that new communication media can be added to the system by simply performing a corresponding subscription.

The design process of the application presented in this chapter follows the standard steps in the development of a LECAP applications: designing the architecture of the application, specifying the components involved in the construction of the application, verifying some local properties about these specifications, composing the specification of the intended application starting with the specifications of the components, verification of the global properties of the application. The final step in this process concerns the stepwise top-down development of the component based on their structural specifications. We have

not performed this step as it resembles the traditional stepwise development processes. Moreover, performing such a top-down development process for the SMU application would clearly be out of the scope this thesis.

The next chapter discusses the redesign of the MOTION platform that motivated the construction of the LECAP methodology. Like SMU, the analysis of the MOTION platform is done within the SEATY framework where rely- and guarantee conditions are replaced with the implicit requirement that any two processes running in parallel access two different sets of variables.

# Chapter 14

# Redesigning the MOTION Platform

MOTION (MObile Teamwork Infrastructure for Organizations Networking) is a platform we designed and prototyped in the MOTION European project [80]. This platform addresses the needs of two well known organizations. The first is a manufacturer of mobile phones and the second is a producer of white goods (e.g. refrigerators, washing machines). The platform has a service architecture that supports mobile teamworking by taking into account different connectivity modes of users, provides access support for various devices, supports distributed search of users and artifacts, offers user management facilities in a way where users can be grouped in communities.

## 14.1 Background

### 14.1.1 The MOTION Architecture

The MOTION system was constructed by assembling different components: DUMAS (Dynamic User Management and Access Control System) [43], an XQL engine [59], a repository (comparable to a file system), an artifact manager (comparable to a shell that provides primitives for accessing the file system), etc. These components are integrated into the platform by means of the event-based architectural style. A layered view of the MOTION platform is presented in Figure 14.1. The bottom layer of the architecture provides the communication infrastructure (realized by PeerWare [101]). The services made available by this middleware include: peer-to-peer file sharing and an event-based system.

On top of this layer, we constructed the teamwork services layer (TWS layer). This TWS layer is composed of DUMAS, a repository, a messaging system, a user oriented publish-subscribe system (TWS P/S), a component for distributed search, and an artifact management component. The functionalities of DUMAS include user management, community management, and access control. The repository is a component responsible for storing different MOTION data such as files, profiles, and access control lists. A conceptual view of the MOTION platform is shown on Figure 2.2. A detailed description of MOTION is presented in Chapter 2 as well as in [79, 80, 81, 82, 109].

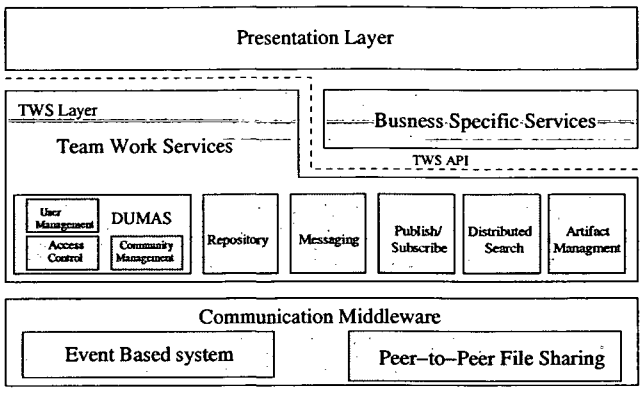


Figure 14.1: The MOTION Architecture

### 14.1.2 The Peer-to-Peer Side of MOTION

This section elaborates on the meaning of peer-to-peer (p2p) for the MOTION platform. The choice of this architecture is justified in [80]. The p2p concept has two facets in the MOTION platform. First, file sharing is performed in a p2p manner; each device has complete control of the files it makes available to the remainder of the system.
The next facet of the p2p architectural style in the MOTION platform is service orientation. Each device may host and manage a service independently of the behavior of other devices. Such a device uses the EB paradigm for notifying other devices of the changes concerning the service it provides. In fact, this facet of p2p may be viewed as a generalization of p2p file-sharing. We, however, separate the two concerns since file-sharing is one of the most advocated services in p2p systems. We call each device in this p2p architecture a peer. This chapter is concerned about the second facet of p2p in MOTION. In particular, we address user management in a p2p environment.

## 14.2 Component Specification

The user management functionality has gained increasing attention and importance in distributed environments. The responsibilities of a user management component have been integrated in many implementations so far–be it as part of an operating system (such as Unix) or as pieces of (vendor-specific) software. Our dynamic user management component (DUMAS) includes functionalities such as access control, user and group creation and deletion, as well as user and profiles manipulation. The initial version of DUMAS was formally specified [43, 44], the specification validated against the informal requirements [43] and used for the construction of an automated oracle [44]. DUMAS was shown to be robust in client-server settings. In the MOTION's event-based peer-to-peer environment, however, there were some severe malfunctions that were mainly due to interference that

could not be observed in client- server settings.

The MOTION platform (see Figure 2.2) supports various kinds of devices that have different capabilities and that cannot, therefore, be equally used for storing data such as user profiles and access control lists. To support this heterogeneity of devices we give the end-users the possibility to specify which profiles they would like to store on their devices. For instance, an end-user, say *Joe*, may configure his system such that only profiles of colleagues in his department are stored on his desktop computer while only profiles of those in the projects he works on are stored on his PDA. In terms of services, we may say that each device hosts a user management service. Although the implementation of this service is the same for all peers, the content of the repository is not the same; *Joe*'s profile may be stored on peers 123, 124, and 125 but not on peers 234, 235, and 236. One of the main challenges in such an architecture is to keep the profiles of all users consistent. Any change to *Joe's* profile performed on Peer 123 needs to be taken into consideration on (perhaps propagated to) the peers 124 and 125.

Although this may resemble the traditional data-consistency requirement in distributed systems, there are, however, some subtle differences. First, the peer 124 may suddenly decide not to be interested in *Joe*'s profile anymore (depending on the interest of the owner of the peer). Next, the user management service on peer 123 has no knowledge about peers interested in events it announces. Third, each peer stores only profiles it is subscribed to. Finally, a peer interested in *Joe*'s profile may be offline when some changes are made to this profile.

As in Chapters 9 and 10, we use a notation that resembles the VDM-SL [102] notation. VDM (Vienna Development Method) is an environment for developing correct model-oriented applications. The method is composed of a formalism for specification (VDM-SL) and techniques for stepwise refinements. This systematic development approach makes it suitable for the development of complex software systems. The language can be used for abstract specification as well as for low-level specification. VDM-SL supports representational and operational abstractions. Representational abstraction describes the modeling primitives necessary to specify a software program. A complete description of the language can be found in [130].

### 14.2.1 Data Modeling

We present some of the types defined in our specification and useful for understanding this chapter. Access control models are based on three notions: *principals*, *subjects* and *access rights*. Informally, a principal is anything capable of possessing access rights. In our model, we identify two types of principals: *users* and *groups*. Each user/group has an identifier. An identifier is a type which is not further defined (declared with `token`).

$$
\begin{array}{lcl}
\mathit{ID} & = & \mathit{token}; \\
\mathit{UserID} & = & \mathit{ID}; \\
\mathit{GroupID} & = & \langle \mathit{default} \rangle \mid \mathit{ID}; \\
\mathit{RightID} & = & \mathit{ID};
\end{array}
$$

User, group, and access right identifiers are some kind of identifiers. A principal is either a user or a group. This can be expressed using the union operator:

$$
\mathit{PrincipalID} \quad = \quad \mathit{UserID} \mid \mathit{GroupID};
$$

A basic subject is anything (other than an access right or a principal) on which an access right may be owned (e.g. files). The only requirement on these elements is to have an identifier. Basic subjects are not registered in the repository (defined below); we have no control over when they are created and destroyed. They are mentioned because users and groups may own access rights on them.

$$
\begin{array}{lcl}
\mathit{BasicSubjectID} & = & \mathit{token}; \\
\mathit{Profile} & = & \mathit{token};
\end{array}
$$

A profile is a set of user specific information such as her expertise, her languages, her time-zone, her gender, her location and her names. Such profiles may have complex structures that we do not want to specify at this level, hence, giving developers more possibilities in the choice of the concrete data structure.

A user is modeled with a set of data related to it: its parents, its access rights, its identifier, and its profile. Each user has at least one parent: its main parent. It can be linked to other groups called parents of this user. The set of parents contains at least the main parent of the user.

$$
\begin{array}{lll}
\mathit{User} :: & & \\
\quad \mathit{parents} & : & \mathit{GroupID}\text{-set} \\
\quad \mathit{mainparent} & : & \mathit{GroupID} \\
\quad \mathit{permissions} & : & (\mathit{RightID} \times \mathit{SubjectID})\text{-set} \\
\quad \mathit{name} & : & \mathit{UserID} \\
\quad \mathit{profile} & : & \mathit{Profile} \\
\mathsf{inv} \quad \mathit{us} \triangleq & & \\
\quad \mathit{us.mainparent} \in \mathit{us.parents}; & &
\end{array}
$$

A group is a set of users sharing some permissions. A group does not need to have a parent; if it, however, has some parents, then, one of them must be its main parent. If it has no parent (i.e. the main parent is nil ), then this group must be the group named $\langle \mathit{default} \rangle$.

$$
\begin{array}{lll}
\mathit{Group} & :: & \\
 & \mathit{members} & : \ \mathit{UserID}\text{-set} \\
 & \mathit{parents} & : \ \mathit{GroupID}\text{-set} \\
 & \mathit{mainparent} & : \ [\mathit{GroupID}] \\
 & \mathit{permissions} & : \ (\mathit{RightID} \times \mathit{SubjectID})\text{-set} \\
 & \mathit{name} & : \ \mathit{GroupID} \\
 & \mathit{profile} & : \ \mathit{Profile} \\
\mathsf{inv} & \mathit{us} \triangleq & \\
 & \multicolumn{2}{l}{\mathit{us.mainparent} \neq \mathsf{nil} \ \Rightarrow \ \mathit{us.mainparent} \in \mathit{us.parents};} \\
 & \multicolumn{2}{l}{\mathit{us.mainparent} = \mathsf{nil} \ \Leftrightarrow \ \mathit{us.name} = \langle \mathit{default} \rangle \wedge \mathit{us.parents} = \{\};}
\end{array}
$$

The type *Principal* is the union of the types *User* and *Group*. It combines these two types into a single one.

$$\mathit{Principal} \ = \ \mathit{Group} \mid \mathit{User};$$

In addition to these basic types, the type *Right* whose definition is omitted is also defined. A subject is the union of the types access rights and principals.

$$
\begin{array}{lll}
\mathit{SubjectID} & = & \mathit{PrincipalID} \mid \mathit{RightID}; \\
\mathit{Subject} & = & \mathit{Principal} \mid \mathit{Right};
\end{array}
$$

We introduce the enumeration type *Prog* for referring to operations defined in this specification and that are elements of the event-based system's set of methods. We assume a system with a finite number of peers; each operation $\mathit{operation}_i$ corresponds to the operation *operation* running on peer $i$.

$$\mathit{Prog} \ = \ \langle \mathit{impl\text{-}ebsimpleMu}_1 \rangle \mid \cdots \mid \langle \mathit{impl\text{-}ebsimpleMu}_n \rangle \mid \langle \mathit{mskip} \rangle;$$

We also introduce a type *EventName* that introduces a classification over events. The only event name needed in the extract presented in this thesis is $\langle \mathit{UserProfileUpdate} \rangle$.

$$\mathit{EventName} \ = \ \langle \mathit{UserProfileUpdate} \rangle;$$

An event is a composite type including the identifier of the announcing peer, a tag for identifying the changes performed on the state, and a payload.

$$
\begin{array}{lll}
\mathit{Event} & :: & \\
 & \mathit{peerid} & : \ \mathbb{N} \\
 & \mathit{action} & : \ \mathit{EventName} \\
 & \mathit{payload} & : \ \mathit{Subject};
\end{array}
$$

$$Subscription \quad = \quad Event\text{-set};$$

A subscription represents the set of events a peer is interested in.

A binding associates each program (element of type *Prog*) to a subscription, i.e. the set of events the program is subscribed to.

$$Binding \quad = \quad Prog \xrightarrow{m} Subscription;$$

We define the key model of our specification, the repository. It is a map of subjects to their identifiers. The first invariant requires that any identifier be mapped to an element with the same identifier. The second and third invariants require that the invariants of any user and group in the repository be satisfied. There are other invariants that are omitted.

$$DB = SubjectID \xleftrightarrow{m} Subject$$

$$\begin{array}{l}
\mathsf{inv}\ db \triangleq \\
\quad \mathsf{let} \\
\qquad inv0 = \forall x \in \mathsf{dom}\ db \cdot x = db(x).name, \\
\qquad inv1 = \forall x \in \mathsf{rng}\ db \cdot \mathsf{is}\text{-}User(x) \ \Rightarrow\ \mathsf{inv}\text{-}User(x), \\
\qquad inv2 = \forall x \in \mathsf{rng}\ db \cdot \mathsf{is}\text{-}Group(x) \ \Rightarrow\ \mathsf{inv}\text{-}Group(x)\ \mathsf{in} \\
\quad inv0 \wedge inv1 \wedge inv2
\end{array}$$

We have advocated that the different devices in the MOTION platform may cache a part of the whole set of information available in the system. Each of these peers, hence, hosts a local repository. The state of the MOTION platform is, thus, composed of a sequence of repositories, each corresponding to a different device.

$$\begin{array}{l}
\mathsf{state}\ System\ \mathsf{of} \\
\quad db : DB^*, \\
\quad \mathcal{B} : Binding \\
\quad \mathsf{inv}\ \mathsf{mk}\text{-}System(db, binding) \triangleq \\
\qquad \forall i \in [1, \mathsf{len}\ db] \cdot \mathsf{inv}\text{-}DB(db[i]) \\
\quad \mathsf{init}\ sys \triangleq \\
\qquad \forall i \in [1,\ \mathsf{len}\ db] \cdot db_i = \{\mapsto\} \\
\mathsf{end}
\end{array}$$

The invariant of the state requires that each peer has a local repository whose invariant holds. The binding is currently undefined as we do not know yet which program must be subscribed to which events.

### 14.2.2 Specification of Components

A number of operations are specified in our model. As an example, we show the structural specification of the operation for updating user profiles. This operation replaces the profile of the given user with the provided profile. This operation is indexed with the identifier of the peer on which it is running. We, therefore, have a number of len $db$ such operations in the system. Each such operation only accesses the repository with the same index. The letter $i$ is used in the following specifications as the name of the current peer and $db_i$ is a shortcut for $db[i]$. Note that although these operations could be defined as higher order functions that take the identifier of the peer and return the corresponding operation, we prefer indexing operations since it is simpler and more intuitive. An alternate solution would have been to use objects in which we could encapsulate the identifier of the peer, its repository and the operations running on this peer. Objects are, however, not yet suppoted by our framework.

The operation *impl-muProfile*$_i$ (the name *muProfile* is an abbreviation of *updateProfile*; *mu* is used in analogy to the VDM operation $\mu$ for updating composite types) satisfying the specification *muProfile*$_i$ is intended to replace the local profile of the user with the given identifier with the provided profile. The structural specification of this operation is the sequential composition of *simpleMu*$_i$ and an event announcement.

$$\begin{array}{l}\textbf{muProfile}_i\textbf{(id : UserID, prof : Profile)} \triangleq \\ \quad \textbf{await true do } \mathit{simpleMu}_i(\mathit{id},\ \mathit{prof}) \textbf{ od}; \\ \quad \textbf{announce}(v_1)\end{array}$$

The operation *impl-simpleMu*$_i$ (satisfying the specification *simpleMu*$_i$) is the basis for updating user profiles. The post-condition ensures that after execution of the operation, the local repository maps the given user identifier to a user with the given profile while keeping other information in the repository unchanged. In addition, this method prepares the event to be announced through the announce construct of *muProfile*$_i$.

$$\begin{array}{l}\textbf{simpleMu}_i\textbf{(id : UserID, pr : Profile)} \triangleq \\ \quad \mathsf{wr} \quad db \\ \quad \mathsf{pre} \quad id \in \mathsf{dom}\ db_i \\ \quad \mathsf{post} \quad db_i(id) = \mu\,(db_i\,(id),\ \mathit{profile} \mapsto pr) \wedge db_i \lhd \{id\} = \overleftarrow{db_i} \lhd \{id\} \wedge \\ \qquad v.peerid = i \wedge v.action = \langle \mathit{UserProfileUpdate} \rangle \wedge v.payload = db_i(id) \ \wedge \\ \qquad \forall t \in [1, \mathsf{len}\ db] \cdot t \neq i \ \Rightarrow \ db_t = \overleftarrow{db_t}.\end{array}$$

The operation $\mu$ is the VDM operator for updating composite types such as *User* and *Group*. In the above specification, the composite element $db(us)$ is updated by replacing the value of the field *profile* with *pr* and keeping other fields unchanged. For instance,

**if** $us_2 = \mu(us_1, profile \mapsto pr_2)$
**then**

$us_2.parents = us_1.parents$
$us_2.mainparent = us_1.mainparent$
$us_2.permissions = us_1.permissions$
$us_2.name = us_1.name$
$us_2.profile = pr_2$

The operation $\lhd$ (domain restricted by) restricts the domain of a map to those elements that are not in the given set. For instance,

$$\{id_1 \mapsto us_1, id_2 \mapsto us_2, id_3 \mapsto us_3\} \lhd \{id_2\} = \{id_1 \mapsto us_1, id_3 \mapsto us_3\}.$$

The method *impl-muProfile*$_i$ is provided to be invoked either by end-user or by other methods. It can, however, not be invoked by the event-based infrastructure for two reasons. First, since it announces an event, it is not suitable for being invoked for replicating actions performed on other peers. Next, the event-based infrastructure requires that any subscriber has a header of the form $op(x : Event)$. We, therefore, propose another method *impl-ebsimpleMu* (the name is an abbreviation of event-based update profile), that basically does the same thing as *impl-simpleMu*$_i$ but requires an event as input value. Its specification *ebsimpleMu* is given as follows:

**ebsimpleMu$_i$(e : Event)** $\triangleq$ **await true do** *simpleMu$_i$*(*e.body.name*, *e.body.profile*) **od**

Due to the different capabilities that peers have, not every peer can store information on all users. The end-users are, therefore, given an operation *impl-userSubscription* satisfying *userSubscription* for defining the kind of information they would like to store on their peers. The input element of this operation is a subscription that characterizes some users. The pre-condition prevents this subscription to match events announced by the current peer as this would result into loops where operations would be subscribed to events they announce. The post-condition is a corresponding update of the binding.

