Impact of Mobility on Information Systems and Information System Design

Vom Fachbereich für Mathematik und Informatik
der Technischen Universität Braunschweig

genehmigte Dissertation

zur Erlangung des Grades eines
Doktor-Ingenieurs (Dr.-Ing.)

von

Dipl.-Inform. Peter Ahlbrecht

1. Referent:	Prof. Dr. H.-D. Ehrich
2. Referent:	Prof. Dr. I. Ramos Salavert
Eingereicht am:	17. September 2004
Mündliche Prüfung am:	10. November 2004

Preface

In the context of this dissertation, information systems are viewed as software systems offering a number of information services based on a typically large data repository. More specifically, an information system is viewed as a reactive, possibly distributed system consisting of one or more large data sources, usually in the form of data bases, plus applications built on top of these.

In most applications, information systems are stationary, not only in the sense that the databases are bound to fixed locations, but also in the sense that they are accessed from fixed locations, i.e., via computers which are bound to a network and have an invariant position in the topology of that network. A typical example is a travel booking system with access from travel agencies all over the country.

However, a growing number of applications require that access to information can move with the user, and the access conditions and also the expected responses are location dependent. Typical examples are handheld devices giving context dependent travel information, e.g., about local hotels, restaurants, cinemas, theatres, sightseeing sites, hospitals and other places of interest. In these cases, the user requests access from varying locations, possibly using different hardware devices with different characteristics, and expects that the responses to his or her queries depend on the location where the query is issued.

The present dissertation investigates the impact of mobility on the design of information systems. While this issue has been addressed before, the salient feature of this approach is to view an information system as consisting of both mobile clients and stationary servers. The mobile clients migrate from server to server, coping with the different contexts they find. This requires some local intelligence on the side of the mobile parts. Here, ideas from agent technology are adopted to equip the mobile agents with the necessary versatility.

Another salient feature of the dissertation is its rigidity and formality. In the tradition of the TROLL approach to information system design, much attention has been given to a formal underpinning of the modelling and specification concepts. The innovative contribution here is to provide formal means for expressing mobility.

Altogether, the present thesis makes a valuable contribution to a timely problem. I wish it the attention it deserves.

Hans-Dieter Ehrich Braunschweig, November 2004

Acknowledgements

This is a doctoral thesis submitted to the Department of Mathematics and Computer Science of the Technical University Braunschweig in partial fulfilment of the degree of a Doktor-Ingenieur (Dr. Ing.).

The work has been carried out at the Institute of Information Systems. First and foremost, I would therefore like to thank the Head of this Institute, Prof. Dr. Hans-Dieter Ehrich, for letting me join his group. Besides all his guidance and counsel, he provided me with scientific freedom where possible and set clear borders, e.g. in projects, where necessary. Furthermore, I would like to express my gratitude to Prof. Dr. Isidro Ramos Salavert from the Technical University of Valencia for acting as the second Referee for my thesis.

I am also grateful to all the past and present group members I had the pleasure to work with, namely Gabi Becker-Würch, Jutta Bleiß, Regine Dalkiran, Christiane Eberhardt-Herr, Dr. Silke Eckstein, Dr. Antonio Grau, Maik Kollmann, Andreas Kupfer, Dr. Juliana Küster Filipe, Thomas Mack, Brigitte Mathiak, PD Dr. Karl Neumann, Dr. Ralf Pinger, Andreas Schoolmann, Claudia Täubner, and PD Dr. Jörg Weimar, for constantly providing such a nice and productive working atmosphere.

Special thanks go to Karl for always being there to have a critical look at my work and making valuable comments and suggestions about it, and to Silke, who patiently found time to answer all my questions regarding TROLL, and who also provided the right arguments concerning my work at the right time.

Along the professional lines, I would also like to acknowledge Marcus Tiedemann for his work on the ODiMoD system.

Personally, I wish to thank my family and friends, especially Kristin and Kathrin, for reminding me that there is indeed life outside a thesis, but also for being patient with me when I was focussed on my work. Furthermore, I am immensely grateful for the love and support my parents and grandparents have given me over the years. And last but not least, here a particular "thank you" goes to my father, who took up the tedious work of proof-reading the thesis.

Peter Ahlbrecht Braunschweig, November 2004

Abstract

From the late 1970s on, mobile software and hardware became a topic in computer science, introducing a set of features to traditional and distributed computer systems comprising issues like restricted resources, access rights, and varying connectivity. These features also apply to information systems—some with particular severity, like the resource poverty mismatching the large amount of data which may be delivered by them—and they furthermore add their own particularities like transaction processing or data replication, which have to be revised if mobile units are to be used.

In parallel to the increasing use of mobile hardware and software, several specification languages for mobile systems have been devised, targeted at various areas like mobile agents or the development of communication protocols, and emphasising different aspects like security issues or the failure of nodes.

In this context, the subject of this thesis is to analyse further the impact of mobile hardware and software on information systems, to survey existing approaches for specifying mobile systems of computer science in general, and to provide suitable means for the formal design of information systems comprising such mobile units in particular.

We consider a *mobile unit* to denote a mobile hardware or software entity, and a *mobile system* as a system comprising or being accessed by such mobile components. The various forms of mobile units occurring in computer science are explained and a taxonomy for them is developed, followed by a detailed discussion of their effects on computer and information systems.

Several approaches for specifying mobile systems are presented and classified, with a particular emphasis on formal methods. As it turns out, these approaches do not allow to describe the set-up and release of communication links or to distinguish between the ever-mobile units of a compound system and those which provide the fixed subsystem as the context for the mobile entities, which are both important aspects to consider when developing information systems with mobile components. Therefore, corresponding constructs are then presented as an extension to the specification language TROLL and its theoretical foundations, i.e. extended data signatures and the Module Distributed Temporal Logic MDTL, both being interpreted over event structures. Finally, the application of the constructs is illustrated with the development of a system for accessing web services from mobile phones, which complements the ongoing example of information retrieval via mobile agents used to explain the constructs and concepts.

Zusammenfassung

Mobile Soft- und Hardware beschäftigt die Informatik seit Ende der 1970er Jahre, da in diesem Bereich eine Anzahl von Eigenheiten, die Punkte wie eingeschränkte Ressourcen, Zugangsrechte und veränderliche Konnektivität umfasst, zu berücksichtigen sind, die in herkömmlichen und verteilten Systemen kaum eine Rolle spielen. Diese Eigenheiten treffen auch auf Informationssysteme zu, und dies teilweise sogar besonders gravierend, wie beispielsweise durch das Missverhältnis zwischen der Ressourcenknappheit und den großen Datenmengen, die von ihnen geliefert werden können. Außerdem werden durch Informationssysteme selbst weitere Besonderheiten, z. B. Transaktionsverwaltung oder Datenreplikation, eingeführt, die vor dem Hintergrund der Verwendung mobiler Einheiten überarbeitet werden müssen.

Parallel zu der zunehmenden Verbreitung mobiler Hard- und Software wurden diverse Spezifikationssprachen für mobile Systeme ausgearbeitet, die auf verschiedene Anwendungsgebiete, wie beispielsweise mobile Agenten oder die Entwicklung von Kommunikationsprotokollen, und unterschiedliche Aspekte, z. B. Fragen der Sicherheit oder der von Knotenausfällen, ausgerichtet sind.

Vor diesem Hintergrund sollen in der vorliegenden Arbeit die Auswirkungen von mobiler Hard- und Software auf Informationssysteme detailliert erörtert werden sowie vorhandene Ansätze zur Spezifikation mobiler Systeme in der Informatik allgemein und für den formalen Entwurf von Informationssystemen mit mobilen Einheiten insbesondere vorgestellt werden.

Mobile Einheit wird dabei als Oberbegriff für mobile Hardware- und Softwarekomponenten verwendet, und ein *mobiles System* ist ein System, das solche mobilen Komponenten beinhaltet oder auf das durch diese zugegriffen wird. Wir beschreiben die verschiedenen Formen, in denen mobile Einheiten in der Informatik auftreten, und entwickeln eine entsprechende Taxonomie, bevor wir deren Auswirkungen auf Computer- und Informationssysteme ausführlich diskutieren.

Verschiedene Ansätze zur Spezifikation mobiler Systeme werden vorgestellt und eingeordnet, wobei das Augenmerk speziell auf formalen Methoden liegt. Es stellt sich heraus, dass es keiner dieser Ansätze ermöglicht, den Auf- und Abbau von Kommunikationsverbindungen zu beschreiben und zwischen den stets mobilen Einheiten und denjenigen zu unterscheiden, die das feste Teilsystem als Kontext für sie bilden. Beides sind aber wesentliche Aspekte, die in der Entwicklung von Informationssystemen mit mobilen Bestandteilen zu berücksichtigen sind. Daher stellen wir dann entsprechende Sprachkonstrukte als Erweiterung der Spezifikationssprache TROLL inklusive der formalen Grundlagen vor. Diese Grundlagen beruhen auf erweiterten Datensignaturen und einer modularen verteilten temporalen Logik MDTL, die beide über Ereignisstrukturen interpretiert werden. Schließlich wird die Verwendbarkeit der Sprachkonstrukte in der Entwicklung eines Systems zur Nutzung von Web-Diensten von Mobiltelefonen aus illustriert.

Contents

Chapter 1

Introduction

Mobility has been a topic in computer science from the late 1970s on, where it first occurred in the form of process migration at the operating system layer. In the 1980s, mobile computers became common, but popularity of mobile hardware increased even more in the 1990s with the boom of wireless hand-held devices such as mobile phones and palm tops. At the same time, mobile software vastly increased in the form mobile agent and mobile object technology.

In developing systems comprising mobile units, a set of issues has to be considered which in traditional and distributed computer systems can usually be neglected. Some examples are: Mobile hardware and software is resource poor compared to stationary systems. Disconnections occur frequently and can usually be anticipated. With frequent dis- and reconnections, the logical structure of a system may change drastically.

The technical achievements in mobile hardware and software also effect information systems, and the issues introduced by mobility pertain to them in the same way or affect them even more. Methods devised to keep replicated data consistent frequently rely on an underlying logical structure like a ring, tree, or grid. With mobile units, this structure may constantly change. Transaction processing for distributed databases systems and centralised databases being accessed by mobile clients has to be reviewed. The amount of information resulting for a query may conflict with the limited resources available at a mobile application, yet mobile objects and mobile agents facilitate remote evaluation of queries to information systems and they also allow for dynamic load balancing.

With the proliferation of mobile hardware and software some languages and methods have been devised alongside which provide means for specifying and reasoning about such systems. However, so far their focus seems to be the general area of mobile systems.

As already the development of centralised information systems is a complex task, which becomes even more severe when special issues such as those imposed by mobility should additionally be considered, the demand for a thorough specification preceding its implementation is undisputed; a prerequisite here is of course that the specification language provides adequate concepts. With regard to thorough specifications, the demand for a rigorous or formal design is frequently

raised, motivating the use of a formal specification language, i.e. one with a formal grammar and semantics. The use of such formal languages leads to specifications which not only permit checking the structure of the design documents with ease, but also allow for their unambiguous interpretation.

In this context, the subject of this thesis is to analyse further the impact of mobile hardware and software on information systems, to survey existing approaches for specifying mobile systems of computer science in general, and to provide suitable means for the formal design of information systems comprising such mobile units in particular.

In **chapter 2** we therefore start with a brief survey of mobility in hardware and software systems, give an overview on the main effects of mobility on computer systems in general, and discuss the special impact of mobility on information systems.

Chapter 3 gives an overview on specification techniques for mobile systems in computer science in general. As it turns out, none of the formal approaches discussed there provides adequate means to express the particularities which occur in information systems with mobile components—like temporary disconnection or mismatching resources between mobile and stationary units—and **chapter 4** and **5** therefore give an introduction to the object-oriented specification language TROLL and its theoretical foundations, which will be used as the basis to develop coherent and formal language constructs for specifying such systems.

The main part of this thesis is **chapter 6**. Language constructs to express location and migration of objects and subsystems, to distinguish between the mobile and stationary parts of a compound systems, and to express the set-up and release of connections will be presented here in a unified way and at a high level of abstraction. It will be complemented by **chapter 7**, which illustrates the use of these constructs in the development of an application for accessing web services from mobile phones.

Finally, **chapter 8** summarises this thesis and gives some perspectives on future research.

Chapter 2

Mobility in Computer and Information Systems

The purpose of this thesis is to investigate the impact of mobile hardware and software on information systems and to provide proper means for the specification of information systems which contain such mobile elements. As a prerequisite, we will therefore first explain our understanding of mobility, where it occurs, and what its effects are on computer systems in general.

In this chapter, we start with a survey on where in computer science mobile entities emerge and use this to derive our comprehension of mobility. Next, we define the term information system and conclude with a discussion of the effects of mobility on computer systems in general and information systems in particular.

2.1 Mobile Entities in Computer Science

Mobile Processes

According to [MDW99], the first mobile entities in computer science were mobile processes, used in process migration by operating systems in order to achieve goals such as load distribution, load balancing, application concurrency, or fault resilience. A process at the operating systems layer is a program in execution. In order to control processes, the operating system administers information about the state of every single one of them, which is known as the process' context. The context may contain information about the general purpose registers, program counter, condition codes, I/O buffers, file pointers, process identifiers, etc., and in transferring this state information from one processor to another movement of a process can be achieved.

Process migration has been implemented for multiprocessor architectures and for networks of homogeneous as well as heterogeneous machines, for example in MOSIX [BW89, BL98], Amoeba [SZM94], and TUI [SH98], respectively. With regard to another classification, it has been realised for monolithic operating systems such as Sprite [DO91] and Locus [PW95], for kernel architectures like Chorus and Mach [MZDG93], as well as for message-based systems such as the

V-System [TLC85] and Charlotte [AF89]. In addition to systems as those just mentioned, where at least a part of the state of a process to be migrated relies on transfer from kernel to kernel, other approaches like Condor [LS92] and Emerald [JLHB88] depend on facilities realised in user space.

Condor, for example, achieves process migration by creating a core image as an executable file containing the text, data, and stack segments of memory, register contents, and the program counter, but ignoring signals, timers, and memory-mapped files. This executable file is then transfered to the new node, where the process is then re-established from it using standard operating system facilities. As Condor avoids restoring, e.g., register contents and the program counter via machine-dependent assembler routines, it trades speed and completeness for flexibility and portability.

The other operating systems mentioned earlier realise a more complete form of process migration in the sense that for every relevant piece of state information—including, for instance, memory-mapped files and inter-process communication (IPC)—they either transfer this unit to the target node or arrange for forwarding. In Charlotte, for example, reply messages as part of the IPC are frozen with the process. In the V-System they are simply discarded, relying on retransmission, while in Amoeba the process is first allowed to run to completion before being frozen. Some of these approaches also strive to minimise the time for which a process is unavailable either by sending memory in advance, repeating copying for intermediately modified pages (pre-copying in the V-System), or by transferring memory on demand belatedly (lazy copying in Accent [Zay87] and Sprite). Unsurprisingly, systems which support these more complex forms of process migration encountered more difficulties in coping with process contexts, an experience parallelled in mobile agent systems supporting strong migration [JRS95, RPV95, PS97] described in the next section.

Mobile Agents, Mobile Objects

The term "mobile agent" roots in two strands of computer science [MRK96, CGH+97, MDW99, Gei01, PW01]: "mobile" is related to the area of process migration discussed in the previous section, and "agent" has widely been used in Artificial Intelligence (AI) [WJ95, FG97]. There, compared to the paradigm of object-orientation the agent concept is used to denote a higher level of abstraction, comprising qualities like autonomy, proactivity, intentionality (having beliefs, desires, goals, etc.), and the ability to learn [Yu01]. However, as there is still no commonly accepted definition of "agent" [GS01, PPW02], we will not dwell upon these aspects and merely use them to delimit our perception of mobile agents against other mobile entities when appropriate.

We adopt an understanding of mobile agents similar to the one in [LO98]: A mobile agent is a software object at the application level that has control over its own actions, is proactive, requires an execution environment, may sense its environment, and can move from one location to another. Non-mobile agents lack this last feature. In combination with the ability to move, control over its

own action (autonomy) can be used to distinguish mobile agents from mobile processes, and the existing specification of an algorithm within the entity's code (proactivity) to support migration decisions may be suited to discern them from mobile objects [Gei01]. Other criteria mentioned for separating mobile objects from mobile agents are that due to the required autonomy the latter will tend to form larger units in terms of the size of the constituting code, and that due to the cooperation with the hosting environment and other agents they will have to comply with a more restrictive API [Nel99]. These borders are not clear-cut [JLHB88, BHRS98], but with regard to this thesis a strict separation is not necessary either, and we will therefore refrain from a more thorough distinction.