**userSubscription$_i$(s : Subscription)** $\triangleq$
pre $\forall e \in s \cdot e.peerid \neq i \ \wedge \ e.action = \langle UserProfileUpdate \rangle$
post $\mathcal{B} = \overleftarrow{\mathcal{B}} \dagger \{impl\text{-}ebsimpleMu_i \mapsto s \cup \overleftarrow{\mathcal{B}}(impl\text{-}ebsimpleMu_i)\}$

An operation *impl-userUnsubscribe* satisfying *userUnsubscribe* is also provided for removing subscriptions. This operation not only removes the given subscription from the binding, but also deletes the entries that match this subscription from the local repository; otherwise, this repository will be inconsistent with the remainder of the system.

$$
\begin{array}{l}
\textbf{userUnsubscribe(e : Event)} \triangleq \\
\quad \mathsf{wr} \quad db_i \\
\quad \mathsf{post} \quad \mathsf{let} \\
\qquad\qquad m = \langle \mathit{impl\text{-}ebsimpleMu_i} \rangle \\
\qquad\qquad X = \{u \in \mathsf{rng}\ db_i \mid \exists\ e \in s \cdot e.payload = u\} \\
\qquad \mathsf{in} \quad db_i \lhd \{e.body.name\} = \overleftarrow{db_i} \lhd \{e.body.name\} \wedge \\
\qquad\qquad \mathcal{B} = \overleftarrow{\mathcal{B}} \dagger \{m \mapsto \overleftarrow{\mathcal{B}}(m) \setminus \{s\}\} \wedge \mathsf{rng}\ db_i \cap X = \emptyset
\end{array}
$$

The specifications are currently structural since we have not defined the binding. The set of global variables is composed of the repositories of peers.

## 14.3 Verification of Local Properties

Before defining the binding of the event-based system, we want to prove some local properties, namely that any operation that satisfies one of the above specifications conserves the invariants of the repository. For this, we assume the empty binding and derive the behavioral specifications based on which we can discharge the proof obligations.

The emptiness of the binding results for any event $e$ in:

$$\mathit{subscribers}(e) = \{\textbf{mskip}\}$$

By the the skip rule we deduce the following behavioral specification that shows that $\mathit{muProfile_i}$ coincides with $\mathit{simpleMu_i}$ when the binding is empty.

$$
\begin{array}{l}
\textbf{behavioral-muProfile}_\textbf{i}\textbf{(id : UserID, prof : Profile)} \triangleq \\
\quad \textbf{await true do } \mathit{simpleMu_i}(\mathit{id},\ \mathit{prof}) \textbf{ od}
\end{array}
$$

Since, however, the operation $\mathit{impl\text{-}muProfile_i}$ is currently considered in an interference free environment, we deduce:

$$\textbf{behavioral-muProfile}_\textbf{i}\textbf{(id : UserID, prof : Profile)} \triangleq \mathit{simpleMu_i}(\mathit{id},\ \mathit{prof})$$

The proof obligation is subsequently formulated as:

**Proof Obligation 34** $\forall\ u : \mathit{UserID},\ p : \mathit{Profile} \cdot \mathit{post\text{-}simpleMu_i}(u, p) \Rightarrow \mathit{inv\text{-}DB}$

Any attempt to discharge this proof obligation, however, fails; the initial state may be such that its invariant is not satisfied. We, therefore, need to strengthen our assumptions on the environment. We also apply the pre-rule to add this information into the post-condition, resulting in:

$$
\begin{array}{l}
\textbf{simpleMu}_i\textbf{(id : UserID, pr : Profile)} \triangleq \\
\quad \mathsf{wr} \quad db \\
\quad \mathsf{pre} \quad id \in \mathsf{dom}\ db_i \wedge \mathsf{inv}\text{-}System \\
\quad \mathsf{post} \quad db_i(id) = \mu\,(db_i\,(id),\ profile \mapsto pr) \wedge db_i \lhd \{id\} = \overleftarrow{db_i} \lhd \{id\} \wedge \\
\qquad v.peerid = i \wedge v.action = \langle UserProfileUpdate \rangle \wedge v.payload = db_i(id) \wedge \\
\qquad \overleftarrow{\mathsf{inv}\text{-}System} \wedge \forall t \in [1, \mathsf{len}\ db] \cdot t \neq i \ \Rightarrow\ db_t = \overleftarrow{db_t}.
\end{array}
$$

The argumentation on the validity of the proof obligation is now straightforward. One must show that:

- for any entry $x$ in the domain of $db_i$, the invariant of $db_i(x)$ is satisfied and
- for any $x$ in the domain of $db_i$, $db_i(x).name = x$.

The proof is by natural deduction:

$$
\begin{array}{ll}
\textbf{from} & post\text{-}muProfile_i(id, pr) \wedge id \in \mathsf{dom}\ \overleftarrow{db_i} \\
& simpleMu_i(id, pr) \\
& \overleftarrow{\mathsf{inv}\text{-}System} \\
& \overleftarrow{\mathsf{inv}\text{-}DB(db_i)} \\
& db_i(id) = \mu\,(db_i\,(id),\ profile \mapsto pr) \\
& db_i \lhd \{id\} = \overleftarrow{db_i} \lhd \{id\} \\
\\
\quad \textbf{from} & db_i \lhd \{id\} = \overleftarrow{db_i} \lhd \{id\} \wedge \overleftarrow{\mathsf{inv}\text{-}DB(db_i)} \\
& \forall x \in \mathsf{dom}\ db_i \cdot x \neq id \ \Rightarrow\ db_i(x) = \overleftarrow{db_i}(x) \\
\qquad \textbf{from} & \overleftarrow{\mathsf{inv}\text{-}DB(db_i)} \wedge x : GroupID \wedge x \in \mathsf{dom}\ \overleftarrow{db_i} \\
& \mathsf{inv}\text{-}Group(\overleftarrow{db_i}(x)) \\
& x \neq id \\
& db_i(x) = \overleftarrow{db_i}(x) \\
& \mathsf{inv}\text{-}Group(db_i(x)) \\
\qquad \textbf{infer} & \forall x : GroupID \cdot \mathsf{inv}\text{-}Group(db_i(x)) \\
\qquad \textbf{from} & \overleftarrow{\mathsf{inv}\text{-}DB(db_i)} \wedge x : UserID \wedge x \neq id \\
& \mathsf{inv}\text{-}User(\overleftarrow{db_i}(x)) \\
& db_i(x) = \overleftarrow{db_i}(x) \\
& \mathsf{inv}\text{-}User(db_i(x)) \\
\qquad \textbf{infer} & \forall x : UserID \cdot x \neq id \ \Rightarrow\ \mathsf{inv}\text{-}User(db_i(x)) \\
\textbf{infer} & \forall x : UserID \cdot x \neq id \ \Rightarrow\ \mathsf{inv}\text{-}User(db_i(x)) \wedge \forall x : GroupID \cdot \mathsf{inv}\text{-}Group(db_i(x))
\end{array}
$$

$$
\begin{array}{ll}
\textbf{from} & db_i(id) = \mu\,(db_i\,(id),\ profile \mapsto pr) \wedge \overleftarrow{\mathsf{inv}\text{-}DB}(db_i) \\
& \mathsf{inv}\text{-}User(\overleftarrow{db_i}(x) \\
\textbf{infer} & \mathsf{inv}\text{-}User(db_i(x) \\
\\
\textbf{from} & db_i \lhd \{id\} = \overleftarrow{db_i} \lhd \{id\} \wedge \overleftarrow{\mathsf{inv}\text{-}DB}(db_i) \\
& \forall x \in \mathsf{dom}\ db_i \cdot x \neq id \ \Rightarrow\ db_i(x) = \overleftarrow{db_i}(x) \\
& \forall x \in \mathsf{dom}\ db_i \cdot \overleftarrow{db_i}(x).name = x \\
& \quad \textbf{from} \quad db_i(id) = \mu\,(db_i\,(id),\ profile \mapsto pr) \wedge \overleftarrow{db_i}(id).name = id \\
& \quad \textbf{infer} \quad \overleftarrow{db_i}(id).name = id \\
& \textbf{infer} \quad \forall x \in \mathsf{dom}\ db_i \cdot \overleftarrow{db_i}(x).name = x \\
\textbf{infer} & \mathsf{inv}\text{-}DB(db_i)
\end{array}
$$

The second proof obligation is that the operation *impl-ebsimpleMu*$_i$ also conserves the invariant of the local repository. The proof is by adequately refining this operation (as in the previous case) and is obtained from the above proof by some minor modifications such as replacing *id* with *e.body.id*, and *pr* with *e.body.profile*.

## 14.4 Application Composition

The composition of application specifications is done by subscribing specification of components to to events in a way that reflects the architecture of the desired application.

### 14.4.1 Subscription of Components

In most event-based applications, the developer/designer is responsible for specifying what components are interested in what events and she indeed performs the integration by means of an integration framework. The verification of the system can, hence, be realized statically. The MOTION platform, however, is different; end-users are allowed to subscribe components to events in an existing application. The verification of the properties of this application is consequently performed under some assumptions. In particular, the specification *userSubscription*$_i$ of the operation *impl-userSubscription*$_i$ requires that the subscription submitted by the end-user excludes events announced by the current peer (in this case the peer $i$).

Let us assume that the owner of the peer $i$ is interested in caching profiles of users whose data satisfy the subscription $s$ which further satisfies the pre-condition of *userSubscription*$_i$. After expressing this need (using an operation satisfying *userSubscription*$_i$), the binding is such that:

$$\mathcal{B} = \overleftarrow{\mathcal{B}} \dagger \{impl\text{-}ebsimpleMu_i \mapsto s \cup \overleftarrow{\mathcal{B}}(impl\text{-}ebsimpleMu_i)\}$$

which requires the event-based infrastructure to invoke the operation $impl\text{-}ebsimpleMu_i$ when an event matches the subscription $s$. If, for instance, we start the system with an empty binding, then, if the owner of the peer $i$ is interested in all user profile updates, the binding will be following:

$$\begin{aligned}\mathcal{B}_1 = \{\ & impl\text{-}ebsimpleMu_i \mapsto \{x : Event \cdot\ x.action = \langle UserProfileUpdate \rangle \wedge x.peerid \neq i\}\},\\ & impl\text{-}ebsimpleMu_j \mapsto \{\},\\ & impl\text{-}ebsimpleMu_k \mapsto \{\},\\ & \mathbf{mskip} \mapsto \{x : Event\}\}\end{aligned}$$

Some could argue that including the tag $\langle UserProfileUpdate \rangle$, which may be an operation name, in the event contradicts the spirit of the event-based paradigm which is loose coupling of components. First, this is a requirement of this specific application, not of the LECAP methodology which does not even define the meaning of an event. Second, there is a substantial difference with strongly coupled systems, such as those based on method invocation. Here, the subscriber defines the publishers it wants to receive events from (which is legitimate), while in strongly coupled systems based on method invocation, the caller needs to know the name of the callee; this is clearly not the case in our example.

### 14.4.2 IDENTIFICATION OF AFFECTED COMPONENTS

The next step in the composition of an event-based application is the identification of components whose behaviors may be affected by a subscription. Since the operations $impl\text{-}ebsimpleMu_i$ announce no event, subscribing them to an event $e$ only impacts their predecessors; by which we mean operations such that $impl\text{-}ebsimpleMu_i$ is invoked in some of their computations.

Starting with the empty binding, we subscribe the method $impl\text{-}ebsimpleMu_i$ to an event $e$ such that $e.name = \langle UserProfileUpdate \rangle$ and $e.peerid \neq i$ (as required by the pre-condition of $userSubscription_i$). An example of the resulting bindings is $\mathcal{B}_1$ defined above. The identifier $e.peerid$ of the subscribing peer may take any value different from $i$. Therefore, the set of announcers of the event $e$ is

$$announcers(e) = \{impl\text{-}muProfile_j \cdot j \neq i\}$$

and, therefore,

$$predecessors(impl\text{-}ebsimpleMu_i) = \{impl\text{-}muProfile_j \cdot j \neq i\}.$$

The reader must be aware that in general, $predecessor(z)$ defines the set of programs such that the program $z$ is triggered in some of their computations. It, therefore, depends upon the binding. Although we have not defined a concrete binding for computing $predecessors(impl\text{-}ebsimpleMu_i)$ (since the binding are constructed dynamically by the end-users), the assumption that any subscription must be performed with a method satisfying $userSubscription_i$ gives us an upper bound on the set of events that $impl\text{-}ebsimpleMu_i$ may be subscribed to; and, hence, an upper bound of $predecessors(impl\text{-}ebsimpleMu_i)$.

### 14.4.3 Deriving the Behavioral Specifications

We proceed to deriving the behavioral specifications of affected components. This process which is accompanied by the derivation and the discharge of some proof obligations results in specifications that are used for the verification of global properties of the application.

We need to derive the behavioral specification of each $z$ in $predecessors(impl\text{-}ebsimpleMu_i)$. Since, however, each method in $predecessors(impl\text{-}ebsimpleMu_i)$ satisfies a specification of the form $muProfile_j$ (where $j \neq i$), it is enough to take an arbitrary $impl\text{-}muProfile_j$ and derive its behavioral specification.

By the announce rule, we need to determine the set of events of $impl\text{-}muProfile_j$ first.

$$\begin{aligned} events(impl\text{-}muProfile_j) &= \{e : Event \cdot \exists pr : Profile, id : UserID, \overleftarrow{s}, s : System \cdot \\ &\quad (\overleftarrow{s}, s) \models \textbf{post-}muProfile_j(id, pr)\} \\ &= \{e : Event \cdot e.peerid = j \wedge e.action = \langle UserProfileUpdate \rangle\} \end{aligned}$$

And, assuming the binding $\mathcal{B}_1$, we derive that

$$\forall e : Event \cdot e.action = \langle UserProfileUpdate \rangle \Rightarrow subscribers(e) = \{impl\text{-}ebsimpleMu_i, mskip\}$$

which results in

$$\forall e \in events(impl\text{-}muProfile_j) \cdot subscribers(e) = \{impl\text{-}ebsimpleMu_i, mskip\}$$

and the behavioral specification of $impl\text{-}muProfile_j$ is therefore:

**muProfile$_j$(id : UserID, prof : Profile)** $\triangleq$
**await true do** $simpleMu_j(id,\ prof)$ **od**;
$\{impl\text{-}ebsimpleMu_i(v) \| mskip(v)\}$

which by the skip rule can be simplified into:

**muProfile$_i$(id : UserID, prof : Profile)** $\triangleq$
**await true do** $simpleMu_j(id,\ prof)$ **od**;
$impl\text{-}ebsimpleMu_i(v)$

The proof obligation for the appropriate behavior of this operation is given by the sequential rule:

**Proof Obligation 35** **post**-$simpleMu_j(id, pr)$ $\Rightarrow$ **pre**-$ebsimpleMu_i(v)$

This proof obligation, however does not necessarily hold. We strengthen the pre-condition of $impl\text{-}muProfile_j$ with **pre**-$simpleMu_i$ and subsequently derive the following specification:

**muProfile$_j$(id : UserID, pr : Profile)** $\triangleq$
pre **pre**-$simpleMu_i \wedge$ **pre**-$simpleMu_j$
post **post**-$simpleMu_i$ | **post**-$simpleMu_j$

## 14.5 Global System Behavior

Based on the behavioral specifications computed in the previous section, we now want to check the behavior of the whole system.

### 14.5.1 Property I: Consistency

Scalability can be a serious problem in peer-to-peer systems. Among others, one of the obstacles to achieving this scalability in the MOTION platform is the requirement of replica consistency. We propose to analyze our system with respect to this property. In particular, we want to show that all user entries replicated in the system will be consistent with each other. That is, for any user identifier $id$ and any two peers $p$ and $q$ such that $x \in \mathsf{dom}\ db_p \cap \mathsf{dom}\ db_q$ it is the case that $db_p(x).profile = db_j(x).profile$.

Formally, the property is formulated as:

**Proof Obligation 36** $C_1 \stackrel{def}{=} \forall p, q \in [1,\ len\ db],\ x \in dom\ db_p \cap dom\ db_q \cdot db_p(x).profile = db_j(x).profile.$

$C_1$ is an invariant that must also be ensured before execution of methods, which means that the pre-conditions of methods must be strengthened with this assertion resulting in:

$$\begin{array}{l}\textbf{muProfile}_\textbf{j}(\textbf{id} : \textbf{UserID}, \textbf{pr} : \textbf{Profile}) \triangleq \\ \quad \mathsf{pre}\ \ \textbf{pre-}simpleMu_i \wedge \textbf{pre-}simpleMu_j \wedge C_1 \\ \quad \mathsf{post}\ \textbf{post-}simpleMu_i \mid \textbf{post-}simpleMu_j\end{array}$$

Despite this refinement, any attempt to discharge $C_1$ fails. To see why, let us assume that the peers 124 and 125 have each an entry corresponding to the user identifier *id* in their respective local repositories. We also assume that the peer 124 is subscribed to updates concerning the user *id* while the peer 125 is not. If a peer 123 now updates the profile of the user *id* (and subsequently announces an event), the peer 124 will receive the event while the peer 125 will not, leading to an inconsistency between the peers 124 and 125. To avoid such situations, we need to add an invariant to the repositories. We require that if there is an entry with identifier *id* in the local repository $db_i$ of peer $i$, then, this peer must be subscribed to updates related to this identifier. The invariant is formulated as:

$$\begin{array}{lll} inv4 & \stackrel{def}{=} & \forall\, i \in [1, \mathsf{len}\ db],\ id : UserID,\ e : Event \cdot \\ & & (id \in \mathsf{dom}\ db_i \ \wedge\ id = e.payload.name) \ \Rightarrow\ e \in \mathcal{B}_1(\langle ebsimpleMu_i \rangle) \end{array}$$

We further strengthen the pre-condition of the above specification (by the consequence rule) and add more information in the post-condition by the post-rule:

$$\begin{array}{l}\textbf{muProfile}_\textbf{j}(\textbf{id} : \textbf{UserID},\ \textbf{pr} : \textbf{Profile}) \triangleq \\ \quad \mathsf{wr}\quad db \\ \quad \mathsf{pre}\quad \textbf{pre-}simpleMu_i \wedge \textbf{pre-}simpleMu_j \wedge C_1 \wedge \mathsf{inv}\text{-}System \\ \quad \mathsf{post}\quad \textbf{post-}simpleMu_i \mid \textbf{post-}simpleMu_j \wedge \overleftarrow{\textbf{pre-}muProfile_j}\end{array}$$

We discharge this proof obligation by distinguishing five main cases:

- p=q; this case is trivial as for any $x \in \mathsf{dom}\ db_p$, $db_p(x) = db_q(x)$.
- $p \neq q$, $p = j$, $q = i$, x=id. In this case, $db_i(id) = db_j(id)$ results from the application of $simpleMu_j(id, pr)$ followed by $ebsimpleMu_i(v_1)$.
- $p \neq q$, $p = j$, $q = i$, $x \neq id$; $db_p(x) = db_q(id)$ results from that the application of $simpleMu_j(id, pr)$ followed by $ebsimpleMu_i(v_1)$ keeps any entry other than *id* unchanged. And, by the validity of $C_1$ in the initial state, the result is ensured.
- $p \neq q$, $p = j$, $q = i$, $x \neq id$, the argumentation is the same as above.

- $p \neq q$, $p = j$, $q = i$, $x = id$ is rendered impossible by the above invariant.

**from** $\mathbf{post}\text{-}muProfile_j(id, pr) \wedge x \in \mathsf{dom}\ db_p \cap \mathsf{dom}\ db_q$
$\mathbf{post}\text{-}simpleMu_j(id, pr) \mid \mathbf{post}\text{-}ebsimpleMu_i$
$\overleftarrow{C_1} \wedge \overleftarrow{\mathbf{pre}\text{-}simpleMu_j}(id, pr) \wedge \overleftarrow{\mathbf{pre}\text{-}ebsimpleMu_i}(id, pr)$

**from** $\overleftarrow{\mathbf{pre}\text{-}simpleMu_j}(id, pr) \wedge \overleftarrow{\mathbf{pre}\text{-}ebsimpleMu_i}(id, pr)$
**infer** $id \in \mathsf{dom}\ \overleftarrow{db_i} \cap \mathsf{dom}\ \overleftarrow{db_j}$

**from** $p = q$
**infer** $db_p(x).profile = db_q(x).profile$

**from** $p \neq q \wedge p = j \wedge q = i \wedge x = id \wedge \mathbf{post}\text{-}simpleMu_j(id, pr) \mid \mathbf{post}\text{-}ebsimpleMu_i(id, pr)$
  **from** $\mathbf{post}\text{-}simpleMu_j(id, pr)$
  **infer** $db_j(id).profile = pr$
  **from** $\mathbf{post}\text{-}ebsimpleMu_i(id, pr)$
  **infer** $db_j = \overleftarrow{db_j} \wedge db_i(id).profile = pr$
$db_j(id).profile = pr \mid (db_j = \overleftarrow{db_j} \wedge db_i(id).profile = pr)$
$db_i(id).profile = db_j(id).profile$
**infer** $db_p(x).profile = db_q(x).profile$

**from** $p \neq q \wedge p = j \wedge q = i \wedge x \neq id \wedge$ **post**-$simpleMu_j(id, pr)$ | **post**-$ebsimpleMu_i(id, pr)$
  **from** $\overleftarrow{C_1}$
  **infer** $\overleftarrow{db_j}(x).profile = \overleftarrow{db_i}(x).profile$
  **from** **post**-$simpleMu_j(id, pr) \wedge x \neq id$
  **infer** $\overleftarrow{db_j}(x).profile = db_j(x).profile$
  **from** **post**-$ebsimpleMu_i(id, pr) \wedge x \neq id$
  **infer** $\overleftarrow{db_i}(x).profile = db_i(x).profile \wedge \overleftarrow{db_j} = db_j$
 $db_j(x).profile = db_i(x).profile$
**infer** $db_p(x).profile = db_q(x).profile$

**from** $p \neq q \wedge p = j \wedge q \neq i \wedge$ **post**-$simpleMu_j(id, pr)$ | **post**-$ebsimpleMu_i(id, pr)$
  **from** **post**-$simpleMu_j(id, pr)$
  **infer** $v_1.payload.name = id \wedge v_1.action = \langle UserProfileUpdate \rangle$
  **from** $\mathcal{B}_1 \wedge v_1.payload.name = id \wedge v_1.action = \langle UserProfileUpdate \rangle \wedge \overleftarrow{inv_4}$
   $subscribers(v_1) = \{ebsimpleMu_i, mskip\}$
   $ebsimpleMu_q \notin subscribers(v_1)$
   $v_1.payload.name \notin \overleftarrow{db_q}$
  **infer** $id \notin \mathsf{dom}\ \overleftarrow{db_q}$
 $x \in \mathsf{dom}\ \overleftarrow{db_q} \wedge id \notin \mathsf{dom}\ \overleftarrow{db_q}$
 $x \neq id$
  **from** **post**-$simpleMu_j(id, pr) \wedge x \neq id$
  **infer** $\overleftarrow{db_j}(x).profile = db_j(x).profile \wedge db_q(x) = \overleftarrow{db_q}(x)$
  **from** **post**-$ebsimpleMu_j(id, pr)$
  **infer** $\overleftarrow{db_j} = db_j \wedge db_q(x) = \overleftarrow{db_q}(x)$
 $db_j(x).profile = db_q(x).profile$
**infer** $db_p(x).profile = db_q(x).profile$
**infer** $C_1$

The property has been shown for the case where *impl-ebsimpleMu*$_i$ is the only operation interested to event named $\langle UserProfileUpdate \rangle$. In this case, there is no concurrency. Let us now assume another operation *impl-ebsimpleMu*$_k$ subscribed to these events, resulting into concurrency.

The binding is now defined as:

$$\begin{aligned}\mathcal{B}_2 = \{\ & impl\text{-}ebsimpleMu_i \mapsto \{x : Event \cdot\ x.action = \langle UserProfileUpdate \rangle \wedge x.peerid \neq i\}, \\ & impl\text{-}ebsimpleMu_j \mapsto \{\}, \\ & impl\text{-}ebsimpleMu_k \mapsto \{x : Event \cdot\ x.action = \langle UserProfileUpdate \rangle \wedge x.peerid \neq k\}, \\ & \textbf{mskip} \mapsto \{x : Event\}\}\end{aligned}$$

which results in the following definition of *subscribers*:

$$
\begin{array}{l}
\forall e : Event \cdot \\
e.action = \langle UserProfileUpdate \rangle \wedge e.peerid = i \;\Rightarrow\; subscribers(e) = \{impl\text{-}ebsimpleMu_k, mskip\} \\
e.action = \langle UserProfileUpdate \rangle \wedge e.peerid = k \;\Rightarrow\; subscribers(e) = \{impl\text{-}ebsimpleMu_i, mskip\} \\
e.action = \langle UserProfileUpdate \rangle \wedge e.peerid \notin [i,k] \;\Rightarrow \\
\qquad\qquad subscribers(e) = \{impl\text{-}ebsimpleMu_k, impl\text{-}ebsimpleMu_i, mskip\}
\end{array}
$$

Since, however, any event announced by $muProfile_j$ is such that $e.peerid = j \notin [i,k]$, the following specification is derived by application of the announce rule followed by the skip rule.

**behavioral-muProfile$_j$(id : UserID, prof : Profile)** $\triangleq$
**await true do** $simpleMu_j(id,\ prof)$ **od**;
$\{ebsimpleMu_i(v) \| ebsimpleMu_k(v)\}$

We need to find $A$ and $B$ such that:

- $A_i \mid post\text{-}ebsimpleMu_k(v) \;\Rightarrow\; A_i$ ,
- $A_k \mid post\text{-}ebsimpleMu_i(v) \;\Rightarrow\; A_k$,
- $post\text{-}ebsimpleMu_i(v) \;\Rightarrow\; A_i$,
- $post\text{-}ebsimpleMu_k(v) \;\Rightarrow\; A_k$,
- $post\text{-}ebsimpleMu_j(v) \mid (A_i \wedge A_k) \;\Rightarrow\; C_1$.

We choose $A_i$ and $A_k$ defined as:

- $A_i \stackrel{def}{=} db_i(id) = pr \wedge db_i \lhd \{id\} = \overleftarrow{db_i} \lhd \{id\} \wedge \forall t \in [1, \mathsf{len}\ db] \setminus \{i,k\} \cdot db_t = \overleftarrow{db_t}$ ,
- $A_k \stackrel{def}{=} db_k(id) = pr \wedge db_k \lhd \{id\} = \overleftarrow{db_k} \lhd \{id\} \wedge \forall t \in [1, \mathsf{len}\ db] \setminus \{i,k\} \cdot db_t = \overleftarrow{db_t}$ .

A proof by natural deduction borrowing many of the arguments of the previous proof can be constructed to discharge the proof obligation. We omit this proof.

### 14.5.2 Property II: Non-Volatility of User Data

A basic property that can be checked is indeed that the peer $i$ always has the current version of any user profile in the system. This property ensures that user data is not volatile and can always be recovered in case of problem.

This proof obligation is formulated as:

**Proof Obligation 37** $C_2 \stackrel{def}{=} \forall t \in [1, len\ db],\ x : UserID \cdot db_i(x).profile = db_t(x).profile$

This is a global invariant of the platform that must hold after the execution of each *impl-muProfile*$_j$. Our proof is based on the validity of

$$\textit{post-muProfile}_j(id,\ pr)\ \Rightarrow\ C_1$$

which is discharged above. The proof is by distinguishing two cases:

- $x \neq id$; in this case, the entry $db_t(x)$ is kept unchanged for any $t$ and from the validity of $C_2$ in the initial state, one derives that $db_i(x) = db_t(x)$ holds.
- $x = id$; the validity of $C_1$ after the execution of *impl-muProfile*$_j$ is applied to infer that $db_i(x).profile = db_t(x).profile$ for any peer $t$ such that $x \in \mathsf{dom}\ db_t$.

**from** **post-***muProfile*$_j(id, pr) \wedge x \in \mathsf{dom}\ \overleftarrow{db_i} \wedge \overleftarrow{C_2}$
$C_1$
**from** $x = id \wedge C_1 \wedge x \in \mathsf{dom}\ \overleftarrow{db_i}$
**from** **post-***muProfile*$_j(id, pr)$
**infer** $x \in \mathsf{dom}\ db_i \wedge db_i(x) = pr$
**from** $C_1$
**infer** $\forall t \in [1, \mathsf{len}\ db] \cdot id \in \mathsf{dom}\ db_t\ \Rightarrow\ db_t(x) = pr$
**infer** $\forall t \in [1, \mathsf{len}\ db] \cdot id \in \mathsf{dom}\ db_t\ \Rightarrow\ db_i(x) = db_t(x)$

**from** $x \neq id \wedge \overleftarrow{C_2} \wedge x \in \mathsf{dom}\ \overleftarrow{db_i} \wedge$ **post-***muProfile*$_j(id, pr)$
**from** **post-***muProfile*$_j(id, pr) \wedge x \neq id$
**infer** $\forall t \in [1, \mathsf{len}\ db] \cdot x \in \mathsf{dom}\ \overleftarrow{db_t}\ \Rightarrow\ \overleftarrow{db_t}(x) = db_t(x)$
$db_i(x) = \overleftarrow{db_i}(x)$
**from** $\overleftarrow{C_2}$
**infer** $\forall t \in [1, \mathsf{len}\ db] \cdot x \in \mathsf{dom}\ \overleftarrow{db_t}\ \Rightarrow\ \overleftarrow{db_i}(x) = \overleftarrow{db_t}(x)$
**infer** $\forall t \in [1, \mathsf{len}\ db] \cdot x \in \mathsf{dom}\ db_t\ \Rightarrow\ db_i(x) = db_t(x)$
**infer** $C_2$

The properties $C_1$ and $C_2$ about the MOTION platform presented in this chapter are examples of those requirements that could not be checked in the original formal design that were oriented towards client/server applications. Discharging the related proof obligations was possible only after some change to the specification of our components. For instance, the mutual exclusion construct was inserted in the specification of *impl-muProfile*$_j$ that was not part of our original specification. As another example, discharging $C_1$ required the formulation of the invariant $inv_4$.

The analysis of the MOTION platform in the LECAP framework is, thus, a successful exercise in that it allows discovering and correcting design flaws of the original proposal.

In addition, this shows that the SEATY style is indeed applicable to non-trivial case studies. In fact, we claim that the MOTION case study and the SMU system are complex case studies in that components can be subscribed and unsubscribed dynamically. We have intentionally chosen such case studies to excercise our approach. The stack-counter example presented in other chapters are typical example of cases with static components.

## 14.6 SUMMARY

The increasing use of the event-based paradigm in applications and systems motivates the need for methodologies to support not only the construction of such systems but also reasoning about their correctness and reliability. Due to the loose coupling of components in such systems, one would expect that both construction and reasoning should be easier to support than in conventional more tightly-coupled systems. Yet our attempt in developing the MOTION (MObile Teamwork Infrastructure for Organizations Networking) platform according to conventional formal methods revealed the lack of any formal support for event-based applications. This experience motivated the development of the LECAP methodology to support the design, construction, and verification of event-based systems.

In this chapter we have reported the results of a case study in applying the LECAP methodology in the redesign of a component of the MOTION platform. We have shown how reasoning and validation about properties of the system can be carried out both about local properties and global properties. Although our experience is promising, the case study also identified some deficiencies that point the way to needed future work.

# CHAPTER 15

# SOUNDNESS

This chapter aims at showing that the proof system presented in this thesis is sound. That is, if we derive a formula using the set of rules we presented, then the validity of this formula also holds at the semantics level. More precisely, if we deduce that the program $z$ satisfies the specification $\vartheta ::S$ (respectively $(\vartheta, \mathcal{B}) ::S$) in the system, then this is also true semantically.

The proof is by showing that each rule of the system is sound; by which is meant that if the premises of the rule hold, so does its conclusion. The soundness of the proof system follows by induction on the number of times that these rules are applied for the derivation of judgments. Our proof system is based on rely/guarantee reasoning; it is therefore not surprising that the soundness proofs of LECAP rules are similar to that of rely/guarantee rules. We show the similarity between these proofs by providing the proofs for some of the rules in the LECAP framework. The announce rule that is typical to event announcement is proven from scratch.

The chapter is organized in the following manner. To proof the soundness of the parallel rules, it is necessary to be able to decompose and compose computations. The next section shows how this is done. In Section 15.2, the set of rules for the decomposition and for the top-down development of components are proven sound. The results of this section are adapted in Section 15.3 to the soundness of the rules for the composition of specifications and for the manipulation of resulting behavioral specifications. The SEATY framework is proven sound in Section 15.4 by transforming its formulas into pure LECAP formulas and applying the related soundness results. Finally, 15.5 concludes the chapter.

## 15.1 COMPOSITION OF COMPUTATIONS

We illustrate how to decompose a computation of a program into computations of its subprograms and how to compose a computation starting with the computations of two subprograms.

**Definition 50** *The computations* $\sigma^1 \in cp[z_1]$ *and* $\sigma^2 \in cp[z_2]$ *compose the computation* $\sigma \in cp[z_1 \| z_2]$ *(denoted* $\sigma \propto \sigma^1 \| \sigma^2$*) iff:*

- $len\ (\sigma^1) = len\ (\sigma^2) = len\ (\sigma)$,
- $S(\sigma^1) = S(\sigma^2) = S(\sigma)$,
- *for any* $1 \leq i \leq len\ (\sigma)$, *one of the following hold:*
  - $L(\sigma_i^1) = v,\ L(\sigma_i^2) = v,\ L(\sigma_i) = v$,
  - $L(\sigma_i^1) = i,\ L(\sigma_i^2) = v,\ L(\sigma_i) = i$,
  - $L(\sigma_i^1) = v,\ L(\sigma_i^2) = i,\ L(\sigma_i) = i$.
- *for any* $1 \leq i \leq len\ (\sigma)$, *one of the following hold:*
  - $Z(\sigma_i^1) = \epsilon,\ Z(\sigma_i^2) = \epsilon,\ Z(\sigma_i) = \epsilon$,
  - $Z(\sigma_i^1) = \epsilon,\ Z(\sigma_i^2) \neq \epsilon,\ Z(\sigma_i) = Z(\sigma_i^2)$,
  - $Z(\sigma_i^1) \neq \epsilon,\ Z(\sigma_i^2) = \epsilon,\ Z(\sigma_i) = Z(\sigma_i^1)$,
  - $Z(\sigma_i^1) \neq \epsilon,\ Z(\sigma_i^2) \neq \epsilon,\ Z(\sigma_i) = \{Z(\sigma_i^1) \| Z(\sigma_i^2)\}$.

The intuition behind this decomposition of a computation $\sigma$ of $\{z_1 \| z_2\}$ is that, any of its transitions is either a transition of the environment or a transition of $z_1$ or $z_2$. If a transition is a program transition of $z_1$ or of $z_2$, then it is a program transition of their composition. Further, the decomposition of a computation into two computations is possible iff the two subcomputations have the same length and the same sequence of states.

**Lemma 1** *It results from the semantics of the parallel composition that*

$$cp[z_1 \| z_2] = \{\sigma \mid \textit{there exists } \sigma^1 \in cp[z_1],\ \sigma^2 \in cp[z_2],\ \textit{such that } \sigma^1 \textit{ and } \sigma^2 \textit{ compose } \sigma\}.$$

## 15.2 Structural Rules

### 15.2.1 Basic Parallel Rule

We discuss the soundness of the basic parallel rule. By assumption, the following formulas hold:

$H_1$: $\models_\pi G_1 \Rightarrow R_2$
$H_2$: $\models_\pi G_2 \Rightarrow R_1$
$H_3$: $\models_\pi z_1 \ \underline{sat}\ \vartheta ::(P,\ R_1,\ G_1,\ E_1)$
$H_4$: $\models_\pi z_2 \ \underline{sat}\ \vartheta ::(P,\ R_2,\ G_2,\ E_2)$

we need to deduce that

$$\models_\pi \{z_1 \| z_2\} \ \underline{sat}\ \vartheta ::(P,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ E_1 \wedge E_2)$$

also holds.

**Lemma 2** *For any computations $\sigma \in cp[z_1 \| z_2] \cap ext[\vartheta, P, R_1 \wedge R_2]$, $\sigma^1 \in ext[\vartheta, P, R_1]$, and $\sigma^2 \in ext[\vartheta, P, R_2]$, such that $\sigma \propto \sigma^1 \| \sigma^2$, any program transition of $\sigma^1$ satisfies $G_1$ and any program transition of $\sigma^2$ satisfies $G_2$.*

**Proof.**
Take any finite subcomputation $\sigma_1[1, \cdots, k]$ of $\sigma_1$ that starts with the same first configuration as $\sigma_1$. This subcompuation is in $ext[\vartheta, P, R_1]$; and, since it is finite, any of its program transitions satisfies $G_1$ (by Hypothesis $H_3$). Therefore, any program transition of $\sigma^1$ satisfies $G_1$.

**Lemma 3** *Any computation $\sigma \in cp[z_1 \| z_2] \cap ext[\vartheta, P, R_1 \wedge R_2]$, is such that any of its program transitions satisfies $G_1 \vee G_2$.*

**Proof.**
From Lemma 1, there exists $\sigma^1 \in cp[z_1]$ and $\sigma^2 \in cp[z_2]$ such that $\sigma \propto \sigma^1 \| \sigma^2$. From Lemma 2, any program transition of $\sigma^1$ satisfies $G_1$ and any program transition of $\sigma^2$ satisfies $G_2$. And, by Definition 50, any program transition of $\sigma$ is either a program transition of $\sigma^1$ or of $\sigma^2$, hence satisfies $G_1 \vee G_2$.

**Lemma 4** *For any computations $\sigma \in cp[z_1 \| z_2] \cap ext[\vartheta, P, R_1 \wedge R_2]$, $\sigma^1 \in ext[\vartheta, P, R_1]$, and $\sigma^2 \in ext[\vartheta, P, R_2]$, such that $\sigma \propto \sigma^1 \| \sigma^2$, any environment transition of $\sigma^1$ satisfies $R_1$ and any program transition of $\sigma^2$ satisfies $R_2$.*

**Proof.**
Any environment transition of $\sigma^1$ is either a program transition of $\sigma^2$ (in which case it satisfies $G_2$, and from $H_2$, it also satisfies $R_1$), or it is an environment transition in $\sigma$, in which case it satisfies $R_1 \wedge R_2$.