Mobile agents ultimately need a runtime environment, which usually comes in the form of an interpreter, as this facilitates using different underlying hardware platforms and operating systems [MDW99]. On top of them, hosts conforming to some agent API provide the direct infrastructure. They enable the creation, execution, and migration of and communication between agents and other hosts. In order to facilitate a broad use of agent technology, standardisation efforts by the Object Management Group (OMG) and the Foundation for Intelligent Physical Agents (FIPA) regarding the infrastructure and provided actions have led to the *Mobile Agent System Interoperability Facility (MASIF)* specification [MAS] and the *FIPA* set of standards [FIP], respectively, where the latter is concerned with agent-related topics in the wider (AI) sense (including migration), while the former focuses on mobile agents in particular [PPW02].

Useful and frequently found are logical groupings, which on the one hand may subdivide a host, for example into several "places" or "contexts" in the Mole [BHRS98] and Ara [PS97], or Aglets [LO98] mobile agent system, respectively. On the other hand, they may also extend across several hosts as well as even across different agent systems, forming regions of authority [KRSW01, MAS]. While the *authority* of an agent denotes the "individual or organization in the physical world that it represents" [Whi96], *permits* [Whi96]—also termed *allowances* [PS97], *permissions* [LO98], etc.—regulate which of a system's resources, ranging from CPU time and disk space to the right to create other agents, are available to an agent or place, and to what extend. These resources may be made available to an agent directly via the host, or via dedicated and then usually *stationary* system agents [MRK96, Whi96, CGH+97, KGN+97, LO98].

For mobile agents, a difference is often made between *strong* and *weak migration*. The former is used to indicate that the agent even retains its execution state when arriving at the target host. The latter signifies that only its operations and data will be identical, but execution has to start all over again from a predefined starting point, possibly governed by data values of the agent's attributes. For Aglet applications, e.g., the "onArrival(...)" method of an agent is called every time it arrives at a new host. This method can be overridden in order to customise the agent's behaviour, and in the overriding version attribute values can be examined and used to decide about further actions, but other entry points besides "onArrival(...)" cannot be used [LO98]. In contrast to this, with systems supporting strong migration like D'Agents [KGN+97] (the former Agent

TCL) or Ara [PS97] it is possible to continue execution from the point directly after the migration instruction.

Extracting the agent's execution state already poses a serious burden on mobile agent systems with strong migration, since the status of system-dependent threads or processes have to be captured. As already mentioned at the end of the last section, usage of other stationary system resources like files or network access complicate strong migration even further, leading, e.g., in TACOMA to the decision to leave the task of capturing this information up to the agent programmer [JRS99]. Both [BHRS98] and [Gei01] discuss several platforms for mobile agents distinguishing between weak and strong migration, while [AGE], [CET] and [MAL] provide more complete lists on mobile agent systems in general.

Frequently mentioned benefits of mobile agents are: they allow asynchronous and autonomous processing of tasks; they can reduce network latency and bandwidth usage by locally reacting to events and filtering data, respectively; they increase the fault tolerance in distributed systems, as they may leave nodes which are about to fail or being shut down and avoid migrating to nodes which are out of operation; and they also increase the flexibility in the development and deployment of distributed applications, as the agent hosts provide a homogeneous environment on heterogeneous hardware and operating systems, where functionality can be added or exchanged and re-deployed as needed [CHK97, FPV98, LO99, MDW99, Gei01, SCH01]. On the downside, issues are listed such as that they require a special infrastructure (1.) in the form of hosts and (2.) those hosts in a sufficient number; that compared to distributed applications security imposes additional and more serious challenges, as hosts have to be protected against mobile agents, agents against hosts and other agents, and both against third parties and vice versa; and that the beneficial use of mobile agents for reducing network bandwidth consumption may be lost if the size of the agent's code or the amount of messages exchanged by them becomes too large [ibid.]. Useful application areas for mobile agents therefore include information retrieval [Joh98, BGM+99, IH99, KSM+99, GKP+01], monitoring [SM99, PW01], software distribution [Nel99, PW01] and adjustment of distributed applications [PW01, SCH01, AR02], as well as asynchronous, autonomous processing, for example in e-commerce [Whi96, SCH01] and from or in combination with mobile hardware [CGH+97, KGN+97, SHB+99, SM99, KP99], which will be the focus of the next section.

Mobile Hardware

Mobile hardware comes in a large variety of different types of devices, ranging from tiny sensors to on-board computers in moving vehicles and various categories of portable equipment [IK96, Per01, Sto02]. These devices may already provide useful services on their own; however, frequently they also provide means to connect to other mobile or stationary systems, thereby increasing the available facilities. And as the types of devices vary, so do the means for wired and wireless connections, which again influence the services offered to the user.

The speed with which the processing capabilities of mobile computers advance and new types of devices occur renders a classification according to technical dimensions like CPU speed, main and secondary storage, etc., to be of little use. In order to provide an overview on mobile hardware, we will therefore instead follow the categorisation given in [Rot02] based on qualitative issues.

[Rot02] distinguishes on the one hand between five categories of mobile terminals, mainly based on their physical size, and on the other hand on whether they are general- or special-purpose devices. The five categories are: standard mobile computers, on-board computers, hand-held devices, wearables, and chip cards. Standard mobile computers provide almost the same functionalities as conventional stationary PCs. Distinguishing features are usually found at the I/O units, e.g. special arrangements of the keys on the keyboard and build-in pointing devices and displays. While the transition from mobile precursors like "luggable" computers to laptops can be fixed to the occurrence of liquid crystal displays, the border between laptop, notebook and subnotebook computers is somewhat arbitrarily based on size and weight [RRH00]. Compared to them, hand-held devices, as the name implies, are small enough to be held in one hand (and operated with the other). Wearables are either directly attached to the body, or integrated into clothing. Chip cards provide storage and processing capabilities, may be programmable, and require a reading device to be accessible. Passive radio frequency identification tags [Sta03] fall into the same category. Finally, on-board computers are specialised hardware permanently mounted to a car, aeroplane, ship, etc. Mobile robots may also be subsumed under this category, with the special feature that here the on-board computer may autonomously steer the hosting hardware. Examples of special-purpose devices for the other categories are notebook computers in particular application areas like survey engineering or cartography; e-books, pagers, and gps-receivers as specialised hand-held devices; pulse meters and wrist watches as dedicated wearables; and SIM cards and cards for chip-based electronic purse systems as special chip cards. Well-known general-purpose devices are notebooks and PDAs, programmable wearables like wrist PDAs [Sti02], and smart cards.

As mobile devices are being produced by individual companies, they evolve at high speed. In contrast to this, the development of network technology to support their connectivity takes more time, as usually some standard has to be agreed upon and often an appropriate infrastructure needs to be set up. Mobile devices may use both wired and wireless connections. The main differences between the two types of networks are that for the latter all users basically share the same transmission channel, and that the medium is more susceptible to distortion, e.g. different types of fading for radio frequency and microwave transmission [IK96, Goo97], dependency on line-of-sight links for infrared and light wave usage, and sensitivity to rain, snow, or air turbulences [Tan96, FRHF03]. Compared to this, wires can easily be shielded against environmental influences, and the number of channels can be increased by using separate cables [Rot02]. As the electromagnetic spectrum above visible light is harmful to living things and difficult to handle, and national and international agreements put additional restrictions

Network	Max. Data Rate	Coverage Area	
Bluetooth	1 MBit/s	$\leq$ 10 m	"Wireless
IrDA			Personal
- SIR	115,2 kBit/s	$\geq$ 0.2 m	Area
- FIR	4 MBit/s	$\geq$ 1 m	Networks"
- VFIR	16 MBit/s	$\geq$ 1 m	
IEEE 802.11	2 MBit/s	30–300 m	"Wireless Local
IEEE 802.11b	11 MBit/s	30–300 m	Area Networks"
Wireless ATM	25 MBit/s	30–300 m	
GSM		0.5–35 km	
- plain	9.6 kBit/s	(DCS1800: 0.2–8 km)	Mobile Phone
- GPRS	171.2 kBit/s		Systems
- HSCSD	115.2 kBit/s		
- EDGE	473.6 kBit/s		
UMTS			
- Pico Cell	2 MBit/s	$\leq$ 50 m	
- Micro Cell	384 kBit/s	$\leq$ 10 km	
- Macro Cell	144 kBit/s		
- Satellite Cell	144–384 kBit/s	$\leq$ 1000 km	
IEEE 802.3	10/100/1000 MBit/s	*not applicable*	"Ethernet"
ATM	155/622 MBit/s	*not applicable*	

Table 2.1: Maximum data rates and coverage areas for selected wired and wireless networks. Even very fast infrared (VFIR) and future cellular systems (UMTS) transmit at orders of magnitude below current wireline technology.

on the usage of the lower frequencies, the susceptibility to distortions and the by orders of magnitude lower data rate for wireless transmission (cf. table 2.1) is likely to remain. Frequency reuse, achieved in terms of space, time, or the application of special encoding schemes (space/time/code division multiple access, modulation technology) produces some relief. However, even though these access methods increase the available bandwidth, they, too, have their restrictions: reducing the cell size for cellular phones leads to a higher number of handovers, complicating their administration; using higher modulation techniques increases bit error rates; etc. [DB96, Rot02].

The quintessence of the last two paragraphs is that the resources of as well as the connections for mobile devices vary to a great extend, but are usually poor compared to stationary devices. This also poses new demands on software to be run on mobile hardware [FZ94]. For example, limited connectivity and restricted storage seriously effect file and database access and consistency [MDW99]. But before we turn to information systems and the impact of mobility on them and on computer systems in general for the rest of the chapter, we should summarise this section by stating that mobile entities in computer science occur as pure software

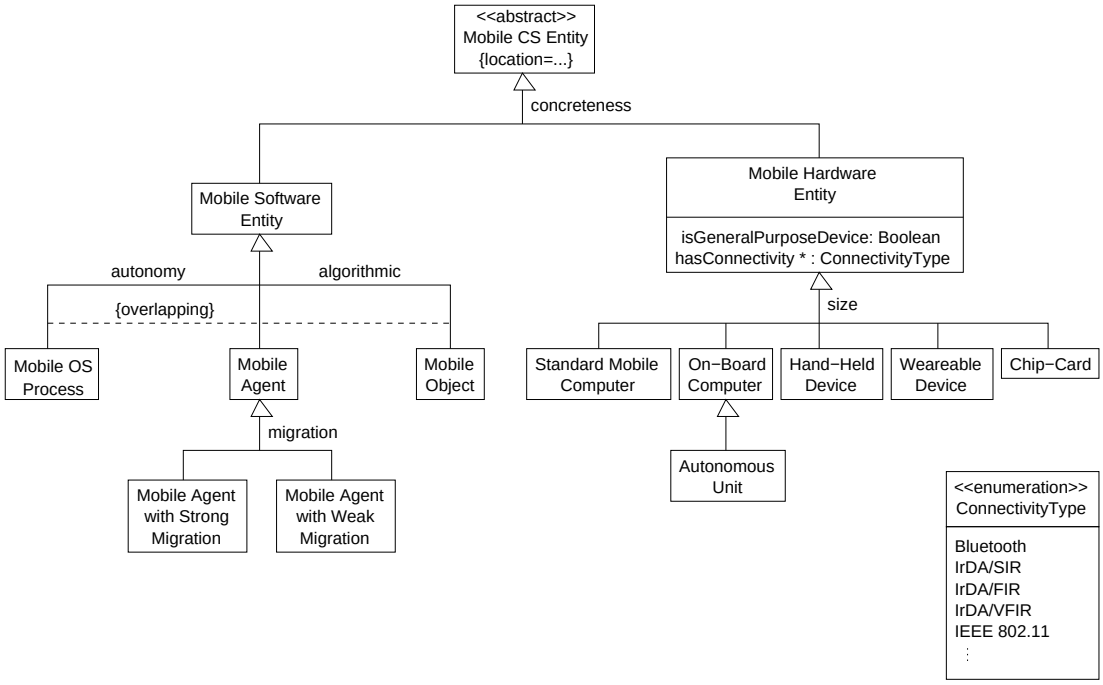


Figure 2.1: Taxonomy of mobile entities in computer science as a UML class diagram

units—which then require a network to be able to migrate—or as hardware, which may or may not be connected to other devices, but in both cases influences the software running on it. Figure 2.1 depicts a taxonomy of mobile entities in computer science according to the discussion in this section as a UML class diagram.

2.2 Information Systems

In the literature, the term *information system* is used in various ways, ranging from a synonym for "data processing system" [RRH00] to the "resources that enable the collection, management, control and dissemination of information throughout an organization" [CBS96], including software, hardware, storage media, and the personnel who use and manage the data [EN00], or even to any kind of computer system whose elements are specialised in the exchange of information through time or space [ALSS03]. One issue not mentioned in these definitions, but frequently found in others, is that the information is usually kept in a database administered by a database management system [RC93, RRH00], which compared to file-based storage provides advantages like uniform administration of possibly large amounts of data, enforcement of integrity constraints and consistency in general, support for transaction processing and recovery, etc. Being sensitive to the consistency of data, information systems fall into the category of reactive systems.

Information systems can be distributed computing systems, i.e. they may be based upon distributed databases or several different (heterogeneous) databases, and also the application programs may again run separately on other nodes in a net of networked computers. The distribution may be a result of the natural development with the organisational structure of a distributed enterprise or company, or may be a result of a rationale that for current or future use the advantages of a distributed system outweigh the additional difficulties. Benefits commonly associated with distributed systems like sharing of resources, better performance, or improved failure handling [CDK01] also apply to distributed databases. On the downside, however, distribution in general also leads to additional challenges, e.g. dealing with concurrency of and heterogeneity in the connected subsystems or transparency in and scalability of the entire system. Also distribution of information provides further serious issues like query processing, transaction management, recovery, etc., and also raises questions on how to fragment, allocate, and possibly replicate data [OV99, EN00].

We will therefore use the term *information system* to denote a *reactive, possibly distributed system consisting of one or more large sources of data, usually in the form of databases, plus applications built on top of them, facilitating the collection, management, control, and retrieval of information.* We will include aspects like hardware, operating systems, system and non-system software inasmuch as they account for effects of mobility, for example with applications running on mobile hardware, or mobile objects and mobile processes being used in information retrieval or load balancing. In a wider sense, we may relax the assumption that the data is stored in a database governed by a database management system (DBMS), though DBMS-related functionality like consistency of data or support for transaction processing should still be provided, for instance via application programs.

2.3 Impact of Mobility

Mobility means the ability to move or be moved between different locations. Such a migration is usually performed because the point of origin provides some undesired quality or because the arrival point offers some desired feature or better opportunities than the source. In this section we will discuss the effects of mobility on computer systems in general and information systems in particular. By these "effects" we mean a set of features emerging in computer systems due to the use of mobile hardware or software. A feature may affect either only the mobile elements or some other components of a compound system comprising mobile entities, or possibly both.

Section 2.1 showed that mobile units in computer science occur in form of both mobile hardware and software, and section 2.2 already mentioned that distributed systems provide several challenges which require additional consideration compared to centralised systems. Regarding information systems, with their own set of special issues in computer science like consistency and transaction sup-

port, mobility has various effects depending on the combination of these different characteristics: the traits of a distributed information system including mobile hardware partly differ from those of a centralised system being accessed by mobile agents, but also share some commonalities. In the following, we will therefore discuss the effects of mobility on computer systems in general and explain how they evince in information systems with respect to the different settings. We will see that the severity of the effects varies according to the conditions under which they are being used, and that they are partly more serious for the mobile units themselves, partly more for some other components of a larger system containing them.

Restricted Resources

Having to cope with restricted resources is probably one of the most frequently mentioned implications of mobility on computer systems [Chr93, PB94b, MDW99, PS01, BNT02]. In case of mobile hardware, it stems from the request for portability and leads to a demand for little weight and small size, imposing limits on the constituting parts as well as the connectivity of a device.

Batteries, as the largest single source of weight, can of course not be reduced arbitrarily, and relying on them other resources have to be restricted. Clock frequencies get scaled back, resulting in slower CPU speed. Screen lighting may be omitted or used sparingly, or the device may switch to a reduced illumination mode soon after user activity stops; hard drives may get spinned down, leading to latency in disk access [IB94]. The request for robustness and small size also enforces reductions. Input and output capabilities pay their toll to small screens and miniaturised keyboards or pens being used instead of them [FZ94].

If a mobile device relies on a wireless connection for communication purposes, its connectivity is also reduced due to the lower bandwidth of this medium [BMGM94, Wat94]. As the particularities like slow and fast fading are inherent to this transmission path and the countermeasure of using higher transmission power conflicts with the limited source of energy [Rot02], this restriction is likely to remain. Furthermore, the general counter-argument that resources will improve is invalidated by the observation that resources of stationary systems will increase, too, and that miniaturisation will continue, leading to the claim that the *relative* bandwidth and resource poverty will remain [Sat96, JWH97, Rot02].