**Lemma 5** *For any computation* $\sigma \in cp[z_1 \| z_2] \cap ext[\vartheta, P, R_1 \wedge R_2]$ *such that* $len\ (\sigma) < \infty$, *and* $\Sigma(\sigma_{len\ (\sigma)}) = \epsilon$, *the formula* $(S(\sigma_1), S(\sigma_{len\ (\sigma)})) \models_\pi E_1 \wedge E_2$ *holds.*

**Proof.**
From Lemma 1, there exists $\sigma^1 \in cp[z_1]$ and $\sigma^2 \in cp[z_2]$ such that $\sigma \propto \sigma^1 \| \sigma^2$. From Lemma 5 , it follows that $\sigma^1 \in ext[\vartheta, P, R_1]$ and $\sigma^2 \in ext[\vartheta, P, R_2]$. By construction of $\sigma^1$ and $\sigma^2$ (Definition 50) it follows from $\mathsf{len}\ (\sigma) < \infty$ and $\Sigma(\sigma_{\mathsf{len}\ (\sigma)}) = \epsilon$ that $\mathsf{len}\ (\sigma^1) < \infty$, $\mathsf{len}\ (\sigma^2) < \infty$, $\Sigma(\sigma^1_{\mathsf{len}\ (\sigma^1)}) = \epsilon$, and $\Sigma(\sigma^2_{\mathsf{len}\ (\sigma^2)}) = \epsilon$ hold. This, clearly results in $\sigma^1 \in int[\vartheta, G_1, E_1]$ and $\sigma^2 \in ext[\vartheta, G_2, E_2]$, and, hence, $(S(\sigma_1), S(\sigma_{\mathsf{len}\ (\sigma)})) \models_\pi E_1 \wedge E_2$ holds.

The validity of the basic parallel rule is now derived from Lemmas 3 and 5.

### 15.2.2 Composed Parallel Rule

The composed parallel rule is investigated; in particular, we derive it soundness from that of the basic parallel rule. By assumtpion, the following formulas hold:

$$H_1\colon \models_\pi G_1 \Rightarrow R_2$$
$$H_2\colon \models_\pi G_2 \Rightarrow R_1$$
$$H_3\colon \models_\pi z\ \underline{sat}\ \vartheta::(P,\ R_1,\ G_1,\ E_1) \| (P,\ R_2,\ G_2,\ E_2)$$

we need to deduce that

$$\models_\pi z\ \underline{sat}\ \vartheta::(P,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ E_1 \wedge E_2)$$

also holds.

**Proof.**
By Definition 27, there exists $z_1$, $z_2$ such that:

$$H_4\colon \models_\pi z_1\ \underline{sat}\ \vartheta::(P,\ R_1,\ G_1,\ E_1)$$
$$H_5\colon \models_\pi z_2\ \underline{sat}\ \vartheta::(P,\ R_2,\ G_2,\ E_2), \text{ and}$$
$$H_7 : z \textbf{ behaves as } \mathbf{z_1; z_2}.$$

By the soundness of the basic parallel rule, $H_1$, $H_2$, $H_4$, and $H_5$ it however, follows that:

$$\models_\pi z_1; z_2\ \underline{sat}\ \vartheta::(P,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ E_1 \wedge E_2)$$

From $H_7$, it follows that for any binding $\mathcal{B}$,

$$\models_\pi z \ \underline{sat}\ (\vartheta, \mathcal{B}) :: (P,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ E_1 \wedge E_2)$$

holds, hence the validity of

$$\models_\pi z \ \underline{sat}\ \vartheta :: (P,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ E_1 \wedge E_2).$$

### 15.2.3 Basic Consequence Rule

We show that

$$\models_\pi z \ \underline{sat}\ \vartheta :: (P_2,\ R_2,\ G_2,\ E_2).$$

follows from the assumptions:

$H_1$: $\models_\pi P_2 \Rightarrow P_1$

$H_2$: $\models_\pi R_2 \Rightarrow R_1$

$H_3$: $\models_\pi G_1 \Rightarrow G_2$

$H_4$: $\models_\pi E_1 \Rightarrow E_2$

$H_5$: $\models_\pi z \ \underline{sat}\ \vartheta :: (P_2,\ R_2,\ G_2,\ E_2).$

In other terms, we need to deduce that

$$ext[\vartheta, P_2, R_2] \cap cp[z] \subseteq int[\vartheta, G_2, E_2].$$

For this, we show the following inclusions from which the result follows.

$$ext[\vartheta, P_2, R_2] \subseteq ext[\vartheta, P_1, R_1]$$

$$int[\vartheta, G_1, E_1] \subseteq int[\vartheta, G_2, E_2].$$

Let us assume a computation $\sigma \in ext[\vartheta, P_2, R_2]$. Its initial $S(\sigma_1)$ is such that $S(\sigma_1) \models_\pi P_2$ holds. From $H_1$, it follows that $S(\sigma_1) \models_\pi P_1$ also holds. On the other hand, if $\langle z_k, s_k \rangle \xrightarrow{v} \langle z_k, s_{k+1} \rangle$ is an environment transition of $\sigma$ then $(s_k, s_{k+1}) \models_\pi R_2$ holds. And from $H_2$ it follows $(s_k, s_{k+1}) \models_\pi R_1$ also holds, hence the first inclusion.

Next, we consider $\sigma \in int[\vartheta, G_1, E_1]$. Any of its program transitions satisfies $G_1$ and from $H_3$, it also satisfies $G_2$. If $\sigma$ is a finite computation, then its final state satisfies $E_1$ which, however, results in $E_2$; hence the soundness of the rule.

### 15.2.4 Composed Consequence Rule

We divide the consequence rule in two cases. The first case consists in replacing $\diamond$ with the sequential composition operator ; while the second case consists in the replacement of the generic operator $\diamond$ with the parallel composition operator.

**Sequential Consequence Rule.**

We assume that the formulas $H_1$ and $H_2$ hold and deduce that $H_3$ holds.

$$H_1\colon \models_\pi t \ \underline{sat}\ \vartheta ::S \ \Rightarrow\ t \ \underline{sat}\ \vartheta ::S'$$
$$H_2\colon \models_\pi z \ \underline{sat}\ \vartheta ::S_1; S; S_2$$
$$H_3\colon \models_\pi z \ \underline{sat}\ \vartheta ::S_1; S'; S_2.$$

By the definition of structural specifications, there exists three programs $z_1$, $y$, and $z_2$ such that

$$H_4\colon \models_\pi z_1 \ \underline{sat}\ \vartheta ::S_1;$$
$$H_5\colon \models_\pi z_2 \ \underline{sat}\ \vartheta ::S_2$$
$$H_6\colon \models_\pi y \ \underline{sat}\ \vartheta ::S$$
$$H_7:\ z \textbf{ behaves as } z_1; y; z_2$$

The vailidy of $H_3$ follows by observing that $z_1$, $y$, and $z_2$ are such that $\models_\pi z_1 \ \underline{sat}\ \vartheta ::S_1$ holds ($H_4$), $\models_\pi z_2 \ \underline{sat}\ \vartheta ::S_2$ holds ($H_5$), and $\models_\pi y \ \underline{sat}\ \vartheta ::S'$ holds ($H_6$ and $H_1$). By $H_7$, $z$ satisfies any specification $\vartheta ::(P, R, G, E)$ that $z_1; y; z_2$ satisfies.

**Parallel Consequence Rule.**

We assume that the formulas $H_1$ and $H_2$ hold and deduce that $H_3$ holds.

$$H_1\colon \models_\pi t \ \underline{sat}\ \vartheta ::S \ \Rightarrow\ t \ \underline{sat}\ \vartheta ::S'$$
$$H_2\colon \models_\pi z \ \underline{sat}\ \vartheta ::\{S_1 \| S \| S_2\}$$
$$H_3\colon \models_\pi z \ \underline{sat}\ \vartheta ::\{S_1 \| S' \| S_2\}.$$

By the definition of structural specifications, there exists three programs $z_1$, $y$, and $z_2$ such that

$H_4$: $\models_\pi z_1$ *sat* $\vartheta::S_1$;

$H_5$: $\models_\pi z_2$ *sat* $\vartheta::S_2$

$H_6$: $\models_\pi y$ *sat* $\vartheta::S$

$H_7$: $z$ **behaves as** $\{z_1 \| y \| z_2\}$

The vailidy of $H_3$ follows by observing that $z_1$, $y$, and $z_2$ are such that $\models_\pi z_1$ *sat* $\vartheta::S_1$ holds ($H_4$), $\models_\pi z_2$ *sat* $\vartheta::S_2$ holds ($H_5$), and $\models_\pi y$ *sat* $\vartheta::S'$ holds ($H_6$ and $H_1$). By $H_7$, $z$ satisfies any specification $\vartheta::(P, R, G, E)$ that $\{z_1 \| y \| z_2\}$ satisfies.

### 15.2.5 Basic Sequential Rule

Suppose the premises of the basic sequential rule hold, namely:

$H_1$: $\models_\pi z_1$ *sat* $\vartheta::(P,\ R,\ G,\ E_1 \wedge P_2)$

$H_2$: $\models_\pi z_2$ *sat* $\vartheta::(P,\ R,\ G,\ E_2)$

Since $events(z_1) = \{\}$, any computation $\sigma$ of $z_1; z_2$ is of one of the two forms:

- $z_1$ blocks before $z_2$ is executed, that is $\sigma = \langle z_1; z_2,\ s_1\rangle \xrightarrow{l_1} \cdots \xrightarrow{l_n} \langle z_1^n; z_2, s_n\rangle$
- $z_1$ terminates, but $z_2$ blocks or deadlocks, that is: $\sigma = \langle z_1; z_2,\ s_1\rangle \xrightarrow{l_1} \cdots \xrightarrow{l_n} \langle z_2^n, s_n\rangle$ where $\langle z_1,\ s_1\rangle \xrightarrow{l_1} \cdots \xrightarrow{l_k} \langle \epsilon, s_{k+1}\rangle$ and $\langle z_2,\ s_{k+1}\rangle \xrightarrow{l_{k+1}} \cdots \xrightarrow{l_n} \langle z_2^n, s_n\rangle$ holds and $z_2^n$ is possibly $\epsilon$.

The computations $\langle z_1,\ s_1\rangle \xrightarrow{l_1} \cdots \xrightarrow{l_k} \langle \epsilon, s_{k+1}\rangle$ and $\langle z_2,\ s_{k+1}\rangle \xrightarrow{l_{k+1}} \cdots \xrightarrow{l_n} \langle z_2^n, s_n\rangle$ are called $\sigma'$ and $\sigma''$ respectively.

Suppose $\sigma \in ext[\vartheta, P, R] \cap cp[z_1; z_2]$; if $z_1; z_2$ deadlocks then, $\sigma \in int[\vartheta, G, E_1 \mid E_2]$. If, however, $\sigma$ terminates, then, since $\sigma'' \in int[\vartheta, G, E_2]$ and $\sigma' \in int[\vartheta, G, E_1]$, it results that $\models_\pi (s_1, s_{k+1})$ *sat* $E_1$ and $\models_\pi (s_{k+1}, s_n)$ *sat* $E_2$, hence $\models_\pi (s_1, s_n)$ *sat* $E_1 \mid E_2$ holds. Further, since any transition of $\sigma$ is either a transition of $\sigma'$ or of $\sigma''$, it results that any program transition in $\sigma$ satisfies $G$, hence the rule.

### 15.2.6 Composed Sequential Rule

Assume:

$$H_1 : \ events(z) = \{\}$$

$$H_2 : \models_\pi z \ \underline{sat} \ \vartheta ::(P,\ R,\ G,\ E_1 \wedge P_2);(P,\ R,\ G,\ E_2)$$

By Definition 32, there exists $z_1$ and $z_2$ such that:

$$H_3 : \models_\pi z_1 \ \underline{sat} \ \vartheta ::(P,\ R,\ G,\ E_1 \wedge P_2)$$

$$H_4 : \models_\pi z_2 \ \underline{sat} \ \vartheta ::(P,\ R,\ G,\ E_2)$$

$z$ **behaves as** $z_1; z_2$.

By the basic sequential rule, it follows that:

$$H_5 : \models_\pi z_1; z_2 \ \underline{sat} \ \vartheta ::(P,\ R,\ G,\ E_1 \mid E_2)$$

And for any binding $\mathcal{B}$,

$$H_6 : \models_\pi z \ \underline{sat} \ (\vartheta, \mathcal{B}) ::(P,\ R,\ G,\ E_1 \mid E_2)$$

Combining this with $H_1$, it follows that:

$$H_6 : \models_\pi z \ \underline{sat} \ \vartheta ::(P,\ R,\ G,\ E_1 \mid E_2)$$

### 15.2.7 Basic Conditional Rule

The assumptions of this rule are:

$$H_2 : \models_\pi z_1 \ \underline{sat} \ \vartheta ::(P \wedge b,\ R,\ G,\ E)$$

$$H_3 : \models_\pi z_2 \ \underline{sat} \ \vartheta ::(P \wedge \neg b,\ R,\ G,\ E)$$

Any computation of $\sigma \in ext[\textbf{if } b \textbf{ then } z_1 \textbf{ else } z_2 \textbf{ fi}, P, R]$ is of the form:

$$\sigma = \langle z,\ s_1 \rangle \xrightarrow{v} \cdots \xrightarrow{v} \langle z, s_n \rangle \xrightarrow{i} \sigma'$$

where

$$\sigma' \in ext[\vartheta, P \wedge b, R] \cap cp[z_1] \ \cup \ ext[\vartheta, P \wedge \neg b, R] \cap cp[z_2]$$

Since, however, $P \wedge b$ and $P \wedge \neg b$ are stable when $R$,

$$\sigma \in ext[\vartheta, P \wedge b, R] \cap cp[z_1] \ \cup \ ext[\vartheta, P \wedge \neg b, R] \cap cp[z_2]$$

also holds.

And, by the assumptions $H_2$ and $H_3$, it results that $\sigma \in int[\vartheta, G, E]$ holds.

### 15.2.8 Pre Rule

The premises of this rule is formulated as:

$$H_1 : \ z \ \underline{sat} \ \vartheta : : (P, \ R, \ G, \ E)$$

By definition, $int[\vartheta, G, \overleftarrow{P} \wedge E] = \{\sigma \in int[\vartheta, G, E] \cdot S(\sigma_1) \models_\pi P\}$

However, for any $\sigma \in int[\vartheta, G, E] \cap ext[\vartheta, P, R]$, $S(\sigma_1) \models_\pi P$, hence, $\sigma \in int[\vartheta, G, \overleftarrow{P} \wedge E]$.

### 15.2.9 Post Rule

Assume a program $z$ such that:

$$H_1 : \ z \ \underline{sat} \ \vartheta : : (P, \ R, \ G, \ E)$$

and a computation $\sigma \in ext[\vartheta, P, R] \cap cp[z]$. Any transition in $\sigma$ is either a environment transition or a program transition. In the first case, it satisfies $R$ while in the second case it ensures $G$ if a state variable is changed. Any transition in $\sigma$, therefore satisfies $R \vee G \vee I_\vartheta$. And, because the final state is reached by a finite number of environment and program transitions, it satisfies not only $E$, but also $(R \vee G \vee I_\vartheta)^*$ which is equivalent to $(G \vee R)^+$. The computation $\sigma$ is, therefore, also in $int[\vartheta, G, (R \vee G)^+ \wedge E]$.

### 15.2.10 Assignment Rule

Given the validity of the formulas:

$$H_1 : P \mid R \Rightarrow P$$
$$H_2 : E \mid R \Rightarrow E$$
$$H_3 : \overleftarrow{P} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \{v\}} \Rightarrow (G \vee I_\vartheta) \wedge E$$

we intend to derive the validity of the formula:

$$v := r \ \underline{sat} \ \vartheta : : (P, R, G, E).$$

Any computation $\sigma \in ext[\vartheta, P, R] \cap cp[v := r]$ of the program $v := r$ is of the form:

$$\sigma = \langle v := r,\ s_1 \rangle \xrightarrow{v} \cdots \xrightarrow{v} \langle v := r, s_k \rangle \xrightarrow{i} \langle \epsilon,\ s_{k+1} \rangle \xrightarrow{v} \cdots \xrightarrow{v} \langle \epsilon, s_{k+n} \rangle$$

The environment may perform a finite number of transitions before the execution of the program $v := r$ which consists in a unique program transition. After this program transition, the environment may also perform a finite number of transitions.

If the program $v := r$ is started in a state satisfying $P$, thanks to the stability of $P$ with respect to $R$ ($H_1$), the program $v := r$ is indeed executed in a state satisfying $P$. After the internal transition, the state $s_{k+1}$ is such that $\models_\pi \overleftarrow{P} \wedge v = \overleftarrow{r} \wedge I_{\vartheta \setminus \{v\}}$ holds. That is, it is such that the assertion $P$ held in the previous state (which is true, $P$ holds in $s_k$), and $v$ now has the value that the expression $r$ had in the previous state (which is also ensured by the semantics of the assignment construct), and any other variable is unchanged (also ensured by the semantics of the assignment construct).

By hypothesis $H_3$, it results that $(G \vee I_\vartheta) \wedge E$ holds which means that $\sigma[1, \cdots, k+1]$, the subcomputation starting with the first configuration $\sigma_1$ and ending with the configuration $\sigma_{k+1}$ is in $int[\vartheta, G, E]$. Thanks to the stability of $E$ with respect to $R$, any state after $s_{k+1}$ also satisfies $E$, and, hence $\sigma \in int[\vartheta, G, E]$ also holds.

### 15.2.11 Global Rule

Let us assume a program $z$, a set of variable $\vartheta$, and a variable $v$ such that:

$H_1$: $\models_\pi z \ \underline{sat} \ \vartheta \setminus \{v\} ::(P,\ R,\ G,\ E)$

Since the program $z$ accesses only variables in $\vartheta \setminus \{v\}$, for any computation $\sigma \in ext[\vartheta \setminus \{v\}, P, R] \cap cp[z]$, any of its program transition satisfies $v = \overleftarrow{v}$. And, since $\sigma \in int[\vartheta \setminus \{v\}, G, E]$, $\sigma$ is also in $int[\vartheta \cup \{v\}, G \wedge v = \overleftarrow{v}, E]$.

## 15.3 Behavioral Rules

We show that our rules on the manipulation and the derivation of behavioral specifications are sound. Some of these rules are obtained from their structural counterparts by simply replacing $\vartheta$ with an event-based system. For these rules, it is clear that the proofs are essentially based on the same arguments. These rules are the pre-rule, the post-rule, the iteration rule (under the assumption that no event is announced within the body of while constructs), the consequence rule, and the parallel rule. To support our argument, we proof the soundness of the behavioral consequence rule below. On the other hand, although the behavioral conditional rule and the behavioral sequential rule are different from their structural counterparts, their proofs are also similar to that of their counterparts. The third class of behavioral rules includes the announce rule which is typical to behavioral specifications.

### 15.3.1 Behavioral Consequence Rule

We show the soundness of the behavioral consequence rule which is obtained from the structural version by a replacement of the set of variables $\vartheta$ with the event-based system $(\vartheta, \mathcal{B})$. Although the proof is based on the same arguments as the proof of the counterpart rule, the two rules have two distinct roles.

A structural judgment of the form $z \ \underline{sat} \ \vartheta ::(P_2,\ R_2,\ G_2,\ E_2)$ means that the program $z$ satisfies the given specification in the event-based system with the empty binding and announces no event while a specification of the form $z \ \underline{sat} \ (\vartheta, \mathcal{B}) ::(P_2,\ R_2,\ G_2,\ E_2)$ means that the program $z$ satisfies the given specification in the event-based system $(\vartheta, \mathcal{B})$. In particular, in the later formulation the program $z$ is allowed to announce an event. The given specification characterizes its behavior and that of all its successors, that is, all programs that are eventually triggered following the announcement of an event by $z$.

We assume that the following formulas hold:

$H_1$: $\models_\pi P_2 \Rightarrow P_1$

$H_2$: $\models_\pi R_2 \Rightarrow R_1$

$H_3$: $\models_\pi G_1 \Rightarrow G_2$

$H_4$: $\models_\pi E_1 \Rightarrow E_2$

$H_5$: $\models_\pi \underline{sat}\ (\vartheta, \mathcal{B})::(P_2,\ R_2,\ G_2,\ E_2)$.

We need to show that:

$$\models_\pi z\ \underline{sat}\ (\vartheta, \mathcal{B})::(P_2,\ R_2,\ G_2,\ E_2).$$

In other terms, we need to deduce that

$$ext[(\vartheta, \mathcal{B}), P_2, R_2] \cap cp[z] \subseteq int[(\vartheta, \mathcal{B}), G_2, E_2].$$

For this, we show the following inclusions.

$$ext[(\vartheta, \mathcal{B}), P_2, R_2] \subseteq ext[(\vartheta, \mathcal{B}), P_1, R_1]$$

$$int[(\vartheta, \mathcal{B}), G_1, E_1] \subseteq int[(\vartheta, \mathcal{B}), G_2, E_2].$$

Let us assume a computation $\sigma \in ext[(\vartheta, \mathcal{B}), P_2, R_2]$. Its initial $S(\sigma_1)$ is such that $S(\sigma_1) \models_\pi P_2$ holds. From $H_1$, it follows that $S(\sigma_1) \models_\pi P_1$ also holds. On the other hand, if $\langle z_k, s_k \rangle \xrightarrow{v} \langle z_k, s_{k+1} \rangle$ is an environment transition of $\sigma$ then $(s_k, s_{k+1}) \models_\pi R_2$ holds. And from $H_2$ it follows $(s_k, s_{k+1}) \models_\pi R_1$ also holds, hence the first inclusion.

Next, we consider $\sigma \in int[(\vartheta, \mathcal{B}), G_1, E_1]$. Any of its program transition satisfies $G_1$ and from $H_3$, they also satisfy $G_2$. If $\sigma$ is a finite computation, then its final state satisfies $E_1$ which, however, results in $E_2$; hence the soundness of the rule.

### 15.3.2 Conditional Rule

We show the soundness of the conditional rule when the binding is defined. Although a sound conditional rule could be derived as for the consequence rule in which no event announcement is allowed, the proposed behavioral conditional rule is superior in that it supports event announcement. Let us assume that the following formulas hold:

$$H_1\colon\ \models_\pi z_1; z\ \underline{sat}\ (\vartheta, \mathcal{B})::(P \wedge b,\ R,\ G,\ E)$$

$$H_2\colon\ \models_\pi z_2; z\ \underline{sat}\ (\vartheta, \mathcal{B})::(P \wedge \neg b,\ R,\ G,\ E).$$

We need to deduce that

$$\models_\pi \textbf{if } b \textbf{ then } z_1 \textbf{ else } z_2 \textbf{ fi}; z\ \underline{sat}\ (\vartheta, \mathcal{B})::(P,\ R,\ G,\ E)$$

Let us assume a computation $\sigma \in ext[(\vartheta, \mathcal{B}), P, R] \cap cp[\textbf{if } b \textbf{ then } z_1 \textbf{ else } z_2 \textbf{ fi}; z]$. This computation is of the form:

$$\sigma = \langle z,\ s_1\rangle \xrightarrow{v} \cdots \xrightarrow{v} \langle z, s_n\rangle \xrightarrow{i} \sigma'$$

where

$$\sigma' \in ext[\vartheta, P \wedge b, R] \cap cp[z_1; z]\ \cup\ ext[\vartheta, P \wedge \neg b, R] \cap cp[z_2; z]$$

Since, however, $P \wedge b$ and $P \wedge \neg b$ are stable when $R$,

$$\sigma \in ext[\vartheta, P \wedge b, R] \cap cp[z_1; z]\ \cup\ ext[\vartheta, P \wedge \neg b, R] \cap cp[z_2; z]$$

also holds.

And, by the assumptions $H_2$ and $H_3$, it results that $\sigma \in int[\vartheta, G, E]$ holds.

### 15.3.3 Sequential Rule

The proof of the soundness of the behavioral sequential rule is also similar to that of the sequential counterpart. This is due to the requirement that the first program announces no event. In effect, considering the assumption:

$$H_1\colon\ \models_\pi z_1\ \underline{sat}\ \vartheta::(P,\ R,\ G,\ E_1 \wedge P_2)$$
$$H_2\colon\ \models_\pi z_2\ \underline{sat}\ (\vartheta, \mathcal{B})::(P,\ R,\ G,\ E_2)$$

The requirement $events(z_1) = \{\}$ still holds and any computation $\sigma$ of $z_1; z_2$ remains of one of the two forms:

- $z_1$ blocks before $z_2$ is executed, that is $\sigma = \langle z_1; z_2,\ s_1\rangle \xrightarrow{l_i} \cdots \xrightarrow{l_n} \langle z_1^n; z_2, s_n\rangle$
- $z_1$ terminates, but $z_2$ blocks or deadlocks, that is: $\sigma = \langle z_1; z_2,\ s_1\rangle \xrightarrow{l_i} \cdots \xrightarrow{l_n} \langle z_2^n, s_n\rangle$ where $\langle z_1,\ s_1\rangle \xrightarrow{l_i} \cdots \xrightarrow{l_k} \langle \epsilon, s_{k+1}\rangle$ and $\langle z_2,\ s_{k+1}\rangle \xrightarrow{l_{k+1}} \cdots \xrightarrow{l_n} \langle z_2^n, s_n\rangle$ hold and $z_2^n$ is possibly $\epsilon$.

In the remainder $\langle z_1,\ s_1\rangle \xrightarrow{l_i} \cdots \xrightarrow{l_k} \langle \epsilon, s_{k+1}\rangle$ and $\langle z_2,\ s_{k+1}\rangle \xrightarrow{l_{k+1}} \cdots \xrightarrow{l_n} \langle z_2^n, s_n\rangle$ are called $\sigma'$ and $\sigma''$ respectively. Since $events(z_1) = \{\}$, $z_1$ satisfies its specification in any binding where the assumptions hold. That is,

$$H_3\colon\ \models_\pi z_1\ \underline{sat}\ (\vartheta, \mathcal{B})::(P,\ R,\ G,\ E_1 \wedge P_2).$$

From this, not only $\sigma' \in ext[\vartheta, P, R] \cap cp[z_1]$ holds, but also $\sigma' \in ext[(\vartheta, \mathcal{B}), P, R] \cap cp[z_1]$ holds.

Suppose $\sigma \in ext[(\vartheta, \mathcal{B}), P, R]$ holds. If $z_1; z_2$ deadlocks then, $\sigma \in int[(\vartheta, \mathcal{B}), G, E_1 \mid E_2]$. If, however, $\sigma$ terminates, then, since $\sigma'' \in int[(\vartheta, \mathcal{B}), G, E_2]$ and $\sigma' \in int[(\vartheta, \mathcal{B}), G, E_1]$, it results that $\models_\pi (s_1, s_{k+1})\ \underline{sat}\ E_1$ and $\models_\pi (s_{k+1}, s_n)\ \underline{sat}\ E_2$, hence $\models_\pi (s_1, s_n)\ \underline{sat}\ E_1 \mid E_2$ holds. Further, since any transition of $\sigma$ is either a transition of $\sigma'$ or of $\sigma''$, it results that any program transition in $\sigma$ satisfies $G$, hence the rule.

### 15.3.4 Announce Rule

We show that the announce rule for the composition of specifications is sound. This is the rule for the derivation of behavioral specifications from structural specifications. The assumptions are the following:

$$H_1\colon\ \models_\pi \{z_{11} \| \cdots \| z_{n1} \| z_{n+1}\}\ \underline{sat}\ (\vartheta, \mathcal{B})::S_1$$

$$H_2\colon\ \models_\pi \{z_{12} \| \cdots \| z_{n2} \| z_{n+1}\}\ \underline{sat}\ (\vartheta, \mathcal{B})::S_2$$

$$H_3\colon\ \models_\pi \forall e \in X_1 \cdot subscribers_{\mathcal{B}}(e) = \{z_{11}, \cdots z_{n1}\}$$

$$H_4\colon\ \models_\pi \forall e \in X_2 \cdot subscribers_{\mathcal{B}}(e) = \{z_{12}, \cdots z_{n2}\}$$

$$H_5\colon\ \models_\pi z\ \underline{sat}\ \vartheta::(P,\ R,\ G,\ E); \textbf{announce}(e)$$

$$H_6:\ events(z) = \{e : Event \cdot \exists s_1, s_2 : State \cdot (s_1, s_2) \models_\pi E \wedge e = e\} = X_1 \uplus X_2.$$

From which we must derive that the following formula holds:

$$C_1\colon\ \models_\pi z; z_{n+1}\ \underline{sat}\ \ (\vartheta, \mathcal{B})::(P,\ R,\ G,\ E); \textbf{if } e \in X_1 \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}.$$

which means that we must proof the existence of three programs $z_{t_0}$, $z_{t_1}$, $z_{t_2}$, such that

$C_2$: $\models_\pi z_{t_0}$ *sat* $\vartheta ::(P,\ R,\ G,\ E)$

$C_3$: $\models_\pi z_{t_1}$ *sat* $(\vartheta, \mathcal{B}) ::S_1$

$C_4$: $\models_\pi z_{t_2}$ *sat* $(\vartheta, \mathcal{B}) ::S_2$

$C_5$: $\models_\pi z; z_{n+1}$ **behaves as** $z_{t_0}$; **if** $b$ **then** $z_{t_1}$ **else** $z_{t_2}$ **fi**

We denote the program $z_{t_0}$; **if** $b$ **then** $z_{t_1}$ **else** $z_{t_2}$ **fi** as $z_t$. Applying Definition 25 to $H_5$ it results that there exists a program, that we call $z_{t_0}$ such that

$C_6$ : $z = z_{t_0}$; **announce**$(e)$

$C_7$: $\models_\pi z_{t_0}$ *sat* $\vartheta ::(P,\ R,\ G,\ E)$.

The existence of $z_{t_0}$ as required by $C_2$ is, hence satisfied. Next we choose $z_{t_1}$ and $z_{t_2}$ such that:

$C_6$ : $z_{t_1} = \{z_{11} \| \cdots \| z_{n1} \| z_{n+1}\}$

$C_7$ : $z_{t_2} = \{z_{12} \| \cdots \| z_{n2} \| z_{n+1}\}$.

By the assumptions $H_1$ and $H_2$ it is clear that these programs satisfy the requirements $C_3$ and $C_4$.

Next, we now need to discharge $C_5$. That is, we show that for any behavioral specification $(\vartheta, \mathcal{B}) ::(P_1, R_1, G_1, E_1)$, the following holds:

$$\models_\pi z_t \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P_1, R_1, G_1, E_1) \ \Rightarrow\ \models_\pi z; z_{n+1} \ \underline{sat}\ (\vartheta, \mathcal{B}) ::(P_1, R_1, G_1, E_1).$$

which means that:

$$ext[(\vartheta, \mathcal{B}), P_1, R_1] \cap cp[z_t] \subseteq int[(\vartheta, \mathcal{B}), G_1, E_1] \ \Rightarrow\ ext[(\vartheta, \mathcal{B}), P_1, R_1] \cap cp[z; z_{n+1}] \subseteq int[(\vartheta, \mathcal{B}), G_1, E_1]$$

This is done by proving that any computation of $z; z_{n+1}$ can be transformed into a computation of $z_t$ with the same sequence of states and the same sequence of transition labels between the states. Considering $C_6$, the program $z; z_{n+1}$ can also be written as $z_{t_0}$; **announce**$(e)$; $z_{n+1}$

First, we consider a computation $\sigma \in ext[(\vartheta, \mathcal{B}), P_1, R_1] \cap cp[z_{t_0}; \textbf{announce}(e); z_{n+1}]$ that has no environment transition. By hypothesis $H_6$, any event announced by $z_t$ is either in $X_1$ or in $X_2$. And, the function $subscribers_{\mathcal{B}}$ is uniform in each of these sets. This computation is, therefore, of one of the forms:

- $\langle z_{t_0}; \mathbf{announce}(e); z_{n+1}, s_1\rangle \xrightarrow{l} \cdots \xrightarrow{l} \langle z'_{t_0}\mathbf{announce}(e); z_{n+1}, s'_1\rangle$
  where $\langle z_{t_0}, s_1\rangle \xrightarrow{l} \cdots \xrightarrow{l} \langle z'_{t_0}, s'_1\rangle$; $z_{t_0}$ deadlocks before the announcement of the event.

- $\sigma_{t_1} = \langle z_{t_0}; \mathbf{announce}(e); z_{n+1}, s_1\rangle \xrightarrow{l} \cdots \xrightarrow{l} \langle \mathbf{announce}(e); z_{n+1}, s'_1\rangle \xrightarrow{i} \langle z_{t_1}, s'\rangle \xrightarrow{l} \sigma''$
  where $\langle z_{t_0}, s_1\rangle \xrightarrow{l} \cdots \xrightarrow{l} \langle \epsilon, s'_1\rangle$; $z_{t_0}$ terminates in the state $s'_1$ such that $s'_1(e) \in X_1$.

- $\sigma_{t_2} = \langle z_{t_0}; \mathbf{announce}(e); z_{n+1}, s_1\rangle \xrightarrow{l} \cdots \xrightarrow{l} \langle \mathbf{announce}(e); z_{n+1}, s'_1\rangle \xrightarrow{i} \langle z_{t_2}, s'\rangle \xrightarrow{l} \sigma''$
  where $\langle z_{t_0}, s_1\rangle \xrightarrow{l} \cdots \xrightarrow{l} \langle \epsilon, s'_1\rangle$; $z_{t_0}$ terminates in the state $s'_1$ such that $s'_1(e) \in X_2$.

These computations are in $int[(\vartheta, \mathcal{B}), G_1, E_1]$ if one of the following holds:

- $z_{t_0}$ deadlocks before the event announcement,
- $z_{t_0}$ terminates but $s'_1(e) \in X_1$ and one of the programs $z_{11}, \cdots, z_{1n}, z_{n+1}$ deadlocks,
- $z_{t_0}$ terminates but $s'_1(e) \in X_2$ and one of the programs $z_{21}, \cdots, z_{2n}, z_{n+1}$ deadlocks.

Let us now assume that this is not the case and all these programs terminate.

The computations $\sigma_{t_1}$ and $\sigma_{t_2}$ are transformed into the following computations:

$\sigma'_{t_1} = \langle z_{t_0}; \mathbf{if}\ e\ \in X_1\ \mathbf{then}\ z_{t_1}\ \mathbf{else}\ z_{t_2}\ \mathbf{fi}, s_1\rangle \xrightarrow{l} \cdots \xrightarrow{l} \langle \mathbf{if}\ e\ \in X_1\ \mathbf{then}\ z_{t_1}\ \mathbf{else}\ z_{t_2}\ \mathbf{fi}, s'\rangle \xrightarrow{i} \langle z_{t_1}, s'\rangle \xrightarrow{l} \sigma''$

$\sigma'_{t_2} = \langle z_{t_0}; \mathbf{if}\ e\ \in X_1\ \mathbf{then}\ z_{t_1}\ \mathbf{else}\ z_{t_2}\ \mathbf{fi}, s_1\rangle \xrightarrow{l} \cdots \xrightarrow{l} \langle \mathbf{if}\ e\ \in X_1\ \mathbf{then}\ z_{t_1}\ \mathbf{else}\ z_{t_2}\ \mathbf{fi}, s'\rangle \xrightarrow{i} \langle z_{t_2}, s'\rangle \xrightarrow{l} \sigma''$

where the program $\mathbf{announce}(e); z_{n+1}$ is replaced with $\mathbf{if}\ e\ \in X_1\ \mathbf{then}\ z_{t_1}\ \mathbf{else}\ z_{t_2}\ \mathbf{fi}$. And in fact, following the semantics of the conditional construct, the latter computations are indeed valid computations of the program $z_{t_0}; \mathbf{if}\ e\ \in X_1\ \mathbf{then}\ z_{t_1}\ \mathbf{else}\ z_{t_2}\ \mathbf{fi}$

In particular, the projections of the computation $\sigma'_{t_1}$ to its set of states is equal to the projection of $\sigma_{t_1}$ to its set of states while any label transition in $\sigma_{t_1}$ is a label transition in $\sigma_{t_1}$. It follows that if $\sigma'_{t_1} \in int[(\vartheta, \mathcal{B}), G, E]$ holds, then $\sigma_{t_1} \in int[(\vartheta, \mathcal{B}), G, E]$ also does. Respectively, if $\sigma'_{t_2} \in int[(\vartheta, \mathcal{B}), G, E]$ holds, then $\sigma_{t_2} \in int[(\vartheta, \mathcal{B}), G, E]$ also does.

On the other hand, if $\sigma_{t_1} \in ext[(\vartheta, \mathcal{B}), P, R]$ holds, then $\sigma'_{t_1} \in ext[(\vartheta, \mathcal{B}), P, R]$ also does. Respectively, if $\sigma_{t_2} \in ext[(\vartheta, \mathcal{B}), P, R]$ holds, then $\sigma_{t_2} \in int[(\vartheta, \mathcal{B}), P, R]$ also does.

It, therefore, follows that $\sigma'_{t_1}$ and $\sigma'_{t_2}$ are in $ext[(\vartheta, \mathcal{B}), P_1, R_1]$ and by the assumption that $\models_\pi z_t\ \underline{sat}\ (\vartheta, \mathcal{B})::(P_1, R_1, G_1, E_1)$ holds, it results that $\sigma'_{t_1}$ and $\sigma'_{t_2}$ are in $int[(\vartheta, \mathcal{B}), G_1, E_1]$, hence $\sigma_{t_1} \in int[(\vartheta, \mathcal{B}), G_1, E_1]$, and $\sigma_{t_2} \in int[(\vartheta, \mathcal{B}), G_1, E_1]$ also hold.