With regard to information systems, the mismatch between restricted resources and the need to obtain or process a large set of data is obvious. [AK93] and [IB94] point out that the cost of sending information (both monetarily as well as in terms of energy consumption) via a wireless connection and the load of heavy database usage on battery power may be reflected in new query optimisers being developed, and that reduced I/O capabilities should be considered in database development for mobile devices [AHK92, MWC96]. Furthermore, data-intensive applications and distributed systems relying on an unstable communication network with low bandwidth are said to have a special need for replication techniques [BD96], and distributed IS comprising mobile hardware clearly belong

to this category. As a beneficial issue from using wireless transmission, however, the "support for broadcast communication within a 'cell' " [BAI93, IB94] could be utilised [OV99].

Restriction of resources also occurs in case of mobile software. Here, the demand for small size of the mobile unit is of course not compulsory, but the quality of being able to reduce the network load frequently mentioned for mobile agents [CHK97, LO99, TR00, GKP+01] becomes questionable if the amount of code constituting them continues to increase [SM99]. [Gei01] illustrates this aspect by comparing the application of monolithic mobile agents to one where the agent is fragmented into several objects with regard to the different subtasks it has to solve, and develops a mathematical model for it, while [CPV97] details points of trade-off for different mobile code paradigms used in information retrieval and [BP98] and [FPV98] provide a similar analysis for network management applications.

The resources for mobile software may also be restricted due to environmental conditions [SZM94] or policies. Already Java applets, a very basic form of mobile code [Nel99], are not allowed to access the local hard drive or spawn new processes, may only establish a network connection to the server they originated from, have to mark windows opened by them, etc. [Fla97]. For mobile objects/agents, the availability or restriction of resources can often be customised by administering security or resource managers [KGN+97, Pet01, Gro02]. [CHK97] and [KGN+97] also mention the possibility of imposing limits on resource utilisation in a currency-based model. The other way around, the possibility to clone agents and send them out to several places in a broadcast-like fashion as a means to overcome restricted resources is also frequently mentioned [BGM+99, LO99].

When viewing a program which is usually used in a stand-alone way, but which runs on a networked computer, as a non-distributed application, then with the possibility to add new or modify existing functionality by means of swapping code mobile software also effects centralised systems. [Nel99] for example presents a dynamically upgradable text editor, and if a browser is used for accessing local data, then by adding another plug-in its capabilities may be increased in absolute or relative terms—new media may be accessed or known ones can be processed faster. In a similar way the schema of a conventional database could be modified or stored procedures be added.

However, the effects of mobility due to mobile software, including the one of restricted resources, are clearly primarily related to distributed systems. Regarding the limitations on the size—and thus functionality—of mobile code, a beneficial use can be obtained easily if the "application involves analysis of enough information, and enough reduction of the data returned" [GKP+01]. This is highly applicable in distributed information retrieval [Joh98, IH99, KSM+99, PSP99], but also considered in other areas, e.g. network management applications [MRK96, BPW98, SM99]. [MRK96] and [Whi96] also mention the potential of mobile agents to reduce bandwidth consumption in a wired network, and [PSP99] shows that database access via mobile agents can be used to overcome low bandwidth and limited storage capacity of mobile devices.

Varying Resources

Having to cope with varying resources is a related, though different effect introduced by mobility. A device like a notebook or PDA may use a wired connection in one place, WLAN access in another, BlueTooth in a third, and cellular networks of differing bandwidth in others [FZ94, Sat96]. [JWH97] presents a client/server system with a browser to be used on a mobile device, where the user may adjust viewing preferences according to the available network connection, resulting for instance in suitable image compression and file type filtering. As the bandwidth may change even during one connection, e.g. due to fading or when using a GPRS connection, the necessity to provide means for applications to adapt to this hardware-related varying connectivity leads to approaches like "application-aware adaptation" [NSN+97] and "adaptation spaces" [BDM+00]. [NSN+97] for example describes a video player, a web browser, and a speech recogniser implemented on top of Odyssey (a platform for mobile computing [NPS95, Nob98] sometimes also referred to as a mobile agent platform [LO98, MDW99]), which adapt autonomously to changing bandwidth.

Information systems are again particularly affected, e.g. with regard to the amount of data they may produce or the way query optimisation should be done [OV99].

The components available to a device may also vary as it gets moved to different locations. At the office a laser printer with high resolution may be used from a notebook, while on a trip only a small ink-jet printer may be available, if at all. An information system may benefit from certain processing capabilities, e.g. more secondary storage capacity on a shared file system, but is not that likely to be affected.

Regarding mobile software, however, applications may benefit significantly from efficient access to local data and services, as already discussed for restricted resources. The distributed information retrieval application described in [KGN+97] for instance detects whether the local sites to be searched provide a low- or high-level interface to their data, and depending on this knowledge decides to send clones to perform several local queries or use only one remote request, respectively.

The potential to use mobile software to react to varying conditions is of course not limited to information systems [JLHB88, LO98, BGM+99], but compared to this gaining profit from it, e.g. by process migration in reaction to changing processor loads [BW89, AF89, SZM94] or using mobile agents in network management [BPW98], are harder to achieve [ELZ88, MZDG93, SHB+99].

Mobile software may also experience varying connectivity. A mobile agent can migrate from a network with high bandwidth to one with lower (for example from a 1000 Mbps LAN to one with only 10 Mbps), sense this, and reflect it in information sent to other parties or deciding on its next destination. Furthermore, [BP98] shows that using mobile agents (itinerant objects) instead of remote evaluation or downloaded code may decrease the bandwidth consumption around a network management station, as, of course, mobile software affects the

bandwidth in networks in general.

Disconnections

As the extreme case, a connection can be lost completely, though for different reasons. With regard to the use of mobile hardware, disconnection may be voluntary and predictable due to the user powering down his device in order to save energy, and preliminarily announcing to do so; unintended but anticipated, as fading and a battery running low on energy may always occur; or unexpected as genuine failures like dropping a device may also happen. Similar situations may occur for mobile software: a mobile agent/object may decide to migrate behind a firewall, it may reside on a computer which gets powered down, or it may be attacked by another agent or agent host. For this reason, disconnection for mobile units is usually considered to be different from failure [BAI93, PB95, Car99].

On closer examination, three issues are introduced by the possible occurrence of temporary disconnections. The first is the one of intermittent connectivity, i.e. simple dis- and reconnection from a mobile unit at the same access point of another system, subsystem, or network. We will refer to this as *disconnection* in the strict sense. Another issue is the one of *dynamically changing connectivity*, where a mobile entity detaches from one point and attaches at another. Finally, a composed or distributed system might rely on a *logical structure* of its components. If in this case a mobile unit disconnects and possibly later reconnects at a different location, algorithms utilising the logical structure have to be adjusted. This might be more difficult to achieve than adapting the target node in case of dynamically changing connectivity.

Intermittent connectivity can lead to resources becoming unavailable and therefore potentially effects any kind of distributed system. Examples are delivery of any kind of data, from RPCs to email and multimedia streams [ALSS03]. However, regarding information systems some of their particularities are already affected even if the data is not partitioned or replicated and distributed. The response to a query may not reach the mobile unit which posed it—costs for sending might still be high, though—and for transactions started from a mobile entity neither a commit nor an abort may be received [DH95, DK99]. Another difficulty related to disconnection of mobile units is that they may remain detached for a long time. This problem has also been considered in other areas of database development (see e.g. [GMS87]), but is most obvious in this context. More generally speaking, in order to overcome unreachableness of mobile devices or to allow for better performance when being faced with long lived transactions, several transaction models have been suggested which relax some of the ACID properties commonly associated with database transactions. For example, [Chr93, PB94a, DHB97] suggest relaxing atomicity, [CR94, YZ94] give up isolation, and [PB94b, PB95, GHOS96] loosen consistency. In a similar way, locking protocols are also reconsidered [HAA00, KPDS02].

In case of distributed data, transaction management gets more complicated, especially if data also resides on mobile units [PB94a, WC97, HAA00]. Updates

may not reach a single destination, or copies may get out of sync [GHOS96]. Furthermore, configuration management, i.e. mobility of resources in general [IB94], here has to be reviewed in terms of data allocation and replica placement [BI92, HSW94, FZ95, BNT02], and answers to queries now may need to be sent to different locations. Other examples from the general area of computer systems are hand-over techniques for cellular phones or migrations of mobile agents, and traffic telematics [SHB+99, ALSS03].

Distributed systems utilising a logical structure of its constituents are again particularly susceptible to mobile components. [BAI93] illustrates this for the example of a logical ring, which could be the token ring frequently mentioned for distributed systems in general, or the ring of the "majority consensus" replication strategy for distributed database systems [Tho79, Dad96]. As other replication strategies also rely on logical structures, e.g. a tree or a grid [AA92, BD96], the effect of having to deal with changing logical structures is highly applicable to information systems. Once again, the changes may be due to mobile hardware or mobile software: the example in [BAI93] is based on mobile devices attaching to different mobile support stations, and [AR02] presents a system where mobile agents can be used to adjust a federated information system to incorporate new or dismantle old component systems.

Location Management

As a prerequisite to be able to deal with dynamically changing connectivity or logical structures, but also just to facilitate communication between different components, the position of a mobile unit has to be known, which raises the issue of location management, i.e. "the management of transient location information" [Wol02]. [FZ94] discusses four methods to determine a mobile computer's current address: "central services" and "home bases" both rely on the address being stored and administered by a single authority, while "selective broadcast" acquires the position dynamically, and with "forwarding pointers" the mobile computer upon leaving informs the current site about the address of its next location. Similar approaches have been used to locate and manage mobile software entities. [AO98], for example, discusses three schemes for locating mobile agents, where "Logging" closely corresponds to forwarding pointers, "Brute Force" to broadcast, and "Registration" to the central services/home base method. The Emerald system, attributed to the area of process migration [MDW99], uses address forwarding and broadcasting to locate mobile processes and objects [JLHB88], and [BR98] uses chains of agent and shadow proxies in the management of mobile agents. Furthermore, location tracking for mobile agents is even covered in the FIPA [FIP] and MASIF [MAS] standards on mobile agents [PPW02].

Address information might thus be maintained in a logically centralised (first approach) or distributed database (second and fourth method). In GSM mobile telephony, for example, both permanent data of users and dynamic information regarding their position is kept in the central home location register, and temporarily copied to the visitors location register of the mobile support station

for the cell currently visited by the mobile phone [Rot02]. The high frequency with which data on mobile devices may change, however, is one of the challenges information systems have to meet: [Wol02] points out that for the straightforward approach of periodically storing a location/time tuple (l, t), indicating that at time t the object of interest is at location l, would require frequent updates from the mobile device, conflicting with the restricted "resources such as bandwidth and processing power" [ibid.]. Other problems identified when using this approach are that it does not support interpolation or extrapolation and that current DBMS software rarely provides adequate means to handle continuously changing data [KGT99]. Furthermore, even though a lot of effort has been put into extending SQL to provide suitable means for dealing with temporal issues, only little support for managing time- *and* space-related data as well as uncertain information—which might occur when the actual location of an object differs from the value stored in the database—is available [GBE+00, Wol02].

Location Dependent Information

Location dependent information is strongly related to the issue of location management [IB92, SWCD97, Wol02]. Though compared to the latter it is not an impact of mobility that *necessarily* has to be dealt with, the *possibility* to provide the user of a mobile device with specific data in relation to his current position has attracted much attention, especially in the area of information systems.

Location dependent information can be given in an active way when detecting that a mobile device enters a certain area or by broadcasting information. It can also be provided passively when answering queries. Both [IB92] and [SDK01] distinguish between queries for locations where the result set does not change when the place of the issuer changes ("location aware queries" such as "find cinemas in Braunschweig"), and the proper location dependent queries like "which is the nearest hospital" where it does. The latter are the actual challenge to information systems, as the value for the location of the user needed to answer the query has to be determined at runtime, either by processing signals or by utilising stored data, which then brings back the problems identified for location management.

The possibility to *broadcast* data actively is inherent to systems using wireless connections, though it is, of course, not limited to this scenario, as wired networks might also support broadcast addresses. With regard to the limited resources of mobile devices, however, it is of particular importance to information systems [OV99]: instead of having to submit energy consuming queries, information is emitted from stationary servers in regular intervals. This might be data assumed to be of general interest like weather or traffic information, or more user-specific data. Since the bulk of information emitted should be of interest to a wide audience, means to determine which data to send more frequently and when to update or replace a broadcasted item are developed as well as methods that guide receivers to specific, less frequently delivered information [IB94, OV99, PS01, CWY03]

As already mentioned, broadcasting facilities might also be used in wired networks, and for mobile agents cloning and to "send out a wave of child agents to visit multiple machines in parallel" [KGN+97] may be considered as a similar means, but in general broadcasting is more closely related to wireless networks, and location dependent information primarily aims at users of mobile hardware.

Security

In contrast to this, special security issues as our final impact of mobility again pertain to both mobile hardware and software. Mobile devices may be lost, dropped or damaged in another way, or stolen [FZ94, Sat96, PS01]. When being integrated into a new network, they may also pose a threat to the existing environment by providing access to sensitive local information, introducing viruses, overly consuming resources, etc. [Car99]. Information transmitted via a wireless connection is more susceptible to eavesdropping than one transfered on wire.

Similar issues occur for mobile software: a mobile unit may perform malicious acts on the hosting system, but it may also be attacked by the environment [MDW99, Nel99], which in case of being burdened with sensitive information about or for its owner becomes a delicate feature of mobile agents [CGH+97, Hoh97, KGN+97, BHRS98]. For mobile agents, too, due to their broader capabilities compared to plain objects providing inter-agent security poses additional demands, and agents as well as their hosts may be threatened by unauthorised third parties, for example by eavesdropping their communication or swamping the host with "fake" agents or messages, resulting in denial of service or replay attacks [Hoh97, LO98]. However, it is also mentioned that the other way around excessive cloning may charge the network, likewise possibly leading to denial of service. Finally, as another passive attack similar to eavesdropping, analysis of traffic and migration patterns is also mentioned as a possible security risk.

As confidential information may be stored in information systems, they are obviously affected by these security issues. However, they do not seem to pose problems on these systems which would not also apply to other areas, or which have not been considered for them before. The additional security-related difficulties introduced by the mobility of hardware or software thus pertains to computer systems in general.

Table 2.2 summarises our discussion on the impact of mobility on computer systems and indicates the relevance and applicability of the individual effects for certain areas of computer science with regard to the originating mobile unit.

Effect of Mobility	due to **Hardware** in				due to **Software** in			
	Centralised Systems		**Distributed Systems**		**Centralised Systems**		**Distributed Systems**	
	General	**IS-specific**	**General**	**IS-specific**	**General**	**IS-specific**	**General**	**IS-specific**
Restricted resources								
· for components	×	✳	×	✳	+	+	×	✳
· for connectivity								
▷ wireless								
○ bandwidth	×	✳	×	✳	–[1]	–[1]	–[1]	–[1]
○ broadcast	+	+	+	+	–[1]	–[1]	–[1]	–[1]
○ dynamics	–	–	✳	✳	–[1]	–[1]	–[1]	–[1]
▷ wired								
○ dynamics	–	–	✳	✳	–	–	✳	✳
Varying resources								
· for connectivity	×	×	✳	✳	+	+	×	×
· for components	×	+	×	+	+	×	×	✳
Disconnection	×	✳	×	✳	+	×	+	×
Changing logical structure	–	–	×	✳	×	×	×	✳
Location management	×	×	✳	✳	×	×	✳	✳
Location dependent information	+	+	×	×	–	–	–	–
Security issues	✳	✳	✳	✳	✳	✳	✳	✳

Table 2.2: Tabular summary of the effects of mobility on computer systems.
–: effect less significant or not applicable, +: significant/applicable, ×: more significant/applicable, ✳: highly significant/applicable
[1] If used for/from mobile devices cf. corresponding effects on hardware.

2.4 Summary

In this chapter, we first outlined where in computer science mobile units occur, defined the term information system, and then discussed the impact of mobility on computer systems in general and on information systems in particular. We showed that mobile entities appear in software as well as in hardware. As subcategories for mobile software we further distinguished between mobile processes at the operating systems layer, and mobile agents and mobile objects as migrating units at the application level. In order to discuss mobile hardware, we picked up a subclassification according to physical size and general-purpose vs. special-purpose usability, indicating that here usability is particularly affected by the connectivity available to the mobile device.