### 15.3.5 Integration Rule

The integration rule is simple and its proof is indeed straightforward. Assume the following:

$$H_1 : z_1 \notin \mathsf{dom}\ \mathcal{B}$$

$$H_2 : \mathcal{B}_1 = \mathcal{B} \cup \{z_1 \mapsto \{\}\}$$

$$H_3 : \models_\pi z\ \underline{sat}\ (\vartheta,\ \mathcal{B}) : : (P,\ R,\ G,\ E)$$

Any computation $\sigma \in ext[(\vartheta,\ \mathcal{B}_1), P, R] \cap cp[z]$ is such that either it contains some event announcement or not.

If it contains no event announcement, then its behavior is not impacted by the binding $\mathcal{B}$ and inserting a new method into it is not relevant. It follows that $\sigma \in ext[(\vartheta,\ \mathcal{B}), P, R] \cap cp[z]$ also holds. And by hypothesis $H_2$, $\sigma \in int[(\vartheta,\ \mathcal{B}), G, E]$ follows. That is, any program transition in $\sigma$ that alters the state satisfies $G$ while the final state satisfies $E$. Since $\sigma$ is, however, independent on the binding, $\sigma \in int[(\vartheta, \mathcal{B}), G, E]$.

Let us now assume that $\sigma$ contains some event announcement and is, hence of one the forms:

$$\sigma' \xrightarrow{l} \langle \{^n\mathbf{announce}(e) \| z\}^m, s\rangle \xrightarrow{i} \langle \{\| subscribers_{\mathcal{B}_1}(e) \| \{^n z\}^{m+1}, s\rangle \xrightarrow{l} \sigma''$$

$$\sigma' \xrightarrow{l} \langle \{^n\mathbf{announce}(e); z\}^m, s\rangle \xrightarrow{i} \langle \{\| subscribers_{\mathcal{B}_1}(e) \| \{^n z\}^{m+1}, s\rangle \xrightarrow{l} \sigma''$$

$$\sigma' \xrightarrow{l} \langle \mathbf{announce}(e), s\rangle \xrightarrow{i} \langle \| subscribers_{\mathcal{B}_1}(e), s\rangle \xrightarrow{l} \sigma''$$

Since, however, $subscribers_{\mathcal{B}_1}(e) = subscribers_{\mathcal{B}}(e)$ for any event $e$ (by $H_1$ and $H_2$) and $\sigma \in [(\vartheta,\ \mathcal{B}), G, E]$ (by $H_3$), it follows that $\sigma \in int[(\vartheta,\ \mathcal{B}_1), G, E]$ holds and subsequently $\models_\pi z\ \underline{sat}\ (\vartheta,\ \mathcal{B}) : : (P,\ R,\ G,\ E)$ also holds.

## 15.4 SeaTy Rules

SeaTy specifications have been defined relative to Lecap specifications. In particular, a Lecap specification is regained from a SeaTy specification by making its rely- and guar-conditions explicit. A SeaTy specification such as $(\vartheta_1, \mathcal{B}) : : (P,\ E)$ is a reformulation of the pure Lecap specification $(\vartheta \cup \vartheta_1, \mathcal{B}) : : (P,\ I_{\vartheta_1},\ I_{\vartheta \setminus \vartheta_1},\ E)$ iff $\vartheta \cup \vartheta_1$ covers $\mathcal{B}$ and $\vartheta \cap \vartheta_1 = \{\}$. Similarly, $\vartheta_1 : : (P,\ E)$ is a reformulation of the pure Lecap specification $\vartheta \cup \vartheta_1 : : (P,\ I_{\vartheta_1},\ I_{\vartheta \setminus \vartheta_1},\ E)$ if $\vartheta \cap \vartheta_1 = \{\}$. A program that satisfies this specification is a program that relies on that the environment will not change the variables that it has

access to. On the other hand, such a program guarantees to change none of the variables it is not allowed to access.

To prove the soundness of SEATY rules, we, therefore, convert them to LECAP specifications. The process is straightforward and we illustrate it for some few rules.

### 15.4.1 SEQUENTIAL RULE

We need to show that

$$H_1\colon \models_\pi z_1; z_2 \ \underline{sat}\ \vartheta :: (P_1,\ E_1 \mid E_2)$$

follows from

$$H_2\colon \models_\pi z_1 \ \underline{sat}\ \vartheta :: (P_1,\ E_1 \wedge P_2)$$
$$H_3\colon \models_\pi z_2 \ \underline{sat}\ \vartheta :: (P_1,\ E_2)$$

Let us consider a set of variables $\vartheta_1$ such that $\vartheta \subseteq \vartheta_1$.

The above formulas are written as the following LECAP specifications:

$$H_1\colon \models_\pi z_1; z_2 \ \underline{sat}\ \vartheta :: (P_1,\ I_\vartheta,\ I_{\vartheta_1 \setminus \vartheta},\ E_1 \mid E_2)$$
$$H_2\colon \models_\pi z_1 \ \underline{sat}\ \vartheta_1 :: (P_1,\ I_\vartheta,\ I_{\vartheta_1 \setminus \vartheta},\ E_1 \wedge P_2)$$
$$H_3\colon \models_\pi z_2 \ \underline{sat}\ \vartheta_1 :: (P_1,\ I_\vartheta,\ I_{\vartheta_1 \setminus \vartheta},\ E_2)$$

And, the rule results from a direct application of the basic structural sequential rule.

### 15.4.2 PARALLEL RULE

Two parallel rules were proposed whose soundness we prove in this subsection. In the first case, two programs running concurrently are required not to share variables. In the other case, each program is executed atomically.

**First Parallel Rule.**
We consider the three following valid formulas:

$H_1$ : $\vartheta_1 \cap \vartheta_2 = \{\}$

$H_2$: $\models_\pi z_1$ *sat* $\vartheta_1 : :(P_1,\ E_1)$

$H_3$: $\models_\pi z_2$ *sat* $\vartheta_2 : :(P_2,\ E_2)$.

Let also $\vartheta$ be a set of variables such that $\vartheta_1 \cup \vartheta_2 \subseteq \vartheta$. $H_2$ and $H_3$ are rewritten as the following LECAP formulas:

$H_2$: $\models_\pi z_1$ *sat* $\vartheta : :(P_1, I_{\vartheta_1}, I_{\vartheta \setminus \vartheta_1},\ E_1)$

$H_3$: $\models_\pi z_2$ *sat* $\vartheta : :(P_2, I_{\vartheta_2}, I_{\vartheta \setminus \vartheta_2},\ E_2)$.

From $\vartheta_2 \cap \vartheta_1 = \{\}$, one derives that $\vartheta_2 \subseteq \vartheta \setminus \vartheta_1$ and $\vartheta_1 \subseteq \vartheta \setminus \vartheta_2$ which results in $I_{\vartheta \setminus \vartheta_1} \Rightarrow I_{\vartheta_2}$ and $I_{\vartheta \setminus \vartheta_2} \Rightarrow I_{\vartheta_1}$. The two programs $z_1$ and $z_2$, therefore coexist and their parallel composition is given by the LECAP rule:

$$\models_\pi \{z_1 \| z_2\}\ \underline{sat}\ \vartheta : :(P_1 \wedge P_2, I_{\vartheta_1} \wedge I_{\vartheta_2}, I_{\vartheta \setminus \vartheta_1} \vee I_{\vartheta \setminus \vartheta_2},\ E_1 \wedge E_2)$$

which we refine into:

$$\models_\pi \{z_1 \| z_2\}\ \underline{sat}\ \vartheta : :(P_1 \wedge P_2, I_{\vartheta_1 \cup \vartheta_2}, I_{\vartheta \setminus (\vartheta_1 \cup \vartheta_2)},\ E_1 \wedge E_2)$$

And, rewriting this in the SEATY style we obtain:

$$\models_\pi \{z_1 \| z_2\}\ \underline{sat}\ \vartheta_1 \cup \vartheta_2 : :(P_1 \wedge P_2,\ E_1 \wedge E_2).$$

**Second Parallel Rule.**

Given the assumptions:

$H_1$: $\models_\pi P_1 \mid (E_2 \wedge B) \Rightarrow P_1$

$H_2$: $\models_\pi A \mid (E_2 \wedge B) \Rightarrow A$

$H_3$: $\models_\pi P_2 \mid (E_1 \wedge A) \Rightarrow P_2$

$H_4$: $\models_\pi B \mid (E_1 \wedge A) \Rightarrow B$

$H_5$: $\models_\pi z_1$ *sat* $\vartheta_1$ : :**await true do** $(P_1,\ E_1 \wedge A)$ **od**

$H_6$: $\models_\pi z_2$ *sat* $\vartheta_1$ : :**await true do** $(P_2,\ E_2 \wedge B)$ **od**

we must derive the validity of the following judgment:

$$H_7\colon\ \models_\pi \{z_1 \| z_2\}\ \underline{sat}\ \vartheta :: (P_1 \wedge P_2,\ A \wedge B).$$

We make explicit the rely- and guarantee conditions of $z_1$, $z_2$, and $\{z_1 \| z_2\}$. $\vartheta$ is a set of variables such that $\vartheta_1 \subset \vartheta$.

$$H_8\colon\ \models_\pi z_1\ \underline{sat}\ \vartheta :: \textbf{await true do } (P_1, I_{\vartheta_1}, I_{\vartheta \setminus \vartheta_1},\ E_1 \wedge A) \textbf{ od}$$
$$H_9\colon\ \models_\pi z_2\ \underline{sat}\ \vartheta :: \textbf{await true do } (P_2, , I_{\vartheta_1}, I_{\vartheta \setminus \vartheta_1},\ E_2 \wedge B) \textbf{ od}$$

we must derive the validity of the following judgment:

$$H_{10}\colon\ \models_\pi \{z_1 \| z_2\}\ \underline{sat}\ \vartheta :: (P_1 \wedge P_2, I_{\vartheta_1}, I_{\vartheta \setminus \vartheta_1},\ A \wedge B).$$

From the definition of await structural specifications, $z_1$ and $z_2$ are two programs of the form:

$$z_1 = \textbf{await true do } z_1' \textbf{ od}$$
$$z_2 = \textbf{await true do } z_2' \textbf{ od}$$

where

$$\models_\pi z_1'\ \underline{sat}\ (P_1, I_{\vartheta_1}, I_{\vartheta \setminus \vartheta_1},\ E_1 \wedge A)$$
$$\models_\pi z_2'\ \underline{sat}\ (P_2, , I_{\vartheta_1}, I_{\vartheta \setminus \vartheta_1},\ E_2 \wedge B)$$

Any computation of $\{z_1 \| z_2\}$ is of one of the following forms:

$$\sigma = \langle \{z_1 \| z_2\}, s_1 \rangle \xrightarrow{v} \cdots \xrightarrow{v} \langle \{z_1 \| z_2\}, s_k \rangle \xrightarrow{i} \langle z_1, s_{k+1} \rangle \xrightarrow{v} \cdots \xrightarrow{v} \langle z_1, s_n \rangle \xrightarrow{i} \langle \epsilon, s_{n+1} \rangle \xrightarrow{v} \cdots \xrightarrow{v} \langle \epsilon, s_m \rangle$$

$$\sigma = \langle \{z_1 \| z_2\}, s_1 \rangle \xrightarrow{v} \cdots \xrightarrow{v} \langle \{z_1 \| z_2\}, s_k \rangle \xrightarrow{i} \langle z_2, s_{k+1} \rangle \xrightarrow{v} \cdots \xrightarrow{v} \langle z_2, s_n \rangle \xrightarrow{i} \langle \epsilon, s_{n+1} \rangle \xrightarrow{v} \cdots \xrightarrow{v} \langle \epsilon, s_m \rangle$$

In the first case, the environment, performs a finite number $k$ of transitions before the control is given to the program $\{z_1 \| z_2\}$ which performs its first program transition consisting in the atomic execution of $z_1$. After, this, a finite number of environment transition

is also performed followed by the atomic execution of $z_2$ and a further finite number of environment transitions.

The second case, differs in that the execution of $\{z_1 \| z_2\}$ is done by executing $z_2$ first and $z_1$ next.

The environment is not allowed to change a variable used by $\{z_1 \| z_2\}$, that is, a variable in $\vartheta_1$. Therefore, the pre-conditions $P_1$ and $P_2$ indeed hold after the first sequence of environment transitions. Next, the program $z_1$ is executed such that $(s_1, s_k) \models_\pi E_1 \wedge A$ holds. Since, however, $P_2$ is stable with respect to $E_1 \wedge A$, the pre-condition $P_2$ also holds in the state $s_k$ and in the state $s_n$. The execution of $z_2$ in the state $s_n$ therefore terminates in the state $s_{n+1}$ with $E_2 \wedge B$. And, since $A$ is stable with respect to $E_2 \wedge B$, $A$ also holds.

## 15.5 Summary

This chapter discussed the soundness of the LECAP proof system. In particular, this was done in analogy to the verifcation of the soundness of the rely-/guarantee proof systems of Stølen [121] and Xu [135]. This similarity justifies the omission of the soundness proof of some other rules.

# Chapter 16
# Future Work

The event-based style is a communication paradigm that is increasingly deployed in the development of emerging software systems ranging from desktop applications to large scale distributed and critical systems. Suitable methodologies that allow constructing reliable and dependable event-based applications are, therefore, required.

In this chapter we identify the next steps in establishing such a methodology in general and in the further elaboration of LECAP in particular. We categorize future works in five sections. Section 16.1 discusses some possible improvements to the LECAP theory while Section 16.2 discusses some issues related to its applicability. In Section 16.3, we discuss extending LECAP to support e.g. data reification, testing, and model-checking. We motivate the need for the identification of further architectural types in Section 16.4 while Section 16.5 discusses the design of event-based applications using other techniques. Section 16.6 summarizes the chapter.

## 16.1 Improvements

The LECAP methodology can be improved in many respects.

### 16.1.1 LOCAL VARIABLES

Local variables are necessary for the practical construction of software systems. The LECAP approach still misses this paradigm. Although the work of Stølen [121] and Xu [136] on which our approach is based illustrate how to tackle the issue of local variables in rely-/guarantee reasoning, they do not support method invocation and event announcement that introduce some difficulties in the manipulation of such variables. Short term future work includes solving this issue.

### 16.1.2 Announcing Events in Loops

The work presented in this thesis does not support the announcement of events in the while construct. Although such uses are not common in practice, some cases exist where announcing events in a loop may be required.

Consider for instance the stack-counter example that we presented. It is possible that *impl-push* be used in a loop for the addition of a finite set of elements on the stack. A typical program that may be used for this purpose is following:

impl-while-push() $\triangleq$
**while** $i > 0$ **do**
elt:=$Element_i$;
i:=i-1;
impl-push;
**if** $i/4 \geq 5$ **then announce**(*mk-Event*($\langle pushaction \rangle$, *elt*)) **else skip fi**
**od**

The issue is how to formulate the specification of such a program. Of course we could allow a structural specification of the form

$\vartheta$ : :**while** $b$ **do** $S_1$**od**

The specification of the above program would then be written as:

**while-push**() $\triangleq$ $\vartheta$ : :**while** $i > 0$ **do** $S_1$;
**if** $i/4 \geq 5$ **then announce**(*mk-Event*($\langle pushaction \rangle$, *elt*)) **else** $S_2$ **fiod**

provided that

- *elt*: = $Element_i$; *i*: = *i*-1; *impl-push* *sat* $\vartheta$ : : $S_1$, *and*
- **skip** *sat* $\vartheta$ : : $S_2$.

are valid.

This is, however, simply postponing the problem. How would we transform such a structural specification into a behavioral specification? The issue needs to be further investigated.

## 16.2 Practical Issues

From a practical viewpoint there are also some points that need to be investigated.

### 16.2.1 Await construct in practice

To the best of our knowledge we are the first to propose the use of distributed synchronization and mutual exclusion in the context of the event-based paradigm. Although many possible semantics could be defined, we adopted what we think may be more useful in practice. Alternate semantics are discussed in the next chapter.

Our semantics of the await construct is the atomic execution of a program including all programs that are eventually triggered following its execution. That is, if a program $z$ announces an event $e_1$ which triggers the program $z_2$ and $z_2$ announces the event $e_3$ which triggers $z_3$, any execution of $z$ in an await construct includes the execution of $z_1$, $z_2$, and $z_3$. Such a construct is clearly indispensable for the implementation of distributed atomic transactions as is often the case in bank and business to business workflow systems.

The requirement in future work would be to provide a prototype implementation of this construct. None of the existing middleware or integration frameworks that we are aware of provides such a construct. Coulouris et al. [29] indeed recognize that today's message oriented middleware lack this construct and can, therefore, be used only for a restricted number of scenarios.

### 16.2.2 Case studies

We have proposed a methodology for the construction of a class of software systems that is increasingly important. Although we believe that this methodology is promising, more substantial and non-trivial case studies are needed for further experimenting the approach since the event-based paradigm itself is very pervasive [40]. In general, while methodologies show how to tackle a problem, case studies are the measure that a proposed solution can indeed be used in practice; they represent a valuable informal way for illustrating how to use the methodology. Therefore, developing more case studies is one of our short term goals. In particular, the MOTION platform is an important case since the application of traditional lightweight formal methods was shown to be insufficient for ensuring its reliability. We have presented one of its facets in this thesis. Other aspects of this platform are being analyzed.

### 16.2.3 Tool support

Tool support is widely recognized as an important factor for the success of any software engineering methodology in general and of formal approaches in particular. Various criteria exist for evaluating such tools: early payback, incremental use for incremental effort, multiple use, ease of use, ease of learning, evolutionary development [28].

The examples presented in this thesis indeed confirm these requirements which range from very simple features (such as syntax highlighting and automatic text completion) to proof obligations generation and automatic verification of properties.

As the examples show, the development of an event-based application follows a a clear development process with clearly defined steps. Ideally, an integrated tool should provide support for each of these steps. At the abstract level, construction of structural specifications should be supported. Next, construction of bindings must also be easily done in the development environment. An important point to tackle is the derivation of behavioral specifications starting with structural specifications. In this derivation process, the tool will need to generate the proof obligations and display them for verification. Such proof obligations can subsequently be discharged using either a theorem prover or an automatic analyzer. Ideally, all these tools should be integrated in the same environment.

Although building such a tool is a real challenge, from a more practical point of view we have started investigating the use of Alloy [70] in the Eclipse [127] development environment to solve some of these requirements. It is a declarative first order language that can be viewed as a subset of Z [33]. Alloy is a declarative language similar to the formal specification languages Z [33] and VDM [102]. Unlike other declarative specification languages such as VDM [102, 75] and Z [33] Alloy is automatically analyzable in the style of model checking giving to designers the kind of immediate feedback that testing gives to programmers.

On the other hand, Eclipse is an open source software development project that aims at providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools.

### 16.2.4 Domain Specific Application

As discussed in the introduction of this thesis, the event-based paradigm is exploited in various application domains: graphical user interface construction, mobile computing, pervasive and ubiquitous computing, component based software engineering, distributed loosely coupled workflow systems, etc.

It is therefore important to investigate how the methodology that we propose can be applied

to meet the specific requirements of such application domains. For instance, distributed workflow systems require to be able to correlate events such that the matching function becomes time dependent. We believe that our model of event-based system can be applied here since we gave no precise semantics to the matching function. Nonetheless, deep research must be undertaken to answer such questions unequivocally.

## 16.3 EXTENSIONS

### 16.3.1 DATA REIFICATION

A stepwise development process is normally divide into 1) operation decomposition (or refinement) and 2) data reification (see [75]).

A part of the rules we proposed in this thesis serves the purpose of decomposing operations. That is, the formal specification of a program is successively decomposed until an implementation is reached. In addition to this, we proposed a process of composing specifications that consists in transforming structural specifications into behavioral specifications and discharging the related proof obligations.

What is missing in our treatment is data reification also called data refinement. Usually, as operations are decomposed, data need also to be successively transformed to eventually become data structures at the implementation level. For instance, for designing a dictionary, one may start with a simple VDM map and successively refine it such that it eventually becomes a Java hashtable.

Date reification (at least for the SEATY type), can be adapted from existing techniques. In [75], a retrieve function *retr* is defined that regains the abstraction from the implementation details. That is, it maps elements from the concrete domain $D_c$ to the abstract domain. Two criteria are defined for constraining such functions: adequacy and totality. By the totality criterion retrieve functions are required to be total, i.e. each element of the concrete domain must be mapped to at least one element of the abstract domain (see Proof Obligation 38).

**Proof Obligation 38** $\forall c \in D_c, retr(c) \in D_a$

Conversely, the adequacy requirement (see Proof Obligation 39) ensures that each element of the abstract domain must be the image of some element of the concrete domain.

**Proof Obligation 39** $\forall a \in D_a, \exists c \in D_c, retr(c) = a$

Retrieve functions may, hence be one-to-many; although many-to-one functions may also arise in practice, Jones [75] rule them out by arguing that "If different abstract values correspond to one concrete value, it is intuitively obvious that such values could have been merged in the abstraction." In the case of event-based systems, the requirement for excluding such situations is more serious. If two programs $z_1$ and $z_2$ subscribe to two events $e_1$ and $e_2$ that are different from each other at the abstract level, there is no apparent reason why they may be required to coexist (at the abstract level). At the concrete level, however, a many-to-one (many abstract values to one concrete value) retrieve function could map them to the same value, leading to the simultaneous invocation of the two programs at the concrete level, hence to interference.

On the other hand, Jones allows one-to-many (one abstract value to many concrete values) retrieve functions in his adequacy requirement. It must be investigated how well this requirement can be accepted in event-based system. We suspect that events and subscriptions must be subject to one-to-one retrieve functions (as required in Section 7.4.12). This must further be investigated.

### 16.3.2 Testing Event-Based Applications

The combination of formal specification and software testing have resulted in an alternate verification technique called formal testing. A program is formally specified and implemented in the light of the formal specification which is further used for the derivation of test cases and for the construction of test oracles.

Independently of the question of whether testing is a valuable solution to ensuring software correctness or not, we argue that a testing process must be developed for event-based applications. Since testing is the most used verification technique, we need to supplement it with the knowledge that was gained in the construction of Lecap .

We envision extending our previous work [43] to event-based applications. This extension must include defining what is a test suite in the context of event-based applications. Next, we must identify what are the parts of the system that need to be tested and finally how to construct a test oracle. Intuitively, we suggest that the testing process should be compositional and be used in the form of a refinement. That is the formal specifications of components are developed, the specification of the application composed as shown in this thesis, the local properties and the global properties of specifications discharged, and the testing process used for ensuring that an implemented component indeed satisfies a given structural specification.

### 16.3.3 Model-Checking Event-Based Applications

Model checking EB applications is an intriguing alternative to the formal proof of software systems as a significant part of the process is carried out automatically. In [56], an attempt to apply model checking to the verification of EB applications is discussed. The authors try to provide a generic framework that can be reused by modelers in the process of defining the abstract structure related to their systems. Indeed, the authors succeeded in factoring the work such that, for instance, the event delivery policy is now a pluggable element with various packaged policies (prepared by the authors) that can be used off-the-shelf. They, however, concentrate on the run-time apparatus, i.e. the middleware. Not much is provided for tackling the correctness of the application (consumers and publishers) built on top of this middleware. This shows that naive approaches to model-checking event-based applications do not work.

We believe that our framework provides a good and realistic starting point for model-checking event-based applications. A compositional approach can be constructed that consists of model-checking the structural specification of components, model-checking the behavioral specifications for discharging global properties and possibly model-checking the component implementations against their structural specifications.

## 16.4 Simplifications

The Lecap framework is a general framework that can be applied to a wide range of software systems. Because of this generality, however, some may find it difficult to apply since most software systems often require only a small subset of theories. For instance, do all applications really need the rely/guarantee conditions? Does the proof system become simpler if any two subscribers are constrained not to share variables? Yes, the SeaTy type presented in Chapter 11 is an answer to this question. Other such simplifications can, and perhaps need to, be investigated.

Although we have not yet investigated some of these questions, we claim that the stack-counter example discussed in this thesis for instance, represents an important class of systems, namely those which announce events only at the end of the execution of methods. It follows that for all such cases, there is no need for the proof system to support the announcement of events at any arbitrary point in a program. We believe, therefore, that event-based architectural types must be identified and the Lecap framework refined to these architectural types with the purpose of simplifying the construction techniques of such systems. An architectural type [13] is obtained from an architectural style by fixing some of its parameters. An architectural type is at an intermediary level of abstraction between the architectural style and an architecture. This may be compared to the concept of problem frame [72] where a frame is defined for handling the requirements of a specific

category of applications.

## 16.5 Related Approaches

We have shown in this thesis (in particular through the operational semantics presented in Chapter 5) that event-based systems share some substantial properties with concurrent systems. For this reason, instead of basing our work on the rely/guarantee paradigm, other frameworks for reasoning about concurrent systems could also have been investigated. In particular, the $\pi$-calculus could be an intriguing alternative since it is intrinsically targeted at designing systems where value passing is important. Note, however, that event-based systems do not only include value passing mechanisms, but also shared variables. And, although reasoning about and writing assertions over communication histories (as in the $\pi$-calculus) may seem to be easier than writing and reasoning about those over state evolutions, the issue of interference clearly remains in the $\pi$-calculus; interference affects liveness arguments as well as safety reasoning [76].

## 16.6 Summary

The chapter presented possible improvements, extensions, and simplifications that can be applied to the LECAP framework for constructing event-based applications that we have proposed.

# Chapter 17

# Discussion and Conclusion

## 17.1 Research Result

We have presented a framework called LECAP for the development of correct event-based applications. The development of this methodology is driven by a set of key requirements for the development of emerging applications presented in Chapter 1. In particular, LECAP is targeted at bringing the intrinsic compositionality of the event-based paradigm to the abstract level such that specifications of components can be composed by constructing bindings that reflect the architecture of the desired applications.

Constructing an application using the LECAP methodology includes the following steps:

1. Designing the architecture of the intended application (identification of components),
2. Constructing the formal structural specifications of the different components which are independent of any binding,
3. Verifying the local properties of the specifications of the components,
4. Transforming the structural specifications of the components into behavioral specifications,
5. Verification of the global properties of the applications based on the behavioral specifications,
6. Top-down refinement of the components.

Our framework for constructing correct event-based applications includes:

1. a core programming language for developing applications. This language is a while parallel language augmented with constructs for supporting 1) synchronization and mutual exclusion, 2) announcement of events, and 3) method invocation.

2. a technique for the specification of event-based applications. Two kinds of specifications are supported: structural specifications for components and behavioral specifications for applications;

3. a set of rules for the stepwise construction of components,

4. a set of rules for the composition of specifications of applications using the specifications of components, and

5. a set of rules for the manipulation of specifications of components.

This thesis also presented some examples for illustrating the approach. Although these examples are neither exhaustive nor canonical, we think that they represent to some degree the essence of the development process of an event-based application.

## 17.2 EXPRESSIVENESS

Many criteria have been proposed for classifying event-based systems (see [56, 57, 96]). In this thesis, we argue that the combination of the LECAP programming language with the definition of event-based systems that we have proposed is, although simple, expressive and flexible enough to encompass many of these criteria. We, therefore, argue that our framework identifies the right abstractions for the event-based paradigm. Many other abstractions have been proposed that were limited on the kind of analysis that they allow on event-based applications [35, 36, 55, 71].

Let us discuss some of these criteria: announcement completion, event-method binding, delivery policy, concurrency, event passing technique, event definition.

- the announcement completion criterion defines when an announcement can be said to be completed. Two important types of completion are distinguished: synchronous and asynchronous. In the first style, the announcement is completed after the complete execution of the subscribers. In the second case, the announcement is completed after the reception of the event by the event-based infrastructure. The default completion mechanism in the LECAP programming language is asynchronous. Synchronous completion can, however, be achieved by embedding an announcement in an await statement.

- the event-method binding criterion determines if a new subscription can be defined at runtime in the system or not. There are static and dynamic bindings. Our system was intentionally designed to support dynamic binding which is more powerful than static binding and is a key requirement for any methodology for constructing event-based applications.

- Four types of delivery policies are distinguished. In full delivery, the announcement of an event results in the invocation of all subscribers. In single delivery, only one subscriber is chosen. In parameter based selection, the invoked subscribers are selected based on the parameters carried by the event. In state based policy finally, a policy is assigned to each event that determines the effect of this event. The LECAP semantics of the announcement construct is based on full delivery. We conjecture that all other delivery mechanisms can easily be adapted from or built on top of LECAP.

- As in programming languages, different techniques may be adopted for passing the announced event to subscribers. The "All parameter" passing mechanism consists of passing exactly the same parameters as are specified by the event. In the "selectable parameters" passing style, the publisher can specify which parameters of the event are passed in the invocation. Finally in "parameter expressions", the invocation passes the results of the passed expressions. Our approach is based on the latter parameter passing technique that corresponds to value passing in programming languages. We believe that this is more suitable to distributed computing.

- The event definition criteria discusses whether an event-based system allows implementers to extend the vocabulary of events or not. Event vocabularies can be constructed by adopting static declaration, dynamic declaration, or no event declaration techniques. This criterion is in fact, obsolete in the LECAP framework; we have not specified the kind of event that we assume, giving developers the possibility to define the kind of vocabulary that better suits their needs. Note that this is different from the "no event declaration" technique as our language is supposed to be typed. Our approach is more related to dynamic declaration.

## 17.3 ALTERNATE SOLUTIONS

Many other solutions have been investigated in the Ph.D. project that resulted in this thesis. We discuss why the solution presented in this thesis is superior.

### 17.3.1 ALTERNATE ANNOUNCEMENT SEMANTICS

The announcement of events can be defined in different ways, resulting in completely different rules.

### Alternate Semantics I

The essence of this approach is to reduce the announcement of an event to writing to a shared variable while another program continuously reads this variable and invokes interested subscribers. This approach requires 1) shared variables for storing events, 2) a program $\pi$ in the form of an await statement whose body is responsible for doing the matching and invoking interested subscribers.

An event-announcement consists of writing to the shared variable(s). After such an announcement, each interested subscriber is invoked and all of them are executed in parallel with each other and with the program $\pi$.

An attempt to formalize this semantics reveals that three non-trivial paradigms are required for giving a semantics to the announce construct: 1) shared variable, 2) synchronization and mutual exclusion, 3) non-termination. This results in very heavy formulas.

Our semantics is derived from this semantics by observing that the shared variables for storing events can be ignored and the invocation of subscribers directly performed by the announce construct. The await-construct, the shared variable, and the requirement for the non-termination of $\pi$ become obsolete, making our semantics simple and natural.

### Alternate Semantics II

The next possible semantics to the announce construct requires a fifth component in the specification of components called announcement condition [46]. Each time this condition is satisfied, a corresponding event is announced and the interested subscribers invoked.

In this approach, a helper program, say *observer*, is needed that runs in parallel with the helped program and waits for announcement conditions to hold. Once an announcement condition holds, the observer invokes the corresponding subscribers.

The advantage of this modeling approach is that it may free the designer from explicitly specifying points of announcement in the code of the program. The definition of such announcement conditions can, hence, be done at the same time as the definition of subscriptions.

The disadvantage of such a semantics is that either 1) it significantly restricts the set of announcement conditions that can be defined or 2) it allows non-deterministic announcement of events. The first disadvantage is illustrated in [46] through the restriction that any two programs running in parallel are not allowed to announce the same event.

We avoid both this non-determinism and these restrictions by allowing the explicit use of the announce construct. Note, however, that this alternate approach can still be con-

structed on top of our language. We believe that our semantics is a reasonable compromise considering the complexity of the semantics, complexity of the derived logic, expressiveness, and applicability.

### 17.3.2 Alternate Specification Techniques

Components that are based on some sort of event-based paradigm may also be specified in various manners. We illustrate two of the approaches that we have investigated.

#### Announcement Conditions

The announcement of events is done by means of a fifth assertion called an ann-(ouncement) condition. A specification is now of the form $(P, R, G, E, Q)$ where $P$ is a unary assertion representing the pre-condition, $R$, $G$, and $E$ are binary assertions representing the rely, guar, and post-conditions respectively. $Q$ is a set of formulas of the form $Q_i \prec e_i$ meaning that whenever the state satisfies the condition $Q_i$ the event $e_i$ must be announced.

In addition to the problem presented in the previous section, this approach introduces a fifth component in the specification of programs making them and the related formulas heavy and difficult to manipulate.

#### Symbolic Variables

A paradigm that we also investigated for specifying event-based applications is that of symbolic variables which are higher-order variables ranging over assertions and used for specifying the behavior of future subscribers. The parallel execution of subscribers to an event $e$ is denoted $z_e$ and specified by means of a symbolic specification $(P_e, R_e, G_e, E_e)$ where $P_e$ is a unary symbolic variable (a variable ranging over unary assertions), while $R_e$, $G_e$, and $E_e$ are binary symbolic variables.

Using such symbolic variables, we can specify the behavior of a program even if the binding is undefined, said to be incomplete. In such an approach, the rules for the decomposition of applications requires postponing the proof obligations until the binding is defined. A fifth component is added to specifications called composition proof obligation that serves as a place holder for proof obligations that we can not yet discharge. A specification was, hence, of the form: $\vartheta::(P, R, G, E)$ **given** $O$ where $O$ is this composition proof obligation. The semantics of such a specification is that a program satisfies it iff for any event-based system $(\vartheta, \mathcal{B})$ where the composition proof obligation $O$ holds, the program $z$ satisfies the related (complete) specification $(\vartheta, \mathcal{B}) : :(P, R, G, E)$. In such a context, the parallel rule is e.g. formulated as:

$$\frac{z_1 \ \underline{sat}\ \vartheta : :(P_1, R_1, G_1, E_1) \textbf{ given } O_1 \quad z_2 \ \underline{sat}\ \vartheta : :(P_2, R_2, G_2, E_2) \textbf{ given } O_2}{\{z_1 \| z_2\} \ \underline{sat}\ \vartheta : :(P_1 \wedge P_2, R_1 \wedge R_2, G_1 \vee G_2, E_1 \wedge E_2) \textbf{ given } O}$$

where $O \stackrel{def}{=} O_1 \wedge O_2 \wedge (G_1 \Rightarrow R_2) \wedge (G_2 \Rightarrow R_1)$.

The requirement for coexistence is postponed since due to symbolic variables, we can not ensure it yet.

Although this technique was shown to be interesting, it resulted in specifications on which some operations such as refinement were difficult to apply. A second kind of specification was needed that we called canonical specifications. The framework required four kinds of specifications: complete specifications, incomplete specifications, canonical specifications, non-canonical specifications. In addition, the paradigms of composition proof obligation and symbolic variables were needed. The theory was clearly very heavy. The framework presented in this thesis results from simplifying this approach by eliminating the concepts complete and incomplete specifications and symbolic variables by observing that composition proof obligations could be derived from canonical specifications (now called structural specifications).

## 17.4 EPILOGUE

It has been argued that the event-based paradigm is troublesome and that developing correct event-based applications is hard [56]. Yet, we argue that although designing correct event-based applications may be a non-trivial task, the difficulties mainly result from the fact that the computational behavior of applications based on this paradigm was not well-understood.

Based on the work in this thesis, we claim that if we put as much effort in the research on the development of correct event-based applications as it was done for shared-variables, procedure invocation, parallel systems, and message passing systems, then developing event-based applications will not be more difficult than building other applications. And, in fact, we showed that an important class of event-based applications can be developed based on the traditional concepts of pre- and post-conditions.

The event-based paradigm differs from many paradigms in the theory of software engineering in that it is strongly motivated by practical, non-trivial, and successful uses. We, therefore, believe that it has some potential in solving many issues in the development of software systems. Although the LECAP framework and its applicability leave us convinced that this work presents a promising first step towards a viable, formal development method-

ology for event-based applications, we plead that the event-based paradigm deserves more attention from theorists.

# BIBLIOGRAPHY


[1] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *ACM Software Engineering Notes*, 18(5):9–20, 1993.

[2] G. D. Abowd, R. Allen, and D. Garlan. Formalizing styles to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, 1995.

[3] P. Aczel. An inference rule for parallel composition. Technical report, University of Manchester, February 1983.

[4] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC 99)*, pages 53–61, 1999.

[5] V. S. Alagar and K. Periyasamy. *Specification of Software Systems.* Springer Verlag, 1998.

[6] Apple Computer. *Inside Macintosh*, volume 6. Addison Wesley, 1991.

[7] K. R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(3):359–385, 1980.

[8] R. J. Back and K. Sere. Action systems with synchronous communication. In *Proceedings of PROCOMET'94 (Programming Concepts, Methods and Calculi)*, pages 107–126, June 1994.

[9] R. J. Back and J. von Wright. Trace refinement of action systems. In *Proceedings of CONCUR'94 (International Conference on Concurrency Theory)*, pages 367–384, August 1994.

[10] D. J. Barret, 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, 1996.

[11] Howard Barringer, Ruurd Kuiper, and Amir Pnueli. Now you may compose temporal logic specifications. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 51–63, 1984.

[12] Klaus Bergner, Andreas Rausch, and Marc Sihling. A componentware development methodology based on process patterns. In *Proceedings of 5th Annual Conference on Pattern Languages of Programs (PLOP98)*, 1998.

[13] Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. On the formalization of architectural types with process algebras. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 140–148. ACM Press, 2000.

[14] Katherine Betz. A scalable stock web service. In *Proceddigns of IEEE International Workshop on Parallel Processing*, pages 145–150, 2000.

[15] K.P. Birman. The progress group approach to reliable distributed computing. *Communications of the ACM*, 12:37–53, December 1993.