Within this thesis, information systems are regarded as reactive, possibly distributed systems consisting of one or more large data sources, usually in the form of databases, plus applications built on top of them. Their purpose is to facilitate the collection, management, control, and retrieval of information. They may be affected by the mobility of hardware as well as of software, with varying degree according to the particular set-up. The last section discussed the effects of mobility on them and on computer systems in general, structured according to restricted and varying resources, three issues related to temporary disconnection, location management and location dependent information, and special security issues introduced by mobility.

In the next chapter, we will discuss several approaches for specifying and modelling mobile systems and see how they can be used to capture these effects.

Chapter 3

Modelling and Specifying Mobile Systems

Having shown that mobile entities occur in computer science in various ways, and that their occurrence gives rise to a number of issues which should be reflected in suitable abstractions for the development of systems comprising such mobile units, we will now sift through existing approaches for specifying and modelling mobile systems. The aim of the chapter is to give an overview and a classification of these approaches, and to analyse to what extend they can be used to capture the effects of mobility in particular on information systems.

When reviewing the literature on modelling mobile systems, one soon realises that a large part is based on process algebras and another on the Unified Modeling Language (UML) [BRJ99, RJB99, OMG]. The former is due to the π-calculus [MPW92], a process algebra, being the first means developed for specifying mobile systems in reaction to the first wave of mobile computing technology [Mil01]. The latter may be attributed to the popularity of the UML in recent years.

We will structure our overview accordingly and first look at the basic UML and suggested extensions, then turn to approaches rooting in process algebras, and finally investigate other means which fall in neither of the two categories.

3.1 UML + Extensions

Already the "basic" version of the Unified Modeling Language [BRJ99, HK99] provides several means to model mobile objects with its ≪become≫ and ≪copy≫ stereotype, a tagged value "location", and the possibility to use its "package" construct to express grouping of, e.g., mobile agents moving as a unit [HK99, GM01]. Figure 3.1 shows an adapted version of the example used in [BRJ99] to illustrate the specification of migrating objects. The mobile *TravelAgent* visits the servers of several airlines in order to obtain the cheapest ticket for his owner.

Additionally, several extensions to the UML have been proposed to make the language more suitable for specifying particular aspects of mobile systems, for

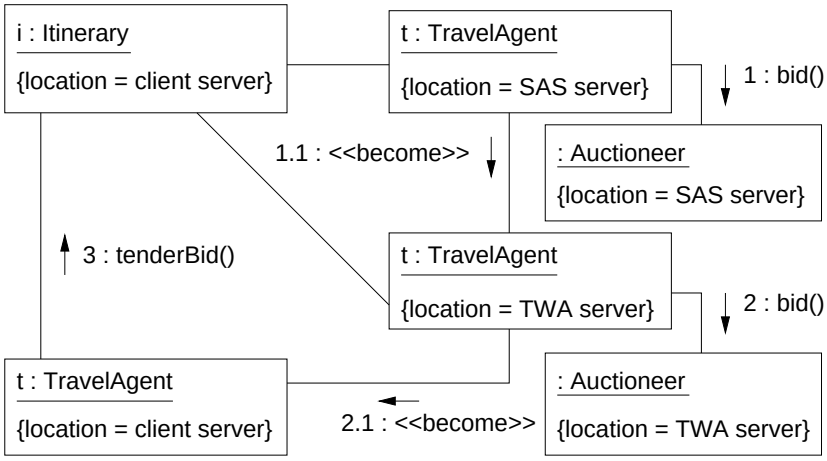


Figure 3.1: Modelling mobile objects in the basic UML: a mobile agent brokering for a flight ticket. Adapted from [BRJ99].

example with regard to mobile agent systems [KRSW01, MG01, Zei03] or mobile components [WMB99, GM01, BKKW03].

The focus of [WMB99] is on modelling dynamic software components, i.e. large-grained building blocks where the membership of subcomponents within components may change over time. The introduced stereotype ≪component≫ characterises a "component link" relationship where a component may be a subcomponent of at most one other component at any time—as for UML composition—but where the superordinate component may vary in the course of time—like in UML aggregations. Other concepts like "component boundary" or "local links" are defined; in order to express migration of (sub-) components, however, the already mentioned ≪become≫ stereotype is used.

In [KRSW01] the development of a quality of service network is used as an example to enrich the UML with several stereotypes, tagged values, and constraints in order to facilitate the design of mobile agent systems, suggesting that the agents can be used to dynamically adjust the routers of the network. Figure 3.2 provides an overview of the added stereotypes together with the corresponding graphical notations, and figure 3.3 illustrates their usage with a part of the specification used in [AR02], where a system has been developed to reconcile data in databases and files via mobile agents. The remaining stereotypes are related to weak migration, role change, region and agent system concepts.

[Zei03] proposes similar additions using stereotypes only, and also details a modelling process for the analysis and design phase of mobile agent systems. In [BKKW03], it has been criticised for not explaining how the extensions fit into the UML metamodel, which also applies to [KRSW01].

In [BKKW03] itself, stereotypes for modelling mobile entities, locations, migration, and cloning are described, which may be used in two variants of UML activity diagrams. One is mainly concerned with the topology of locations, while the other emphasises responsibilities of movements. The stereotypes root in the

Stereotype	Applies to symbol	Meaning	Graphical notation
mobile agent	class	Specifies a mobile agent class; a class, which instances can migrate from the system where they have been created to other systems that also provide an agency.	Agent name : Class
agency	class	Specifies a class that provides an execution environment for mobile agents; a context, where mobile agents can be created, destroyed, activated and deactived, where they can execute and migrate to and from.	Agency name : Class
migrate	dependency	Specifies that the target (B) is the same mobile agent as the source (A), but at a later point in time and a different point in space, possibly with different values, state, or roles.	A B
clone	dependency	Specifies that the target (B) is an exact copy of the source (A), a mobile agent, with the same methods and identical values for its attributes.	A B

Figure 3.2: An extension of the UML for modelling mobile agents.

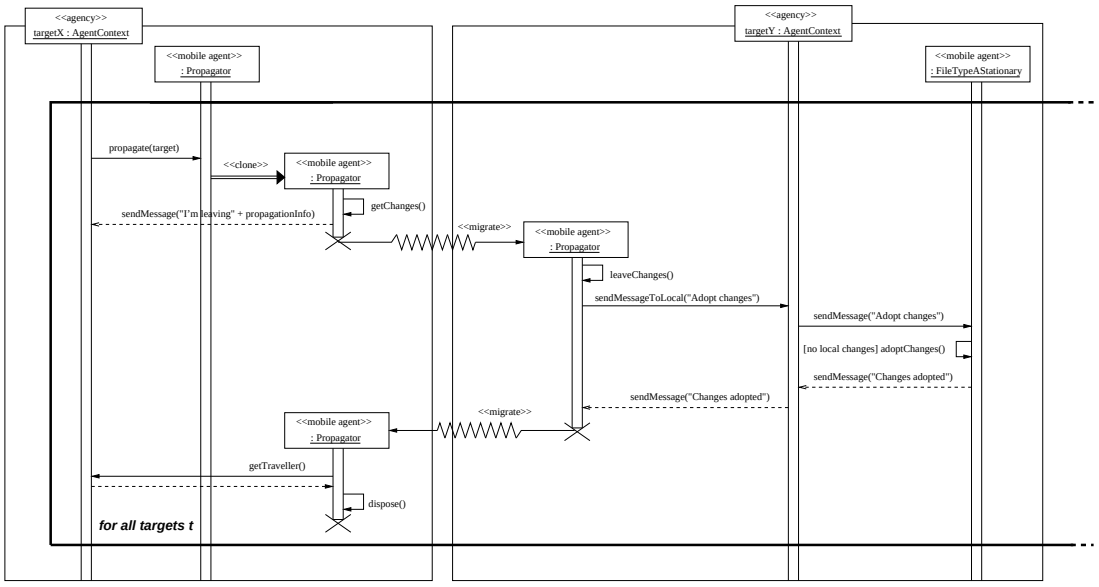


Figure 3.3: Using an extended UML to model mobile-agent-based data integration. Adapted from [AR02].

"atLoc" attribute and relation, which is introduced and anchored in the UML metamodel. Classes and objects bearing the stereotype ≪location≫ are required to provide a value for atLoc, and a change of the value then indicates movement. This potential to move is expressed using the ≪mobile≫ stereotype. Locations may be mobile, too, resulting in a stereotype ≪mobile location≫, which inherits from both classes "location" and "mobile" in the metamodel, and which can then be used to model, e.g., persons boarding and flying a plane.

While the constructs presented in [BKKW03] are not restricted to the development of mobile agent applications, [MG01] again turns back to this particular branch of mobile systems. Much like [KRSW01], it is argued that due to their adaptiveness mobile agent technology already eases the complexity of telecommunication development, but in order to further increase productivity, adequate and reusable specification constructs are required. [MG01] then presents a UML profile with extensions capturing the requirements set up for mobile agent platforms by the MASIF [MAS] specification, where an *Agent* may execute within one of several *Places* run by an *AgentSystem,* the latter providing services via its *CoreAgency* and being grouped into *Regions,* which support agent localisation. In addition to providing corresponding stereotypes, several mobile-agent-specific operations for them like "beforeMove()" and "afterMove()" are also defined, before closing with a discussion on tagged values which can be used to define various kinds of transparencies (location transparency, migration transparency, etc.) in the development of distributed systems.

The aim of [GM01] is to provide a means for evaluating whether or not a mobile code approach should be used in a distributed system instead of client/server interaction. The stereotypes ≪moveTo≫ and ≪moveTo?≫ are introduced for UML collaboration and sequence diagrams, which should be used to indicate migration or *possible* migration, respectively, of objects. Next, an algorithm is presented which can be used to derive several types of Markov Processes from diagrams using these stereotypes, which then support the decision-making.

Finally, in [HKRS02] a framework for modelling ubiquitous web applications is presented based on the rationale that the existing modelling methods are insufficient for the development of personalised, anytime, anywhere, anyplace, and accessible-via-any-medium services. A distinction is made between the variable, context-sensitive part of a web application and its stable part. The framework consists mainly of a customisation model comprising a model for the physical and logical context, and a rule model integrating rules about how an application should react to changes in the context. The logical context model represents information at a higher level of abstraction than the physical context model, for example by distinguishing locations in terms of "AtHome" and "AtWork", while the physical context uses a "cellID"-level of granularity.

As can be seen from this discussion, already in the field of only one (though highly popular) semi-formal specification language a lot of research to capture the impact of mobility on computer systems is being conducted. However, since our focus is on formal approaches we now leave this area and turn to formal

specification languages.

3.2 Process-Algebra-Based Approaches

Another group of related approaches for specifying and modelling mobile systems can be characterised by all being based on process algebras. Here, according to [NFP98, MWZ03] the π-calculus [MPW92, Mil01] was the first formal method for the development of such systems, being an extension of the "initiator of the field of process algebra" [BW90], the Calculus of Communicating Systems (CCS) [Mil80, Mil89]. In reaction to the advent of mobile computing, Milner et al. realised that suitable means to express mobility were lacking, and augmented CCS with a construct to describe the exchange of channels across channels [Car99], which thus allowed to talk of movement of *access* to processes rather than of movement of processes themselves [Mil01]. In the sequel, other researchers identified different key aspects like failure and locality in distributed systems, or security, and suggested corresponding formalisations.

Probably the point most frequently criticised in the π-calculus is that it "lacks a suitable notion of locality" [Car99, VC99, RPM00]. All the extensions discussed in the following therefore added constructs to express location, either in absolute or relative terms. By the former we mean that places are named and can be referenced that way, e.g. in constructs like *moveTo(ℓ)* for a location named ℓ. In the latter approach, locations are arranged in hierarchical structures (like a town consisting of several streets, each street with several houses, each house with several rooms) and movement usually happens along the hierarchy (to visit someone, go from your house into the street, then to the street of the person's house, then enter the house).

Both the enriched π_1-calculus [Ama97] and the distributed π-calculus of [RH97] deal with location in absolute terms. Their focus is on modelling failure in distributed systems, therefore both also provide constructs to test processes at locations for availability, as well as primitives to stop and (re)start them there. With respect to mobility, these primitives suffice to model, e.g., weak migration of mobile agents. The two approaches are quite similar, the main difference being that the distributed π-calculus does not support value-passing [RH97].

The Nomadic π-calculi [SWP99] and the Kernel Language for Agents Interaction and Mobility, KLAIM [NFP98, NFP00], also express location in absolute terms, but while KLAIM also follows the stop/(re)start paradigm to express movements the Nomadic π-calculi use a construct "migrate to s." The focus of the calculi is on communication primitives for interaction between mobile agents. While the low-level Nomadic π-calculus provides location dependent constructs for communication, the high-level one adds a location independent possibility. Sewell at al. argue that location independence may substantially simplify the development of mobile applications, but that it also raises several difficulties like tracking of mobile entities, routing of messages, heterogeneity, etc., and that therefore their twofold approach with a translation mechanism between the cal-

culi seems reasonable [SWP99].

Besides providing support for localities, KLAIM is mainly concerned with coordination models and security, in particular for describing mobile agents and their interaction strategies. It draws on concepts developed in the coordination language Linda [Gel85] with multiple tuple spaces, and operators from CCS [NFP98]. Tuples in Linda consist of values and variables, can be put into a shared global environment called tuple space, and may be retracted from them using pattern-matching. KLAIM modifies one of the access primitives to use processes as arguments rather than tuples, and with this modification permits mobile agents to be programmed [NFP98]. Regarding security, KLAIM allows to associate a set of access capabilities with every process and site, facilitating hierarchies of access rights.

Seal [VC98, VC99] is also concerned with modelling security aspects in mobile computations. It achieves access restriction by distinguishing local and non-local communication between and migration of processes referred to as "seals." The seals are nested in a tree-structure. Local communication and migration are confined to a single level in the hierarchy, and subject to permission by the parent process. Non-local moves and communication require the use of "portals," which are permissions to use a specified channel *only once.* Vitek and Castagna point out that their approach is sensible since control over a process' communication abilities is a key security policy of Java implementations, and also because unrestricted communication and mobility force the underlying architecture to keep track of the respective location of communicants, which for an Internet wide tracking system seems highly questionable [VC98].

The seal approach is similar to mobile ambients [CG98, Car99], with the main difference being that the latter does not provide for non-local migration. Ambients, too, are named places for a collection of processes and subambients, and therefore also express location in relative terms. Movement is restricted to a single step of the location hierarchy at a time, but while in Seal migration can only be initiated by the parent process ("objective move"), in Ambients a process may also initiate its own migration ("subjective move"). Due to the hierarchical structure, in both languages it is also possible to express movement of substructures larger than the unit of specification, the process, which parallels mobile computing and computation when, e.g., all the threads and file systems within a notebook or the parts of a mobile agent migrate together with the enclosing unit [Car99, VC99].

This also holds for the framework built on top of MobileSpaces [Sat00] and the Distributed Join-Calculus [FGL$^+$96]. Both focus on issues related to the development of mobile agent systems. Like the enriched π_1-calculus, the distributed Join-Calculus [FGL$^+$96] also introduces constructs to model failure of a location, while [Sat00] additionally allows to express marshaling.

Table 3.1 summarises which extension of the π-calculus provides language constructs for modelling which issues of mobile systems. More information on calculi for mobile processes can be found at [MOC].

As the last process-algebra-based approach for modelling mobile systems, the

Modelling in	Distr. π [RH97]	Nomadic π [SWP99]	Ambients [CG98]	Distr. Join [FGL+96]	KLAIM [NFP98]	Enriched π_1 [Ama97]	"MobileSpaces" [Sat00]	Seal [VC99]
Explicit location								
- absolute	×	×			×	×		
- relative			×	×			×	×
Mobility via								
- stop/(re)start	×				×	×		
- move		×	×	×			×	×
Failure	×			×		×		
Security					×			×
Move substructures			×				×	×

Table 3.1: Extensions to the π-calculus provide language constructs for modelling various aspects of mobile systems.

Algebra of Itineraries [LSZ99] should be mentioned. Its outstanding feature is that it provides means to model itineraries of movement for arbitrary entities, in a flat structure of places. Unlike all the aforementioned approaches, it does not extend the π-calculus, but also follows algebraic specification principles.

3.3 Other Approaches

As mentioned in the introduction of this chapter, in this section specification techniques for mobile systems will be presented which neither pertain to the UML nor to a process algebra. Most of them are at least related to some logic, but approaches based on petri nets and coordination languages are also included. Regarding the former [MWZ03], [CG00], and [NL00] again develop genuine logics, while Mobile Maude [DELM00] and Mobile UNITY [RMP97, MR98, RM02] mix them with programming notations.

[MWZ03] extends Lamport's Temporal Logic of Actions (TLA) [Lam94] by spatial modalities. Location is expressed in relative terms, with the entire structure of a mobile system again being given by a tree, and mobility by a change of this topology, i.e. movement of subtrees. The resulting Mobile TLA (MTLA) is a linear-time temporal logic, which also provides an "everywhere" and "move" operator, and where terms and formulae are evaluated relative to a named location, e.g. $loc[\varphi]$. Furthermore, while $loc[\varphi]$ is used to express that φ holds at loc if loc exists, $loc\langle\varphi\rangle$ additionally asserts that loc exists. MTLA formulae are interpreted over *runs,* which are sequences of pairs of trees and valuations of variables.

In contrast to this, the spatio-temporal logic developed in [CG00] uses the Ambient Calculus discussed in the previous section as its model. It also describes location in relative terms and offers a "somewhere" modality, which for MTLA can be derived from its "everywhere" operator analogously to the well-known $\exists x P(x) \equiv \neg\forall x \neg P(x)$ and vice versa. However, a distinction between definitely and possibly existing locations is not made, but additionally two operators to express security properties are provided.

[NL00] provides a logic for proving properties of KLAIM specifications (cf. previous section). Accordingly, it includes constructs to describe the existence of a tuple t in a tuple space of node s, $t@s$, as well as the insertion or retraction of tuples and, hence, the tuple space's evolution. In contrast to the two other logics, location is treated in absolute terms.

Mobile UNITY [RMP97, MR98, RM02] also expresses location directly, using a distinguished variable λ associated with every unit of specification. As another extension to standard UNITY [CM88], assignment statements are labelled, and based on these two additions constructs which reflect location dependency in communication, namely transient variable sharing, and which allow entities to share data and synchronise actions when in close proximity, transient action synchronisation, are then provided. For example, for two entities A and B, the expression "$A.x \approx B.y$ **when** $A.\lambda = B.\lambda$" indicates that changes to variable x will be propagated to y and vice versa when A and B are at the same location (shared variable); "**disengage**$(A.x, B.y)$ **when** r **value** δ_1, δ_2" describes that δ_1 will be assigned to $A.x$ and δ_2 to $B.y$ when condition r is satisfied (disengage clause); and assuming labelled statements $incx :: x := x + 1$ in A and $incy :: y := y + 1$, "$A.incx \,\Upsilon\, B.incy$ **when** r" denotes that incrementation of both x and y is selected for simultaneous execution if r is true (coselection transient action synchronisation).

While MTLA and the logic for Ambients directly support the specification of nested subsystems due to the tree structures they use to represent locations in relative terms, neither Mobile UNITY nor the logic for KLAIM facilitate this feature. The mobile agent language Mobile Maude [DELM00], an extension of the Maude high-level language and a high-performance system supporting executable specification and declarative programming in rewriting logic [CDE+01], provides a third possibility. Its main entities are stationary processes and mobile objects, represented by the two classes P and MO, respectively. Here, while a process can be nested within another, the same is not possible for a mobile object. The semantics of Mobile Maude can be given by a small number of standard Maude rewrite rules. [DELM00] states that such a specification is executable and can be used as a Mobile Maude simulator. It also details that this is already possible for standard Maude, and that a corresponding extension for Mobile Maude is under development.

[AB96] adds constructs to express mobility with Petri Nets in a way similar to the mobility concept of the π-calculus described in the previous section: equating places of Place/Transition Nets with channels of the π-calculus, tokens can be regarded as names for places, and the destinations in the postset of a transition

can then be made dependent upon the input tokens read in in the preset of that transition. This approach can be extended even further by letting transitions generate not only new markings, but also new sets of places and transitions, resulting in what Asperti and Busi refer to as "Dynamic Nets."

In order to conclude our review on approaches for specifying mobile systems, we come back to another tuple-space-based approach. Different from KLAIM (cf. previous section), however, MobiS [Mas99] is not related to process algebras. The specification language MobiS features hierarchical tuple spaces and multiset rewriting. It distinguishes three types of tuples, namely program tuples representing code, space tuples representing locations, and ordinary tuples as ordered sequences of values. Location is therefore again treated in relative terms, and similar to KLAIM, mobility is expressed via consuming and producing tuples.

Figure 3.4 provides a graphical representation of the approaches reviewed in this chapter, again as a UML class diagram.

Summing up our overview, we see that so far none of the specification techniques can be used to reflect the particular effects which mobility has on information systems, i.e. mismatching resources between mobile and stationary components, intermittent connectivity, and coping with transactions. In the following chapters, we will develop some constructs which provide a first step into this direction. The constructs will be based on the object-oriented specification language TROLL, and we will therefore start with an introduction to TROLL in the next chapter, summarising its historical development as well as its main features with regard to this thesis. We will also exemplify the latter with a sample specification that already illustrates our suggestions concerning the effects of mobility. Following this, chapter 5 covers the theoretical background underlying TROLL specifications, and in chapter 6 we will then use this in order to provide a formal underpinning of our suggestions.

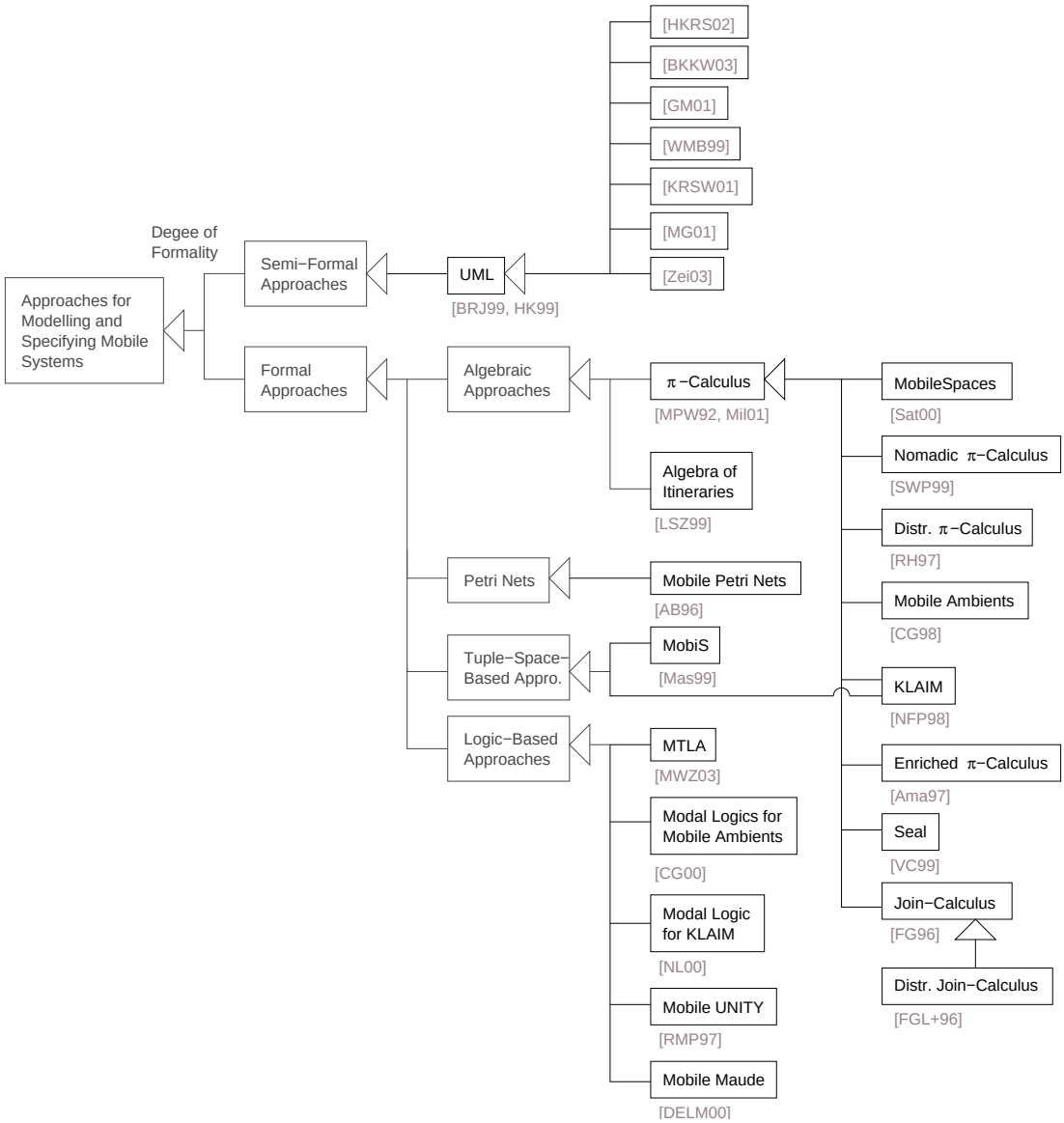


Figure 3.4: Specification approaches for mobile systems.

Chapter 4

A Brief Introduction to TROLL and its Extensions for Mobility

In this chapter, we will provide a short overview of the Textual Representation of an Object Logic Language TROLL, an object-oriented specification language, as a precursor to the detailed discussion in the following two chapters. We set out with a digest of its historical development, which will show that the specification approach actually comprises several branches, turning TROLL into a family of languages. The most recent achievements will be covered in some more detail in section 4.2, as we will build upon them in order to introduce our constructs concerning mobility-related issues in section 4.3. Section 4.4 closes the chapter by providing an example specification which illustrates the usage of TROLL in general and also of our suggestions regarding the effects of mobility in particular. This example will be frequently referred to in the following chapters 5 and 6, where we discuss the formal syntax and semantics of the language and our extensions outlined here in depth.

4.1 History

Starting with the publication [SSE87], which presented several concepts for specifying objects and object systems, on the one hand several other works like [ESS88, ESS89, ESS90, SE90, EDS93, ES95, EH96, ECSD98, Ehr99, EC00] focussed on theoretical issues of object systems, and on the other hand different dialects of object-oriented specification languages were developed.

Regarding the latter, first of all the OBLOG (Object logic) language [SSE87] has to be mentioned, which later evolved to OBL-89 described in [CSS89]. Oblog$^+$ [SJ91, JSS91]—subsequently renamed as TROLL [JSHS91] in order to avoid confusions—and a graphical variant [SSG$^+$90] of OBL-89, again referred to as OBLOG, can be considered as immediate successors of OBL-89. The first version of TROLL, described among others in [JSHS91] and [Saa93], advanced first to TROLL$_2$ [HSJ$^+$94] and later on to TROLL$_3$ [EH96, Har97, DH97, GKK$^+$98]. TROLL *light* [CGH92, GCH93] is another dialect. In the following years, OBLOG

on the one hand evolved into a commercial product [ESDI93] and on the other hand into an academic variant subsequently referred to as GNOME [RS95].

A more thorough discussion of the differences between the various dialects and versions of the OBLOG family can be found in [Saa93, Har96, Har97]. The differences between the members of the TROLL subfamily mainly root in the amount of provided language constructs, which is much larger for the earlier versions, and in that TROLL_3 is the first one which has been formalised completely. Furthermore, TROLL_3 is the first of the dialects considering aspects of concurrency and distribution.

Among others, experiences gained from the development of an information system of considerable size to provide Computer-Aided support in the process of Testing and Certifying electronic equipment for use in explosive environments CATC [KKH+96, HDK+97, SK97] at the German Federal Institute of Weights and Measures lead to the demand to equip TROLL with additional concepts for structuring large specifications and to facilitate reuse of parts of a specification. This task was approached in [Eck01], which in turn used module-theoretic results achieved in [KF00] in order to provide the formal background for the suggested language constructs on structuring and reuse. Since we will use hierarchical structures in order to express locations in a relative way, which we can then use to specify mobility-related issues, in the following section we will summarise the relevant features of TROLL_3 and the last variant of TROLL with Modules, TROLL_M, according to [Eck01]. But before doing so, in the light of structuring and reuse also the approach for the refinement of specifications developed in [Den96] should be mentioned, which also suggests some language constructs to describe transactions.

4.2 TROLL_3 and TROLL_M

This section provides an extended version of the material presented in [EAN01].

TROLL_3 has been developed for specifying information systems at a high level of abstraction, regarding them as being communities of concurrent, interacting (complex) objects. Objects in turn are units of structure and behaviour, having their own state spaces and life cycles, which are sequences of events. A TROLL specification describes an object system, is named accordingly, and consists of a set of data type and a set of object class specifications as well as of a number of object declarations and of global interaction relationships.

The language provides a number of predefined elementary data types. Additionally, it offers the possibility to construct new, problem-specific, complex data types by using type constructors. The predefined types are usually composed of the data sorts and operators assigned to them, and the type constructors consist of sort constructors and generic operations.

Object classes describe the potential objects of the system by means of structure, interface, and behaviour specifications. Attributes are used to model the state spaces of the objects. Together with the actions they form the objects' local

signatures. Attributes have data types and may be declared as hidden—i.e. only locally visible—optional or constant. They can be derived, meaning their values are calculated from the values of other attributes, or initialised.

The second part of the signature determines the actions of the objects. Each action has a unique name and an optional parameter list. Input and output values are distinguished by marking the latter with a preceding "!", and each parameter has a certain data type. The visibility of actions can be restricted in such a way that they do not belong to the interface of the object class, but can be used internally only. Actions that create objects of a class, so-called birth actions or constructors, and actions that destroy objects, so-called death actions or destructors, are explicitly marked by a "$*$" or "$+$" respectively.

While the signature part of an object class determines the interface of the objects, the behaviour part constitutes the reactions of the objects to calls of the actions declared in the interface. The admissible behaviour of the objects can be restricted by means of initialisation constraints and invariants.

Complex objects may be built using aggregation, with the consequences that the component objects can exists only in the context of the complex one and that the superior object can restrict the behaviour of its components. By means of specialisation hierarchies, base classes may be extended with further aspects.

The set of potential instances is determined by object declarations at the object system level. At runtime, concrete instances can be created through the calling of birth actions of the respective object classes. All instances of a system are concurrent to each other and synchronise when they interact. Interactions are global behaviour rules that together with the local ones describe the behaviour of the system.

All these aspects are presented in depth in [Har97] and [DH97], where the latter is also currently available as an online tutorial at `http://www.cs.tu-bs.de/idb/publications/tr97/tr97.html`.

With regard to hierarchical structures, TROLL$_M$ extends TROLL$_3$ in the way that the description of an object system may now contain *either* the specifications of object classes and objects, of data types, and of interaction relationships as already explained, *or* the specifications of subsystems, which in turn may consist of either the former items or other subsystems. Furthermore, a subsystem may use so-called offer- and request-objects, in order to declare the services it makes available to or expects to be provided by other subsystems, respectively. Objects of these two special types are therefore used to describe the interface of a subsystem, rendering all other items contained invisible and inaccessible to the outside. Object relationships like aggregation or specialisation as mentioned above are not allowed accross subsystem boundaries, as the idea for the module concept is to keep the subsystems as independent of each other as possible.

In addition to subsystems, which are used to structure large specifications, TROLL$_M$ also introduces units of reuse. However, as we only require the hierarchical structuring in order to introduce our mobility-related language constructs, we refrain from discussing these latter concepts here and instead now turn to our "mobile TROLL."

4.3 Mobile TROLL

From section 2.3 we know that one important difference between mobile and stationary units is that the former are (relatively) resource-poor with regard to the latter. This distinction can be captured in TROLL specifications if we replace the grammar rule on subsystems (<*subsystemSpec*>) in [Eck01] by the following three productions

Syntax

<*statSpec*> ::= **stationary** <*ident*>
(<*statSpec*> | <*mobiSpec*> | <*subSpec*>)
{ ';' <*statSpec*> | <*mobiSpec*> | <*subSpec*> |
<*statInstanceDecl*> | <*mobiInstanceDecl*> }
[**onEntry** { <*linkDef*> ';' } [<*actionTerm*>] **end**]
[**onExit** { <*unlinkDef*> ';' } [<*actionTerm*>] **end**]
end_stationary

<*mobiSpec*> ::= **mobile** <*ident*>
(<*mobiSpec*> | <*subSpec*>)
{ ';' <*mobiSpec*> | <*subSpec*> |<*mobiInstanceDecl*> }
[**onEntering** { <*linkDef*> ';' } [<*actionTerm*>] **end**]
[**onExiting** { <*unlinkDef*> ';' } [<*actionTerm*>] **end**]
end_mobile

<*subSpec*> ::= [<*offerSpec*> { ';' <*offerSpec*> }]
[<*requestSpec*> { ';' <*requestSpec*> }]
[<*specItem*> { ';' <*specItem*> } ';']

---

and modify the "root" production, i.e. the one concerning the entire object system, to reflect this change:

Syntax

<*systemSpec*> ::= **object system** <*ident*>
(<*partSpec*> { ';' <*partSpec*> } ';' |
<*specItem*> { ';' <*specItem*> } ';')
end.