[16] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Object Technology Series. Addison Wesley, 1999.

[17] K. Brockschmidt. *Inside OLE.* Microsoft Press, Redmond, 1995.

[18] A. W. Brown and K. C. Wallnau. Engineering of component-based systems. In *Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems, Montreal, Canada*, pages 414–422, October 1996.

[19] S. N. Burris and H.P. Sankappanavar. *A Course in Universal Algebra, The Millennium Edition.* Springer Verlag, January 1981.

[20] C. Bussler and S. Jablonski. Implementing agent coordination for workflow management systems using active database systems. In *Proceedings of RIDE-ADS'94, the 4th International Workshop on Research Issues in Data Engineering, Houston*, pages 53–59, February 1994.

[21] Alexis Campailla, Sagar Chaki, Edmund Clarke, Somesh Jha, and Helmuth Veith. Efficient Filtering in Publish/Subscribe Systems using Binary Decision Diagrams. In *Proceedings of the 21st International Software Engineering Conference (ICSE), Toronto, Canada*, pages 443–452, May 2001.

[22] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *ACM-SIGMOD 1994 International Conference on Management of Data, Minneapolis, Minnesota*, pages 383–394, 1994.

[23] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 3(19):332–383, August 2001.

[24] Antonio Carzaniga, Elisabetta Di Nitto, David S. Rosenblum, and Alexander L. Wolf. Issues in supporting event-based architectural styles. In *Proceedings of 3rd International Software Architecture Workshop, Orlando FL, USA*, pages 17–20, November 1998.

[25] Z. Chen and C. Hoare. Partial correctness of communicating sequential processes. In *Proceedings of the Second IEEE International Conference on Distributed Computer Systems*, April 1981.

[26] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[27] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999.

[28] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, July 1996. Report by the Working Group on Formal Methods for the ACM Workshop on Strategic Directions in Computing Research.

[29] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, Concepts and Design.* Addison-Wesley, 1994. Second Edition.

[30] G. Cugola and E. Di Nitto. Using a Publish/Subscribe Middleware to Support Mobile Computing. In *Proceedings of the Workshop on Middleware for Mobile Computing, in association with IFIP/ACM Middleware 2001 Conference, Heidelberg, Germany*, November 2001.

[31] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI Event-Based Infrastructure and its Application to the Development of the OPSS WFMS. *Transaction of Software Engineering (TSE)*, 27(9):827–850, September 2001.

[32] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[33] Antoni Diller. *Z:An Introduction to Formal Methods.* Oreilly, Mai 1996.

[34] J. Dingel. *Systematic parallel programming.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, December 1999.

[35] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasonning about Implicit Invocation. In *Proceedings of the 6th International Symposium on the Foundations of Software Engineering, FSE-6, Lake Buena Vista, FL*, pages 209–221. ACM, November 1998.

[36] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Towards a formal treatment of implicit invocation using rely/guarantee reasoning. *Formal Aspects of Computing*, 10:193–213, 1998.

[37] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML, The Catalysis Approach.* Addison Wesley Longman, Inc., 1998.

[38] Herbert B. Enderton. *A mathematical introduction to logic.* Hardcourt/Academic Press, 2nd edition, 2001.

[39] C. Ene and Traian Muntean. A broadcast-based calculus for communicating systems. Technical report, Laboratoire d'Informatique de Marseille, 2000.

[40] P.Th. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[41] Pascal Fenkam. Visual Validation of VDM-SL Based Specification. Technical report, IST Technical University Graz, June 2000. Available from http://www.ist.tu-graz.ac.at.

[42] Pascal Fenkam, Schahram Dustdar, Engin Kirda, Harald Gall, and Gerald Reif. Towards an access control system for mobile peer-to-peer collaborative environments. In *IEEE 11th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2002)*, pages 95–100, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, Jun. 10-12 2002. IEEE Computer Society Press.

[43] Pascal Fenkam, Harald Gall, and Mehdi Jazayeri. Constructing CORBA Supported Oracles: A Case Study in Automated Software Testing. In *Proceedings of the 17th IEEE Automated Software Engineering Conference, Edinburgh, Scotland*, pages 129–138, September 2002.

[44] Pascal Fenkam, Harald Gall, and Mehdi Jazayeri. Visual Requirements Validation: Case Study in a Corba-supported environment. In *Proceedings of the 10th IEEE Joint International Requirements Engineering Conference, Essen, Germany*, pages 81–90, September 2002.

[45] Pascal Fenkam, Harald Gall, and Mehdi Jazayeri. A Systematic Approach to the Development of Event-Based Applications. In *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems (SRDS 2003), Florence, Italy.* IEEE Computer Press, October 2003.

[46] Pascal Fenkam, Harald Gall, and Mehdi Jazayeri. Composing Specifications of Event Based Applications. In *Proceedings of FASE 2003 (Fundamental Approaches to Software Engineering 2003), Warsaw, Poland*, LNCS, pages 67–86. Springer Verlag, April 2003.

[47] Pascal Fenkam, Harald Gall, and Mehdi Jazayeri. Constructing Deadlock Free Event-Based Applications: A Rely/Guarantee Approach. In *Proceedings of FM 2003: the 12th International FME Symposium, Pisa, Italy*, LNCS, pages 632–657. Springer Verlag, September 2003.

[48] Pascal Fenkam, Harald Gall, and Mehdi Jazayeri. Designing an event-based peer-to-peer application for mobile collaboration (submitted for publication). Technical report, DSG, Technical University of Vienna, March 2003.

[49] D. Ferraiolo and R. Kuhn. Role-Based Access Controls. In *Proceedings of 15th NIST-NCSC National Computer Security Conference*, pages 554–563, October 1992.

[50] L. Fiege, G. Muhl, and F. Gartner. A modular approach to building structured event-based systems. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 385–392, Madrid, Spain, 2002. ACM Press.

[51] Ludger Fiege, Mira Mezini, Gero Muhl, and Alejandro P. Buchmann. Engineering event-based systems with scopes. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, volume 2374, pages 309–333. LNCS, 2002.

[52] Wireless Application Protocol Forum. Wireless Application Protocol Service Indication Specification. Technical report, Wireless Application Protocol Forum, November 1999. Available at http://www.wapforum.org/.

[53] N. Francez and A. Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9:133–157, 1978.

[54] Marcelo F. Frias, Gabriel A. Baum, and Thomas S.E. Maibaum. Interpretability of first-order dynamic logic in an extension of fork algebras. In Harrie C. M. de Swart, editor, *Proceedings of the 6th International Conference in Relational Methods in Computer Science(RelMICS 2001)*, volume 2561 of *LNCS*, pages 66–80. Springer Verlag, 2002.

[55] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of Fourth International Symposium of VDM Europe: Formal Software Development Methods*, Noordwijkerhout, Netherlands, October 1991. LNCS 551.

[56] David Garlan and Serge Khersonsky. Model checking implicit-invocation systems. In *Proceedings of the 10th International Workshop on Software Specification and Design, San Diego, CA*, pages 23–30, November 2000.

[57] David Garlan, Serge Khersonsky, and Jung Soo Kim. Model checking publish-subscribe systems. In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software (SPIN 03), Portland, Oregon*, pages 166–180, May 2003.

[58] Andreas Geppert and Dimitrios Tombros. Event-based distributed workflow execution with EVE. Technical Report IFI-96.05, University of Zurich, 20, 1996.

[59] GMD. Xql ipsi, http://xml.darmstadt.gmd.de/xql/, 2002.

[60] gnutella.com. Gnutella, http://www.gnutella.com, 2002.

[61] David Gries. *The Science of Programming.* Springer Verlag, 1981.

[62] David Gries and Gary Levin. Assignment and procedure call proof rules. *ACM Transactions on Programming Languages and Systems*, 2(4):564–579, October 1980.

[63] M. Hauswirth. *Internet-Scale Push Systems for Information Distribution—Architecture, Components, and Communication,*. PhD thesis, Distributed Systems Group, Technical University of Vienna, October 1999.

[64] M. Hauswirth and M. Jazayeri. A component and communication model for push systems. In *Proceedings of the ESEC/FSE 99 - Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, pages 20–38, September 1999.

[65] George T. Heineman and Wiliam T. Councill. *Component-Based Software Engineering, Putting the Pieces Together.* Addison Wesley, 1999.

[66] Matthew Hennessy and Julian Rathke. Bisimulations for a calculus of broadcasting systems. In *International Conference on Concurrency Theory*, pages 486–500, 1995.

[67] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

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

[69] M. Hsu and C.Kleissner. ObjectFlow: Towards a Distributed Process Management Infrastructure. Distributed and Parallel Databases. *Distributed and Parallel Databases*, 4:2, February 1996.

[70] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering Methododlogy*, 11(2):256–290, April 2002.

[71] Daniel Jackson. Automatic analysis of architectural styles. Technical report, MIT Laboratory for Computer Sciences, Software Design Group, Unpublished Manuscript. Available at http://sdg.lcs.mit.edu/ dnj/publications.html.

[72] Michael Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, 1995.

[73] Pankaj Jalote. *An Integrated Approach to Software Engineering.* Springer Verlag, 1997.

[74] Jean Raymond Abrial. *The B-Book; Assigning Programs to Meanings.* Cambridge University Press, 1996.

[75] C. B. Jones. *Systematic software development using VDM.* Prentice-Hall International, 1990. 2nd edition.

[76] C. B. Jones. Wanted: a compositional approach to concurrency. In *Programming Methodology*, pages 1–15. Springer Verlag, 2000.

[77] C.B. Jones. Tentative steps towards a development method for interfering programs. *Transactions on Programming Languages and Systems*, 5(4), October 1983.

[78] G. Kahn. A Preliminary Theory of Parallel Programs . Technical Report TR-98-04, Laboria n 6, IRIA, Le Chesnay, France, 1973.

[79] Engin Kirda, Pascal Fenkam, Gerald Reif, and Harald Gall. A service architecture for mobile teamwork. In *Proceedings of the 14th International Conference on Software Engineering Conference and Knowledge Engineering Ischia, ITALY*, July 2002.

[80] Engin Kirda, Harald Gall, Pascal Fenkam, and Gerald Reif. MOTION: A Peer-to-Peer Platform for Mobile Teamwork Support. In *Cooperative Support for Distributed Software Engineering Processes Workshop, 26th COMPSAC Conference, Oxford, England*, pages 1115–1117. IEEE Computer Society Press, August 2002.

[81] Engin Kirda, Harald Gall, Gerald Reif, Pascal Fenkam, and Clemens Kerer. Supporting mobile users and distributed teamwork. In *Proceedings of ConTEL 2001 – 6th International Conference on Telecommunications*, Zagreb, Croatia, June 13-15 2001, Jun. 2001.

[82] Engin Kirda, Gerald Reif, Harald Gall, and Pascal Fenkam. TWSAPI: A Generic Teamwork Services Application Programming Interface. In *International Workshop on Mobile Teamwork 2002 (Vienna, Austria), 22nd International Conference on Distributed Computing Systems.* IEEE Computer Society Press, July 2002.

[83] B. Krishnamurthy and N.S. Barghouti. Provence: A process visualization and enactment environment. In *Proceedings of 4th European Software Engineering Conference*, pages 451–465, 1993.

[84] Marc Langheinrich, Friedemann Mattern, Kay Rmer, and Harald Vogt. First Steps Towards an Event-Based Infrastructure for Smart Things. In *Ubiquitous Computing Workshop (PACT 2000)*, October 2000.

[85] Lennart Lövstrand. Being selectively aware with the khronika system. In *Proceedings of the 6th European Conference on Computer Supported Cooperative Work-ECSCW'91*, pages 17–31, September 1991.

[86] Maria Manzano. *Extensions of First Order Logic.* Cambridge University Press, 1996.

[87] Ken McCrary. Jtella homepage, http://www.kenmccrary.com/jtella/, 2002.

[88] Rene Meier and Vinny Cahill. Steam: Event-based middleware for wireless ad hoc networks. In *22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02), Vienna, Austria*, pages 639–644, July 2002.

[89] R. Milner. *The Calculus of Communicating Systems.* Prentice Hall, 1993.

[90] R. Milner. *Communicating and Mobile Systems: the pi-Calculus.* Cambridge University Press, May 1999.

[91] MusicCity. Morpheus, http://www.musiccity.com, 2002.

[92] Napster. Napster homepage, http://www.napster.com, 2002.

[93] Peter G. Neumann. Risks to the public in computers and related systems. *ACM SIGSOFT Software Engineering Notes*, 26(1):14–38, 2001.

[94] Leonor Prensa Nieto. The rely-guarantee method in isabelle/hol. In *Proceedings of ESOP 2003*, volume 2618, pages 348–362. Springer Verlag, 2003.

[95] Takahiko Nomura, Koichi Hayashi, Tan Hazama, and Stefan Gudmundson. Interlocus: Workspace configuration mechanisms for activity awareness. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work, Seattle*, pages 19–28, November 1998.

[96] David Notkin, David Garlan, William G. Griswold, and Kevin Sullivan. Adding implicit invocation to languages: Three aproaches. In *Proceedings of JSSST Symp. Object Technologies for Advanced Software*, volume 742, pages 227–233. Springer Verlag, November 1993.

[97] Object Management Group. OMG Formal Documentation. Technical report, OMG, December 1999.

[98] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[99] D. L. Parnas. A Technique for Software Module Specification With Examples. *Communication of the ACM*, 15(5):330–336, May 1972.

[100] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems Into Modules. *Communication of the ACM*, 5(12):1053–1058, December 1972.

[101] Gian Pietro Picco and Gianpaolo Cugola. PeerWare: Core Middleware Support for Peer-To-Peer and Mobile Systems. Technical report, Dipartimento di Electronica e Informazione, Politecnico di Milano, 2001.

[102] Nico Plat and Peter Gorm Larsen. An Overview of the ISO/VDM-SL Standard. In *ACM SIGPLAN Notices*, pages 76–82. ACM SIGPLAN, September 1992.

[103] K. Prasad. A Calculus of Broadcasting Systems. In S. Abramski and T. Maibaum, editors, *Proceedings of TAPSOFT'91*, volume 493, pages 338–358, Brighton, UK, 1991. Springer-Verlag, Berlin.

[104] K. Prasad. A calculus of value broadcasts. In *Parallel Architectures and Languages Europe*, pages 391–402, 1993.

[105] K. Prasad. Programming with broadcasts. In *International Conference on Concurrency Theory*, pages 173–187, 1993.

[106] K. Prasad. Broadcasting with priority. In *European Symposium on Programming*, pages 469–484, 1994.

[107] Wolfgang Prinz. Nessie: An awareness environment for collaborative settings. In *Proceedings of the 6th European Conference on Computer Supported Cooperative Work-ECSCW'99*, pages 391–410, September 1999.

[108] J. M. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.

[109] Gerald Reif, Engin Kirda, Harald Gall, Gian Pietro Picco, Gianpaola Cugola, and Pascal Fenkam. A web-based peer-to-peer architecture for collaborative nomadic working. In *10th IEEE Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), Boston, MA, USA*, pages 334–339. IEEE Computer Society Press, June 2001.

[110] S. P. Reiss. Connecting tools using message passing in the field program development environment. *IEEE Software*, 19(5):57–66, July 1990.

[111] Ed Roman, Scott W. Ambler, and Tyler Jewell. *Mastering Enterprise JavaBeans.* John Wiley & Sons, second edition, 2002.

[112] J. Rothfeder. It's late, costly, incompetent, but try firing a computer system. *Business Week*, pages 164–165, November 1998.

[113] R. S. Sandhu. The schematic protection model: Its definition and analysis for acyclic attenuating schemes. *Journal of the ACM, 35(2):404-432*, April 1988.

[114] K. O. Sandor and A. Schmer. Supporting social awareness @ work, design and experience. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work, Boston*, 1996.

[115] K. Sere and R. J. R. Back. From action systems to modular systems. In *Proceddings of FME'94: Industrial Benefit of Formal Methods*, pages 1–25. Springer-Verlag, 1994.

[116] Kaisa Sere. Procedures and atomicity refinement. *Information Processing Letters*, 60(2):67–74, 1996.

[117] D. N. Smith. *Concepts of Object-Oriented Programming.* McGraw-Hill, 1991.

[118] N. Soundararajan. A proof technique for parallel programs. *Theoretical Computer Science*, 31:13–29, 1984.

[119] C. Stirling. A generalization of owicki-gries's hoare logic for a concurrent while language. *Theoretical Computer Science*, 58(1-3):347–359, 1988.

[120] Ketil Stølen. *Development of Parallel Programs on Shared Data-Structures.* PhD thesis, Department of Computer Science, University of Manchester, 1990.

[121] Ketil Stølen. A Method for the Development of Totally Correct Shared-State Parallel Programs. In *Proceedings of CONCUR'91*, pages 510–525. Springer Verlag, 1991.

[122] Ketil Stølen. An Attempt to Reason about Shared-State Concurrency in the Style of VDM. In *Proceedings of VDM'91*, pages 510–525. Springer Verlag, 1991.

[123] Sun Microsystem Inc. The Java Message Service 1.0.2, November 1999. Available from http://www.javasoft.com.

[124] SunSoft. *The ToolTalk Service: An Inter-operability Solution.* Prentice-Hall, 1993.

[125] Peter Sutton, Rhys Arkins, and Bill Segall. Supporting disconnectedness-transparent information delivery for mobile and invisible computing. In *Proceedings of 2001 IEEE International symposium on Cluster Computing and the Grid*, page 277, May 2001.

[126] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison Wesley, 1999.

[127] The Eclipse Consortium. The eclipse project. URL: http://www.eclipse.org.

[128] The Institute for Apply Computer Science, IFAD. *The IFAD VDM Toolbox.* IFAD Danemark, 1999. Available from www.ifad.dk.

[129] The RAISE Language Group. *The RAISE Specification Language.* Prentice Hall, 1992. ISBN: 0-13-752833-7.

[130] The VDM Tool Group and The Institute of Applied Computer Science. *The IFAD VDM-SL Language.* IFAD, December 1996. Available from http://www.ifad.dk.

[131] TIBCO Software Inc. TIB/Rendezvous TX Concepts Release 1.1. Technical report, TIBCO Software Inc.,Palo Alto, CA, November 2002. http://www.tibco.com.

[132] Dimitris Tombros, Andreas Geppert, and Klaus R. Dittrich. Semantics of reactive components in event-driven workflow execution. In *Proceedings of the Conference on Advanced Information Systems Engineering*, pages 409–422, 1997.

[133] Mark Weiser. Some computer science issues in ubiquitous computing. In *Proceedings of CACM*, pages 74–84, July 1993.

[134] J. M. Wing. *A specifiers Introduction to Formal Methods.* Prentice-Hall International, 1990.

[135] Q. Xu, W.-P. de Roever, and J. He. The rely guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9:149–174, 1997.

[136] Q. Xu and J. He. A theory of state-based parallel programming by refinement: part 1. In J. Morris and R. Shaw, editors, *Proceedings of the 4th BCS-FACS Refinement Workshop*, pages 326–359. Springer Verlag, 1991.