<*partSpec*> ::= <*statSpec*> | <*statInstanceDecl*> |

<*mobiSpec*> | <*mobiInstanceDecl*> |
<*dataTypeSpec*> | <*behaviorSpec*>

Just like for object classes and objects, we distinguish between the description of a subsystem's general features using the <*statSpec*> and <*mobiSpec*> productions, and the declaration of an actual instance of it:

Syntax

<*statInstanceDecl*> ::= **stationaries** <*ident*> **:** <*ident*>
<*mobiInstanceDecl*> ::= **mobiles** <*ident*> **:** <*ident*>

Here, the "type" specified after the colon should, of course, match an <*ident*> following the **mobile** or **stationary** keyword from a general specification.

In addition to distinguishing between the mobile and stationary parts of a complex system, these rules also allow to group related functionality together, which is not possible with the formal approaches discussed in chapter 3. It is thus possible to specify that a mobile agent will migrate together with all the objects it contains, that a mobile hardware device moves together with all the subsystems it comprises, or that mobile software moves within mobile hardware by wrapping it into a **mobile** subsystem. This nesting also applies to stationary entities, but while a stationary unit may contain other mobile *and* stationary subsystems in addition to elementary classes and objects, mobile entities may only contain the latter and other mobile units, but *no* stationaries. This should seem reasonable, since any item with a built-in fixed unit can itself not be mobile.

Connections between mobile and stationary entities, however, are possible and can be described in the form of communication links. The <*linkDef*>/<*unlinkDef*> productions can be used to identify actions of requestor offer-objects: a <*linkDef*> clause specifies an action for which a link can be established if a matching action with an equal number of corresponding types of parameters is offered in the same way by the entering unit or the unit being entered; similarly, an <*unlinkDef*> clause is used to determine which established connection will be released if a unit departs from another:

Syntax

<*linkDef*> ::= **link** <*ident*> '.' <*actionSignature*>
<*unlinkDef*> ::= **unlink** <*ident*> '.' <*actionSignature*>

The optional <*actionTerm*> in the **onEntry/onExit/onEntering/onExiting** sections can be used to name an action which should be triggered when

the corresponding event occurs. The prerequisite for this in turn is that a migration takes place, which can be specified using a special type of action rule:

Syntax

<*movement*> ::= **if** <*pFormula*> **moveTo** <*target*>
<*target*> ::= <*ident*> | { '/' <*ident*> } '/' <*ident*>

This rule can be used for an object contained within a mobile entity to specify another (mobile or stationary) subsystem as the *target* into which the unit enclosing this object should migrate if the proposition <*pFormula*> can be evaluated to true. The *target* can be given as an absolute value or by a variable or attribute, but is required to denote a place in the location hierarchy implied by the nesting of mobile and stationary subsystems.

In a nutshell, these are already all the language constructs suggested as an extension to TROLL$_M$. As the syntax of the language is fairly intuitive, we postpone details on further grammar and frame rules to subsequent chapters and instead now turn to an example of how these constructs can be used.

4.4 Sample Application

In this section, the language constructs of TROLL described in the previous section will be illustrated by giving a specification of a system for mobile-agent-based retrieval of information from text documents, as this is a well-known area for the beneficial application of mobile software (cf. section 2.1), which on the one hand suffices to illustrate all of our extensions, but on the other hand is still fairly intuitive in order to be understandable without deeper knowledge of technical details. A continuous representation of this example is given in appendix A.1.

At the top level, we assume the system to comprise two units: one representing its users and the other containing the sites providing documents and the mobile agent to search them:

```
object system DocumentRetrieval

  stationary Users ... end_stationary;
  stationary LiteratureSystem ... end_stationary;
  stationaries usrs: Users;
  stationaries litSys: LiteratureSystem;

  behavior
    usrs.searchFacility.doKeywordSearch(userName,
                                        keywords,
                                        homeAndFirstTarget,
```

```
                                          resultSet)
    do
      litSys.searchEng.searchKeywords(userName,
                                      keywords,
                                      homeAndFirstTarget,
                                      resultSet)

    od;
  end;
  data type litInfo = record(title: string, authors: list(string),
                             year: nat, type: string);
end.
```

The `behavior` rule states that any time the action `doKeywordSearch` of the interface object `searchFacility` within the instance `usrs` of the `Users` subsystem is called, the `searchKeywords` action for the `searchEng` interface object contained in the instance `litSys` of the subsystem `LiteratureSystem` occurs simultaneously for the same parameters. This usage already indicates that `searchFacility` will be a request-object of `usrs` and `searchEng` an offer-object of `litSys`, as we will see in the following.

We anticipate two types of users of the system: on the one hand people wanting to search for information on documents containing certain keywords, and on the other hand an administrator registering and deregistering the former, adding new sites to be analysed to the system etc. For both types, the specification provides corresponding object classes and instances:

```
stationary Users
  request_object searchFacility
    actions
      doKeywordSearch(uname: string,
                      kw: set(string),
                      targets: list(string),
                      ! answer: set(litInfo));
  end;

  object class USER
    attributes
      name: string;
      keywords: set(string);
      targetList: list(string);
     ...
    actions
      * becomeAUser(n: string);
      setKeywords(keyw: set(string));
      setTargets(targets: list(string));
      subscribe();
```

```
      searchByKeywords(! resultSet: set(litInfo));
      searchByKeywords(k: set(string),
                       t: list(string),
                       ! r: set(litInfo)) hidden;
     ...
    behavior
      becomeAUser(n) do name:= n od;
      setKeywords(keyw) do keywords:= keyw od;
      setTargets(targets) do targetList:= targets od;
      searchByKeywords(r) do
        searchByKeywords(keywords, targetList, r)
      od;
     ...
  end;

  object class ADMINISTRATOR
     ...
    actions
      * becomeAdmin();
      registerUser(userName: string);
     ...
  end;

  objects admin: ADMINISTRATOR;
  objects user(name: string): USER;

  behavior
    user(name).subscribe() do
      admin.registerUser(name);
    od;
    // communication with interface objects:
    user(name).searchByKeywords(keywords, targetList, resultSet)
    do
      searchFacility.doKeywordSearch(name,
                                     keywords,
                                     targetList,
                                     resultSet);
    od;
     ...
  end;
end_stationary;
```

An object class is used to describe the properties which any instances of the class will have. The interface which an object class provides consists of attributes and actions. The range of values for the former will be given by a predefined or

complex data type. Actions may have parameters used to describe the exchange of data between the caller of an action and the action itself. The output parameters, to be determined by the action, are marked by a preceding exclamation mark. In the passage given above, a call to the `searchByKeywords(! resultSet: set(litInfo))` action should deliver a relevant set of literary information. As detailed in the `behavior` section of the `USER` class, a call to this action—which may correspond to a human user pressing a button—is synchronised with the `hidden` action of the same name, which accesses the object's attributes in order to provide the values for the input parameters. A (human) user could therefore use the same set of keywords for searches at different target sites, or look at the same sites using another set of keywords. For example, if the action `setKeywords` is called with the parameter `keyw`, then the `behavior` section also states that the attribute `keywords` should be set to this value.

The `behavior` section of the stationary subsystem `Users` states that a call to the `subscribe` action for a `USER` object identified by a certain `name` will occur simultaneously with the `registerUser` action of the `ADMINISTRATOR` instance `admin` for the same `name`. More interestingly, however, it also synchronises the hidden action `searchByKeywords` with the action of the request-object `searchFacility`. As already mentioned, the latter is in turn linked to the `searchKeywords` action offered by the `searchEng` interface object of the `litSys` instance of the subsystem `LiteratureSystem`:

```
stationary LiteratureSystem
  offer_object searchEng
    actions
      searchKeywords(userName: string,
                     keywords: set(string),
                     homeAndFirstTarget: list(string),
                     ! answer: set(litInfo))
  end;
 ...
  stationaries managingUnit: ManagingUnit;
  mobiles retrievalUnit: RetrievalUnit;
  stationaries litHost1: LiteratureHost1;
 ...
```

The declarations of stationary and mobile subsystems already indicate that the `LiteratureSystem` comprises a mobile `RetrievalUnit` representing the mobile agent, which will migrate from `LiteratureHost` to `LiteratureHost` (for reasons of brevity, here and in the following we consider only one declaration/description) searching for relevant information, guided by a `ManagingUnit`. A rationale for specifying the latter is that with such a central point `LiteratureHosts` could be added to or removed from the system, user-access could be supervised etc. This entity is also the one which provides the object to deal with the call to the action of the interface object `searchEng`, and upon authorisation sends the `RetrievalUnit` to its first target. As the corresponding

specification involves another chain of `behavior` rules which do not differ much from those already discussed, we omit these details here and instead turn to the parts of the specification illustrating the other language constructs suggested for a "mobile TROLL."

The mobile `RetrievalUnit` contains a number of interface objects and an instance of the `RETRIEVER` class:

```
mobile RetrievalUnit
  offer_object searchFacility
    actions
      searchKeywords(keywords: set(string),
                     homeAndFirstTarget: list(string),
                     ! resultSet: set(litInfo));
  end;
  request_object tripPlanner
    actions
      getNextTarget(someLoc: string, ! nextLoc: string);
  end;
  request_object searchEngine
    actions
      searchKeywords(keywords: set(string), ! resultSet: set(litInfo));
  end;

  object class RETRIEVER ... end;

  objects retriever: RETRIEVER;

  behavior
    retriever.searchForKeywords(keywords, resultSet) do
      searchEngine.searchKeywords(keywords, resultSet);
    od;
    retriever.determineNextTarget(someLoc, nextLoc) do
      tripPlanner.getNextTarget(someLoc, nextLoc)
    od;
    searchFacility.searchKeywords(keywords,
                                  homeAndFirstTarget,
                                  resultSet)
    do
      retriever.launchRetrieval(keywords,
                                homeAndFirstTarget,
                                resultSet);
    od;
  end;
  onEntering
    link searchEngine.searchKeywords(set(string), ! set(litInfo));
```

```
    retriever.continueRetrieval();
  end
  onExiting
    unlink searchEngine.searchKeywords(set(string), ! set(litInfo));
  end
end_mobile;
```

The `searchFacility` object is offered to the `ManagingUnit` in order to allow it to initiate a search, and the `tripPlanner` object is used to request the next destination from it while a search visiting several sites is in progress. Both objects are used in persistent communication links. The `searchEngine` object, however, represents a connection that should be established upon entering another `LiteratureHost`, and released upon leaving it. The offer-object `searchF` of a `LiteratureHost` contains a matching action with an equal number of corresponding types of parameters:

```
stationary LiteratureHost1
  offer_object searchF
    action
      searchKeyw(keyw: set(string), ! resultS: set(litInfo));
  end;
  stationary Reception
    offer_object searchFacility
      actions
        searchByKeywords(keywords: set(string),
                         ! resultSet: set(litInfo));
    end;
     ...
  end_stationary;
  stationaries reception: Reception;
   ...
  onEntry
    link searchF.searchKeyw(set(string), ! set(litInfo));
  end
  onExit
    unlink searchF.searchKeyw(set(string), ! set(litInfo));
  end
end_stationary;
```

The `onEntering` clause of the `RetrievalUnit` also details that when reaching a new destination the `retriever` object should proceed with its `continueRetrieval()` method, which should add new information on relevant literature to the one obtained so far, determine a suitable destination for another migration (via the `ManagingUnit`) and move to it unless it has already completed its itinerary, as outlined in the `behavior` section of the `RETRIEVER` class:

```
object class RETRIEVER
  attributes
    homeLoc: string;
    currentLoc: string initialized ‘‘none’’;
    nextTarget: string;
    keywords: set(string);
    litrInfo: set(litInfo);
  actions
    * becomeRetriever();
    launchRetrieval(kw: set(string),
                    tL: list(string),
                    ! resultSet: set(litInfo));
    searchForKeywords(keywords: set(string),
                      ! resultSet: set(litInfo));
    determineNextTarget(someLoc: string,
                        ! nextLoc: string);
    continueRetrieval();
    + cease();
  behavior
    launchRetrieval(kw, tL, resultSet)
      onlyIf cnt(tL) > 1
    do
      keywords:=kw,
      homeLoc:=tL(1),
      nextTarget:=tL(2), ...,
      if (true) moveTo tL(1),
    od;
     ...
    continueRetrieval()
      var newLiterature: set(litInfo),
          nxtLocBuf: string,
          nxxtTargetBuf: string
    do
      litrInfo:= litrInfo + newLiterature,
      searchForKeywords(keywords, newLiterature),
      currentLoc:=nxtLocBuf,
      nextTarget:=nxxtTargetBuf,
      determineNextTarget(currentLoc, nxtLocBuf),
      determineNextTarget(nextTarget, nxxtTargetBuf),
      if (currentLoc # homeLoc) moveTo nextTarget
    od;
end;
```

The values to be assigned to the object's attributes should be provided by the `ManagingUnit`, and should be consistent with the location hierarchy. The

usage of additional buffering attributes (`nxtLocBuf, nxxtTargetBuf`) is due to the way in which read and write access to attributes and variables are performed in TROLL. Details on this will be provided in the next two chapters, where we turn to the theoretical foundations of the language. Chapter 5 is concerned with the theoretical framework of TROLL in general, while chapter 6 is dedicated to the formal semantics of the new mobility-related language constructs in particular. Readers mainly interested in using these constructs may skip to chapter 7, which illustrates their use in the development of an application to access a web service from mobile phones.

Chapter 5

Theoretical Foundations

Information systems can be regarded as communities of (complex) concurrent interacting objects, where objects are considered to be units of structure and behaviour, having their own state spaces and life cycles consisting of sequences of events. An object system contains a number of concurrent objects, every object with a set of possible sequential life cycles. Interactions between these objects can be specified via global actions, which are the actions being executed simultaneously by all the involved objects, thus representing a synchronisation between these objects. Such an object system can be described by a TROLL specification, whose attributes, actions, components, and specialisation hierarchies then determine the system's signature, while the behaviour and interaction rules, preconditions, etc. provide axioms for the behaviour of the system.

For object systems without subsystems, at the formal level the signature-related part is given in terms of an extended data signature consisting of a (simple) data signature and a class signature. It determines the alphabet of the system specification, from which terms and formulae can be constructed. The axioms of the system, which describe its dynamic properties, are given in terms of a distributed temporal logic. For object systems *with* subsystems, [Eck01] developed a TROLL extension which allows to capture this distinction in specifications, using a module concept as a means to structure large specifications and reuse parts of a model. As a hierarchical structure can also be used to express locations—as a prerequisite for describing mobility—in relative terms (cf. section 3, in particular table 3.1), we build upon this work and the corresponding mathematical framework developed in [KF00] in order to introduce our language constructs in chapter 6. This chapter provides the corresponding theoretical foundations, mainly as a translation and adaptation from [Eck01] and [KF00].

In the following section, we start by considering systems without subsystems, as this provides the required basis for the more advanced concepts in section 5.2, which then allow us to formalise hierarchical location structures.

5.1 Object Specifications

This section provides a formal semantics for the signature- and behaviour-related parts of object specifications. Considering the former, we apply concepts used in relation to algebraic specification of abstract data types, like data signatures, class signatures, extended data signatures, instance signatures, algebras, and the construction of respective terms. Furthermore, in conjunction with instance signatures event structures will be introduced, and the latter are then reused to define the semantics of the Distributed Temporal Logic (DTL) discussed with regard to the behaviour-related part of object specifications.

5.1.1 Signatures

Extended data signatures are constructed from (simple) data signatures and class signatures, and they can be regarded as generic descriptions of possible instances of objects from an object system. They also allow to formulate identity and action terms, which can then be interpreted by a suitable algebra. However, in order to establish a mathematical model for a system of concurrent objects, a generic description of object instances is not sufficient. Rather, a particular specimen is needed, and a so-called "instance signature" is used to describe it, consisting of a set of object identities and a set of global actions. The global actions are constructed over the action alphabets of the individual objects, describing the communication between concurrent objects and, hence, their synchronisation.

An instance signature can be interpreted by a labelled prime event structure, where every single object is assigned to a sequential event structure. The latter can be viewed as a tree, so that a possible life cycle of an object in a sequential event structure corresponds to a branch of the tree. Sequential event structures are concurrent to each other and get synchronised by shared communication events.

Labelling functions are used to relate the events of an event structure to the actions of the underlying instance signature.

Following this outline, we start by defining data signatures, class signatures, and extended data signatures together with the relevant algebras and rules on suitable term constructions, before closing this section with instance signatures, event structures, and labelling functions.

Definition 5.1 (data signature) A data signature is a pair $\Sigma_D = (S_D, \Omega_D)$, where S_D is a set of data sorts and $\Omega_D = \{\Omega_{x,s}\}_{x \in S_D^*, s \in S_D}$ is an $S_D^* \times S_D$-indexed family of sets of operators. □

Informally, a (data) sort can be considered as being a name of a type, and an operator as denoting a function [EGL89, LEW96, ST99].

We use $\omega : s_1 \times \ldots \times s_n \to s \in \Omega$ as an alternative notation for $\omega \in \Omega_{s_1 \ldots s_n, s}$, and also $\omega : x \to s \in \Omega_D$ provided $x = s_1 \ldots s_n$. If $n = 0$, then the operator $\omega : \to s$ is called a *constant of sort s.* We use the terms *operator* and *operation symbol* synonymously.

Many-sorted algebras as introduced in the following definition determine interpretation structures for signatures by providing for every sort a carrier set of concrete data values and for every operator a specific operation/function defined on the data values of the corresponding carrier set.

Definition 5.2 (Σ_D-algebra) Given a data signature $\Sigma_D = (S_D, \Omega_D)$, a Σ_D-algebra $A(\Sigma_D) = (A(S_D), A(\Omega_D))$ is a pair consisting of

1. a set $A(S_D) = \{A_s\}_{s \in S_D}$ of sets (carrier sets), one carrier set for every data sort $s \in S_D$, and

2. an $S_D^* \times S_D$-indexed family $A(\Omega_D) = \{A_\omega \,|\, \omega \in \Omega_{x,s}\}_{x \in S_D^*, s \in S_D}$ of mappings, one mapping $A_\omega : A_{s_1} \times \ldots \times A_{s_n} \rightarrow s$ for every operation symbol $\omega \in \Omega_D, \omega : s_1 \times \ldots \times s_n \rightarrow s$, provided $x = s_1 \ldots s_n$.

□

A data signature can be used as the basis for defining data terms. The following definition shows how to do this and additionally introduces the family of sets of free variables, $free(t)$, of a data term t.

Definition 5.3 (data terms) Let a data signature $\Sigma_D = (S_D, \Omega_D)$ be given and let $X = \{X_s\}_{s \in S_D}$ be the family of sets of sort-indexed variables over Σ_D. The family of sets of Σ_D data terms over the variables X, denoted by $T_{\Sigma_D}(X)$, and the family of sets of the free variables $free(t) = \{free_s(t)\}_{s \in S_D}$ of a data term t are defined as follows:

1. Any variable $x \in X_s, s \in S_D$, is a data term of sort s, and $free_s(x) = \{x\}, free_{s'}(x) = \{\}$ for $s \neq s'$.

2. If $t_1, \ldots, t_n$ are data terms of sorts $s_1, \ldots, s_n$, respectively, and if $\omega : s_1 \times \ldots \times s_n \rightarrow s$ is an operation symbol from $\Omega_{s_1 \ldots s_n}$ then $\omega(t_1, \ldots, t_n)$ is a data term of sort s and $free(\omega(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} free(t_i)$. In case of constants, we have $free(\omega()) = \{\}$.

3. No other string is a data term in $T_{\Sigma_D}(X)$.

□

If a term contains variables, a certain value has to be assigned to every variable before the term can be evaluated:

Definition 5.4 (assignment) Let a data signature Σ_D, a Σ_D-algebra $A(\Sigma_D) = (A(S_D), A(\Omega_D))$, and a family $X = \{X_s\}_{s \in S_D}$ of sets of sort-indexed variables be given, then a variable assignment ρ is defined by a family of sort-indexed mappings

$$\rho = \{\rho_s : X_s \rightarrow A_s\}_{s \in S_D},$$

which associates every variable $x \in X$ with a value $\rho(s)$ from the carrier set A_s.
□

Definition 5.5 (data term interpretation) Given an interpretation structure $A(\Sigma_D) = (A(S_D), A(\Omega_D))$ and a variable assignment ρ, the interpretation $[\![t]\!]^{\rho}_{A(\Sigma_D)}$ of a data term $t \in T_{\Sigma_D}(X)$ is defined inductively by:

1. $[\![x]\!]^{\rho}_{A(\Sigma_D)} = \rho_s(x)$ for $x \in X_s$,
2. $[\![\omega(t_1, \ldots, t_n)]\!]^{\rho}_{A(\Sigma_D)} = A_\omega([\![t_1]\!]^{\rho}_{A(\Sigma_D)}, \ldots, [\![t_n]\!]^{\rho}_{A(\Sigma_D)})$ for $\omega \in \Omega_{s_1 \ldots s_n, s}$ and $t_i \in T_{\Sigma_D}(X), i = 1, \ldots, n$.

□

Using the definitions given above, we are now able to determine class signatures and extended data signatures, which in turn allow us to represent the object-oriented concepts "identity" and "action" with the help of algebraic techniques. Here, a class signature permits to express the signature-related part of the specification of an object system in an abstract syntax, the signature consisting of a set of object sorts plus, for every object sort, a set of instance symbols as well as of a set of action symbols.

In contrast to other previous works (e.g. [Den96, Har97, Eck01]), like [KF00] we also explicitly specify attribute symbols, as this will ease establishing the formal semantics of our language constructs for modelling mobile and stationary units, cf. definition 5.52. However, in accordance with the earlier theses we continue to introduce read and write operators for attributes as additional action symbols, since it is in general the action symbols which are used as the range for a labelling function on event structures, and [KF00] accesses attributes merely in a read-only way.

Definition 5.6 (class signature) Given a data signature $\Sigma_D = (S_D, \Omega_D)$, a class signature $\Sigma_C = (S_O, I, AT, AC)$ over Σ_D consists of

1. a set S_O of object sorts,
2. a family $I = \{I_{x,b}\}_{x \in S^*_{DO}, b \in S_O}$ of sets of instance symbols,
3. a family $AT = \{AT_{b,s}\}_{b \in S_O, s \in S_{DO}}$ of sets of attribute symbols, and
4. a family $AC = \{AC_{x,b}\}_{x \in S^*_{DO}, b \in S_O}$ of sets of action symbols,

where $S_{DO} = S_D \cup S_O$. □

In analogy to the alternative notation for operators of data signatures, we may write $i : x_1 \times \ldots \times x_n \to b$ for $i \in I_{x_1 \ldots x_n, b}$, $at : b \to s$ for $at \in AT_{b,s}$, and $ac : x_1 \times \ldots \times x_n \to b$ for $ac \in AC_{x_1 \ldots x_n, b}$.

In order to illustrate this and the following definitions, we will examine the sample application introduced in section 4.4, where for now we restrict ourselves to the passages relating to the `RETRIEVER` object class and the corresponding object declaration.

Example 5.7 (Mobile-Agent-Based IR—class signature) We assume a data signature being given which contains some basic data sorts like string, nat, int, etc. as well as a data sort `litInfo` (cf. section 4.4), whose structure here will not be analysed any further.

The set S_O contains one element for the object class `RETRIEVER`. To ease reading, we choose the same name for the matching sort:

$$S_O = \{\texttt{RETRIEVER}\}.$$

The corresponding set of instance symbols comprises only the instance symbol `retriever`, $I_{\epsilon,\texttt{RETRIEVER}} = \{\texttt{retriever}\}$. Applying the notation used in [Eck01] we obtain

$$I = \{\texttt{retriever} : \rightarrow \texttt{RETRIEVER}\}$$

and, likewise,

$$\begin{aligned} AT = \{&\texttt{homeLoc} : \texttt{RETRIEVER} \rightarrow \mathsf{string},\\ &\texttt{currentLoc} : \texttt{RETRIEVER} \rightarrow \mathsf{string},\\ &\texttt{nextTarget} : \texttt{RETRIEVER} \rightarrow \mathsf{string},\\ &\texttt{keywords} : \texttt{RETRIEVER} \rightarrow \mathsf{set}(\mathsf{string})\\ &\texttt{litrInfo} : \texttt{RETRIEVER} \rightarrow \mathsf{set}(\texttt{litInfo}) \end{aligned}$$

for the family of sets of attribute symbols and

$$\begin{aligned} AC = \{&\texttt{becomeRetriever} : \rightarrow \texttt{RETRIEVER}\\ &\texttt{launchRetrieval} :\\ &\mathsf{set}(\mathsf{string}) \times \mathsf{list}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{RETRIEVER},\\ &\texttt{searchForKeywords} : \mathsf{set}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{RETRIEVER},\\ &\texttt{determineNextTarget} : \mathsf{string} \times \mathsf{string} \rightarrow \texttt{RETRIEVER},\\ &\texttt{continueRetrieval} : \rightarrow \texttt{RETRIEVER}\\ &\texttt{cease} : \rightarrow \texttt{RETRIEVER}\}. \end{aligned}$$

for the family of sets of action symbols. □

In order to construct an extended data signature for a given class signature, first two data sorts have to be introduced for every object sort—one for the object identities and the other for their actions. Then, for each of these new sorts the set Ω has to be extended by an operation symbol for every instance symbol and every action symbol of the class signature:

Definition 5.8 (extended data signature) Given a data signature $\Sigma_D = (S_D, \Omega_D)$ and a class signature $\Sigma_C = (S_O, I, AT, AC)$ over Σ_D, an extended data signature $\Sigma = (S, \Omega)$ consists of

1. the sorts $S = S_D \cup S_O^i \cup S_O^{at} \cup S^{ac}$ with disjoint sets $S_O^i = \{b^i \mid b \in S_O\}$, $S_O^{at} = \{b^{at} \mid b \in S_O\}$, and $S_O^{ac} = \{b^{ac} \mid b \in S_O\}$, and

2. the operators $\Omega = \Omega_D \cup \Omega_O^i \cup \Omega_O^{at} \cup \Omega_O^{ac}$, where $\Omega^i_{O;x^i,b^i} = I_{x,b}$, $\Omega^{at}_{O;b^i b^{at},s} = AT_{b,s}$, and $\Omega^{ac}_{O;b^i x^i,b^{ac}} = AC_{x,b}$ holds for any $b \in S_O, s \in S_{DO}, x \in S^*_{DO}$. All other sets of the families of sets are empty. For $s_1 \ldots s_n = x \in S^*_{DO}$, the expression x^i is defined as follows: let $S^i = S_D \cup S^i_O$, then $s^i_1 \ldots s^i_n = x^i \in S^{i*}$, where, for $j = 1, \ldots, n$, $s^i_j = s_j$ for $s_j \in S_D$ and $s^i_j = b^i_j$ for $s_j \in S_O$.

For $b^i \in S^i_O$ and $\alpha \in \Omega^{ac}_{O;b^i x^i,b^{ac}}$, α is an *elementary action symbol* for b^i and $\Omega^{ac}_{O;b^i x^i,b^{ac}}$ denotes the set of elementary action symbols for b^i.

We may also write $obj : s_1 \times \ldots \times s_n \to b^i$ for $obj \in \Omega^i_{O;x^i,b^i}, x^i = s_1 \ldots s_n$, $att : b^i \times b^{at} \to s$ for $att \in \Omega^{at}_{O;b^i b^{at},s}$ and $act : b^i \times s_1 \times \ldots \times s_n \to b^{ac}$ for $act \in \Omega^{ac}_{O,b^i x^i,b^{ac}}, x^i = s_1 \ldots s_n$. □

The following example illustrates how to construct an extended data signature from a class signature:

Example 5.9 (Mobile-Agent-Based IR—extended data signature) The class signature given in example 5.7 determines the following sorts and operators for the corresponding extended data signature:

$$
\begin{aligned}
S^i_O &= \{\texttt{RETRIEVER}^i\} \\
S^{at}_O &= \{\texttt{RETRIEVER}^{at}\} \\
S^{ac}_O &= \{\texttt{RETRIEVER}^{ac}\} \\
\Omega^i_O &= \Omega^i_{\epsilon,\texttt{RETRIEVER}^i} \\
&= \{\texttt{retriever} : \to \texttt{RETRIEVER}^i\} \\
\Omega^{at}_O &= \{\texttt{homeLoc} : \texttt{RETRIEVER}^i \times \texttt{RETRIEVER}^{at} \to \mathsf{string}, \\
&\quad \texttt{currentLoc} : \texttt{RETRIEVER}^i \times \texttt{RETRIEVER}^{at} \to \mathsf{string}, \\
&\quad \texttt{nextTarget} : \texttt{RETRIEVER}^i \times \texttt{RETRIEVER}^{at} \to \mathsf{string}, \\
&\quad \texttt{keywords} : \texttt{RETRIEVER}^i \times \texttt{RETRIEVER}^{at} \to \mathsf{set}(\mathsf{string}), \\
&\quad \texttt{litrInfo} : \texttt{RETRIEVER}^i \times \texttt{RETRIEVER}^{at} \to \mathsf{set}\texttt{litInfo}\} \\
\Omega^{ac}_O &= \{\texttt{becomeRetriever} : \texttt{RETRIEVER}^i \to \texttt{RETRIEVER}^{ac}, \\
&\quad \texttt{launchRetrieval} : \\
&\quad \texttt{RETRIEVER}^i \times \mathsf{set}(\mathsf{string}) \times \mathsf{list}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \to \texttt{RETRIEVER}^{ac}, \\
&\quad \texttt{searchForKeywords} : \\
&\quad \texttt{RETRIEVER}^i \times \mathsf{set}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \to \texttt{RETRIEVER}^{ac}, \\
&\quad \texttt{determineNextTarget} : \texttt{RETRIEVER}^i \times \mathsf{string} \times \mathsf{string} \to \texttt{RETRIEVER}^{ac}, \\
&\quad \texttt{continueRetrieval} : \texttt{RETRIEVER}^i \to \texttt{RETRIEVER}^{ac} \\
&\quad \texttt{cease} : \texttt{RETRIEVER}^i \to \texttt{RETRIEVER}^{ac}\}
\end{aligned}
$$

□

An extended data signature can be interpreted by an algebra just like a (simple) data signature by a Σ_D-algebra. Again, every sort has to be associated with a carrier set, and every operation symbol with an operation/function on the carrier

sets of the sorts for its parameters. In order to guarantee unique object identities, here the identity sorts have to be interpreted by disjunct carrier sets. The operation symbols of an extended data signature derived from action symbols of a class signature contain a reference to the corresponding instance as their first parameter, which is needed when constructing action terms. However, in an interpretation this reference is not required. The operations of the algebra therefore contain one parameter less than the corresponding operation symbols of the extended data signature. Similarly, in the interpretation of an operation symbol derived from an attribute symbol of a class signature, both the reference and the target sort are neglected, as these operation symbols "are to be understood as constants" [KF00].

Definition 5.10 (algebras for extended data signatures) Let $\Sigma = (S, \Omega)$ be an extended data signature. A Σ-algebra $A(\Sigma) = (A(S), A(\Omega))$ over Σ is a pair defined by

1. a set $A(S) = \{A_s\}_{s \in S}$ of sets (carrier sets), one carrier set for every sort $s \in S$, and
2. an $S^* \times S$-indexed family $A(\Omega) = \{A_\omega \mid \omega \in \Omega_{x,s}\}_{x \in S^*, s \in S}$ of mappings, one mapping for every operation symbol $\omega \in \Omega$, such that
   (a) if $\omega \in \Omega_D, s_1 \ldots s_n = x \in S_D^*$, and $s \in S_D$, then $A_\omega : A_{s_1} \times \ldots \times A_{s_n} \to A_s$,
   (b) if $\omega \in \Omega_O^i$ and $x = x^i, s = b^i$ as in definition 5.8, then $A_\omega : A_{s_1^i} \times \ldots \times A_{s_n^i} \to A_{b^i}$,
   (c) if $\omega \in \Omega_O^{at}$ and $x = b^i b^{at}$ and s as in definition 5.8, then $A_\omega :\to A_{b^{at}}$, and
   (d) if $\omega \in \Omega_O^{ac}$ and $x = b^i x^i, s = b^{ac}$ as in definition 5.8, then $A_\omega : A_{s_1^i} \times \ldots \times A_{s_n^i} \to A_{b^{ac}}$

   holds.

Furthermore, for any two arbitrary object sorts $b_1, b_2 \in S_O$, the interpretations $A_{b_1^i}$ and $A_{b_2^i}$ of the corresponding sorts b_1^i, b_2^i have to be disjunct, i.e. $A_{b_1^i} \cap A_{b_2^i} = \{\}$. □

Just like a simple data signature can be used to define a set of data terms (cf. definition 5.3), an extended data signature provides the means for deriving identity terms, attribute terms, and action terms.

Definition 5.11 (identity terms, attribute terms, action terms) Given an extended data signature $\Sigma = (S, \Omega)$ and a family $X = \{X_s\}_{s \in S}$ of sets of sort-indexed variables over Σ, the family of sort-indexed identity terms, the family of sort-indexed attribute terms, and the family of sort-indexed action terms denoted by $T_{\Sigma_{Id}}(X) = \{T_{\Sigma_{Id,s}}(X)\}_{s \in S_O^i}$, $T_{\Sigma_{At}}(X) = \{T_{\Sigma_{At,s}}(X)\}_{s \in S_O^{at}}$, and $T_{\Sigma_{Ac}}(X) = \{T_{\Sigma_{Ac,s}}(X)\}_{s \in S_O^{ac}}$, respectively, are defined as follows:

1. Any variable $x \in X_{b^i}, b^i \in S_O^i$, is a term of sort b^i.

2. If $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$, respectively, and if $\omega : s_1 \times \ldots \times s_n \rightarrow b^i$ is an operation symbol from $\Omega^i_{O;x^i,b^i}$, $x^i = s_1 \ldots s_n$, then $\omega(t_1, \ldots, t_n)$ is an identity term of sort b^i. For constants, the empty parentheses may be omitted.

3. If t and t_{at} are terms of sorts b_i and b_{at}, respectively, and if $a : b^i \times b^{at} \rightarrow s$ is an operation symbol from $\Omega^{at}_{O;b^i b^{at},s}$, then $t.a$ is an attribute term of sort s.

4. If $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$, respectively, t is a term of sort b^i, and $\omega : b^i \times s_1 \times \ldots \times s_n \rightarrow b^{ac}$ is an operation symbol from $\Omega^a_{O;b^i x^i,b^{ac}}$, $x^i = s_1 \ldots s_n$, then $t.\omega(t_1, \ldots, t_n)$ is an action term of sort b^{ac}. Here, empty parentheses should be given.

5. Nothing else besides the terms defined in 1. and 2. are identity terms, nothing else besides the terms defined in 3. are attribute terms, and nothing else besides the terms defined in 4. are action terms.

As an extended data signature is a genuine extension of a (simple) data signature, the clauses from definition 5.3 remain valid. □

Note that while, for example, $T_{\Sigma_{Id}}$ denotes the *family* of sets of all identity terms, $T_{\Sigma_{Id,s}}, s \in S_O^i$ is just a set—the set of all the identity terms of sort s.

In the notation introduced above, an attribute or action term is prefixed by an identity term i, indicating that $i.a$ or $i.\omega(x_1, \ldots, x_n)$ is an attribute or action term of object i, respectively. Overall, we can now determine the set of terms derivable from an extended data signature as follows:

Definition 5.12 (Σ-Terms) Let $\Sigma = (S, \Omega)$ denote an extended data signature and $X = \{X_s\}_{s \in S}$ a family of sets of sort-indexed variables over Σ, then the family $T_\Sigma(X)$ of sets of terms derivable from Σ is given by

$$T_\Sigma(X) = T_{\Sigma_D}(X) \cup T_{\Sigma_{Id}}(X) \cup T_{\Sigma_{At}}(X) \cup T_{\Sigma_{Ac}}(X),$$

i.e. the union of the data terms, identity terms, attribute terms, and action terms derivable from it. □

Example 5.13 (Mobile-Agent-Based IR—terms) With regard to the extended data signature given in example 5.9,

`retriever`

is an identity term from $T_{\Sigma_{Id,\texttt{RETRIEVER}^i}}(X)$,

`retriever.homeLoc`

is an attribute term from $T_{\Sigma_{At,\texttt{RETRIEVER}^{at}}}(X)$, and

`retriever.continueRetrieval()`

as well as

```
retriever.determineNextTarget(someTarget, nextTarget)
```

are action terms from $T_{\Sigma_{Ac,\mathtt{RETRIEVER}^{ac}}}(X)$ provided $X_{\mathsf{string}} = \{$ `someTarget, nextTarget, ...` $\}$ holds. □

Assignment and interpretation of data, identity, attribute, and action terms for extended data signatures are carried out in analogy to those for simple data signatures (cf. definition 5.4 and 5.5):

Definition 5.14 (term interpretation) Let $\Sigma = (S, \Omega)$ denote an extended data signature, $X = \{X_s\}_{s \in S}$ a family of sort-indexed variables over Σ, and $A(\Sigma) = (A(S), A(\Omega))$ a Σ-algebra over Σ. Furthermore, let $\rho : \{\rho_s : X_s \to A_s\}_{s \in S}$ be a variable assignment associating to each variable $x \in X_s$ a value ρ_s from the carrier set A_s, then the interpretation of a term $t \in T_{\Sigma_D}(X) \cup T_{\Sigma_{Id}}(X) \cup T_{\Sigma_{At}}(X) \cup T_{\Sigma_{Ac}}(X)$ in $A(\Sigma)$ for an assignment ρ over X is defined as follows:

1. $[\![x]\!]^{\rho}_{A(\Sigma)} = \rho_s(x)$ for $x \in X_s, s \in S_D \cup S^i_O$.
2. $[\![\omega(t_1, \ldots, t_n)]\!]^{\rho}_{A(\Sigma)} = A_\omega([\![t_1]\!]^{\rho}_{A(\Sigma)}, \ldots, [\![t_n]\!]^{\rho}_{A(\Sigma)})$ for $\omega \in \Omega_{s_1 \ldots s_n, s}$ and $s \in S_D, t_i \in T_{\Sigma_D}(X) \cup T_{\Sigma_{Id}}(X), i = 1, \ldots, n$.
3. $[\![\omega(t_1, \ldots, t_n)]\!]^{\rho}_{A(\Sigma)} = A_\omega([\![t_1]\!]^{\rho}_{A(\Sigma)}, \ldots, [\![t_n]\!]^{\rho}_{A(\Sigma)})$ for $\omega \in \Omega_{s_1 \ldots s_n, s}, s \in S^i_O, t_i \in T_{\Sigma_D}(X) \cup T_{\Sigma_{Id}}(X), i = 1, \ldots, n$ and A_ω as in definition 5.10 (2.(b)).
4. $[\![t.a]\!]^{\rho}_{A(\Sigma)} = [\![t]\!]^{\rho}_{A(\Sigma)}.A_\omega(a)$ for $t \in T_{\Sigma_{Id}}(X)$ and A_ω as in definition 5.10 (2.(c)).
5. $[\![t.\omega(t_1, \ldots, t_n)]\!]^{\rho}_{A(\Sigma)} = [\![t]\!]^{\rho}_{A(\Sigma)}.A_\omega([\![t_1]\!]^{\rho}_{A(\Sigma)}, \ldots, [\![t_n]\!]^{\rho}_{A(\Sigma)})$ for $t \in T_{\Sigma_{Id}}(X)$ and A_ω as in definition 5.10 (2.(d)).

□

The extended data signatures discussed so far represent generic descriptions of the instances of object classes. In contrast to this, the instance signatures introduced in the following describe concrete instances of some class, and also their behaviour. Every instance signature contains the set of all object identities and the set of global actions, which can be constructed over the objects' local actions. An instance signature can be derived from an extended data signature and may, for example, be interpreted by labelled event structures.

The definition of an instance signature requires the terms "local" and "global action" as well as "partners of an action." The specification of an object class determines the elementary actions of an individual object: definition 5.8 introduced elementary action symbols, which by an interpretation are mapped to elementary actions of an algebra. A local action of an object is a non-empty finite subset of the object's elementary actions; the occurrence of such a local action means that all involved elementary actions happen simultaneously. Global actions in turn are sets of local actions occurring at the same time, thus representing synchronisation points of the otherwise concurrent objects.

Definition 5.15 (actions) Given a set Id of object identities and a set EAc_{id} of elementary actions for an object identified by $id \in Id$, the set LAc_{id} of local actions for id is defined as $LAc_{id} \subseteq 2^{EAc_{id}} - \{\}$.

The set of all local actions is determined by $LAc = \bigcup_{id \in Id} Ac_{id}$, and the identity of a local action by $ID(\alpha^{local}) = id \in Id$, provided $\alpha^{local} \in LAc_{id}$.

The set of global actions is defined by $Ac \subseteq 2^{LAc} - \{\}$, where $\alpha_i^{local} \neq \alpha_j^{local} \Rightarrow ID(\alpha_i^{local}) \neq ID(\alpha_j^{local})$ has to hold for all $\alpha \in Ac, \alpha_i^{local} \in \alpha, \alpha_j^{local} \in \alpha$. □

The last constraint restricts every object to participate in a global action with at most one local action. The following definition describes the set of partners of a global action as the set consisting of the identities of the involved objects.

Definition 5.16 (partners of an action) The set of partners of an action $\alpha \in Ac$ is given by

$$P(\alpha) = \{ID(\alpha^{local}) | \alpha^{local} \in \alpha\}.$$

□

An instance signature can now be defined as follows:

Definition 5.17 (instance signature) An instance signature $\Sigma_I = (Id, Ac)$ consists of

1. a set Id of identities and
2. a set Ac of global actions constructed over the elementary actions $EAc_{id}, id \in Id$.

□

With this definition, we are now able to describe how to construct the instance signature implied by an extended data signature.

Definition 5.18 (induced instance signature) Given an extended data signature Σ and a Σ-algebra A, the instance signature $\Sigma_I = (Id, Ac)$ induced by Σ and A consists of

1. a set of object identities $Id = \bigcup\limits_{b^i \in S_O^i} Id_{b^i}$, where

$$Id_{b^i} = \{A_\omega(\vec{u}) \mid \omega \in \Omega^i_{O;x^i,b^i}, \vec{u} \in A_{x^i}\}, \text{ and}$$

2. a set of global actions Ac constructed over the elementary action alphabets $EAc_{id}, id \in Id$, where

$$EAc_{id} = \{id.A_\omega(\vec{u}) \mid \omega \in \Omega^{ac}_{O;b^i x^i,b^{ac}}, id \in A_{b^i}, \vec{u} \in A_{x^i}\},$$

with $\vec{u} = u_1, \ldots, u_n$ and $\vec{u} \in A_{x^i}$ denoting $u_j \in A_{s_j}, j \in \{1, \ldots, n\}$. □

The following example illustrates how to obtain the set Ac of global actions from the elementary action alphabets according to definition 5.15. In order to be able to illustrate global actions, we need several objects which participate in them. The parts of the specification we have been using so far in order to exemplify the definitions of our formal model, however, concern only one object—an instance of the RETRIEVER class (cf. example 5.7 and 5.9). We therefore now switch to the passages of the example specification which model the users of the system, i.e. the passages specifying the object classes USER and ADMINISTRATOR, the corresponding instance declarations, and their communication relations.

Example 5.19 (Mobile-Agent-Based IR—instance signature) In analogy to example 5.7 and 5.9, from the specification of the classes USER and ADMINISTRATOR and the corresponding instance declarations we obtain the following sorts and operators for an extended data signature

$$
\begin{aligned}
S^i_O &= \{\mathtt{ADMINISTRATOR}^i, \mathtt{USER}^i\},\\
S^{at}_O &= \{\mathtt{ADMINISTRATOR}^{at}, \mathtt{USER}^{at}\},\\
S^{ac}_O &= \{\mathtt{ADMINISTRATOR}^{ac}, \mathtt{USER}^{ac}\},\\
\Omega^i_O &= \Omega^i_{O;\epsilon,\mathtt{ADMINISTRATOR}^i} \cup \Omega^i_{O;\mathsf{string},\mathtt{USER}^i}\\
&= \{\mathtt{admin} : \longrightarrow \mathtt{ADMINISTRATOR}^i, \mathtt{user} : \mathsf{string} \rightarrow \mathtt{USER}^i\}\\
\Omega^{at}_O &\quad \textit{(omitted)}\\
\Omega^{ac}_O &= \{\mathtt{becomeAdmin} : \mathtt{ADMINISTRATOR}^i \rightarrow \mathtt{ADMINISTRATOR}^{ac},\\
&\qquad \mathtt{registerUser} : \mathtt{ADMINISTRATOR}^i \times \mathsf{string} \rightarrow \mathtt{ADMINISTRATOR}^{ac},\\
&\qquad \mathtt{deregisterUser} : \mathtt{ADMINISTRATOR}^i \times \mathsf{string} \rightarrow \mathtt{ADMINISTRATOR}^{ac},\\
&\qquad \mathtt{becomeAUser} : \mathtt{USER}^i \times \mathsf{string} \rightarrow \mathtt{USER}^{ac},\\
&\qquad \mathtt{setKeywords} : \mathtt{USER}^i \times \mathsf{set}(\mathsf{string}) \rightarrow \mathtt{USER}^{ac},\\
&\qquad \mathtt{setTargets} : \mathtt{USER}^i \times \mathsf{list}(\mathsf{string}) \rightarrow \mathtt{USER}^{ac},\\
&\qquad \mathtt{subscribe} : \mathtt{USER}^i \rightarrow \mathtt{USER}^{ac},\\
&\qquad \mathtt{unsubscribe} : \mathtt{USER}^i \rightarrow \mathtt{USER}^{ac},\\
&\qquad \mathtt{searchByKeywords} : \mathtt{USER}^i \times \mathsf{set}(\mathtt{litInfo}) \rightarrow \mathtt{USER}^{ac}\}
\end{aligned}
$$

as well as

`admin`, `user(Alice)`, and `user(Bob)`

as some examples of identity terms and

```
user(Alice).subscribe(),
user(Bob).setKeywords(keyw),
user(Alice).setTargets(targetList)
admin.register(Alice),
admin.register(Bob),
user(Alice).searchByKeywords(resultSet),
user(Bob).unsubscribe()
```

as some possible action terms provided `Alice` and `Bob` are constants of sort string, `keyw` is a constant of sort set(string), `targetList` is a constant of sort list(string), and $X_{\mathsf{set(litInfo)}} = \{\texttt{resultSet}, \ldots\}$ holds.

Assuming the interpretation for the sort string to be

$$A_{\mathsf{string}} = \{Alice, Bob, \ldots\},$$

this induces an instance signature, a part of which is given by

$$\begin{aligned} ID &= Id_{\texttt{ADMINISTRATOR}^i} \cup Id_{\texttt{USER}^i} \\ &= \{A_{\texttt{admin:}\longrightarrow\texttt{ADMINISTRATOR}^i}()\} \cup \\ &\quad \{A_{\texttt{user:string}\longrightarrow\texttt{USER}^i}(Alice), A_{\texttt{user:string}\longrightarrow\texttt{USER}^i}(Bob), \ldots\} \\ &= \{admin\} \cup \{Alice, Bob, \ldots\} \end{aligned}$$

for the set of object identities and

$$\begin{aligned} EAc_{Alice} &= \{\ldots, user(Alice).A_{\texttt{subscribe:USER}^i\longrightarrow\texttt{USER}^{ac}}(), \ldots\} \\ &= \{\ldots, user(Alice).subscribe(), \ldots\}, \\ EAc_{Bob} &= \{\ldots, user(Bob).A_{\texttt{subscribe:USER}^i\longrightarrow\texttt{USER}^{ac}}(), \ldots\} \\ &= \{\ldots, user(Bob).subscribe(), \ldots\}, \\ EAc_{admin} &= \{\ldots, admin.A_{\texttt{register:ADMINISTRATOR}^i\times\mathsf{string}\longrightarrow\texttt{ADMINISTRATOR}^{ac}}(Alice), \\ &\qquad admin.A_{\texttt{register:ADMINISTRATOR}^i\times\mathsf{string}\longrightarrow\texttt{ADMINISTRATOR}^{ac}}(Bob), \ldots\} \\ &= \{\ldots, admin.register(Alice), admin.register(Bob), \ldots\} \end{aligned}$$

for the elementary action alphabets, from which according to definition 5.15 local actions like

$$\begin{aligned} &\{user(Alice).subscribe()\} \in LAc_{user(Alice)}, \\ &\{user(Bob).subscribe()\} \in LAc_{user(Bob)}, \\ &\{admin.register(Bob)\} \in LAc_{admin}, \text{ or} \\ &\{admin.register(Alice), admin.register(Bob)\} \in LAc_{admin} \end{aligned}$$

and global actions like

$$\{\{user(Bob).subscribe()\}, \{admin.register(Bob)\}\}$$

or

$$\begin{aligned} &\{\{admin.register(Alice), admin.register(Bob)\} \\ &\ \{user(Alice).subscribe()\}, \{user(Bob).subscribe()\}\} \end{aligned}$$

can be constructed. Intuitively, the last global action states that two users ("Alice" and "Bob") inform the administrator ("admin") that they would like to use the system, who in turn registers them with it. The occurrence of this global action therefore synchronises all three involved objects. □

Interpretation structures for instance signatures are concurrent processes in which every instance is interpreted by a sequential process. The concurrent processes forming a system synchronise via the occurrence of global actions. Different interpretation structures exist which can be used for the description of concurrent object systems. The event structures used in the following—so-called *discrete prime event structures*—were introduced in [Win82, WN95] and other works. A detailed overview on different types of event structures as well as on other mathematical models which can be used as an interpretation structure can be found in [KF00].

We proceed by first giving some basic definitions regarding prime event structures and then connecting them to instance signatures with the help of labelling functions.

Definition 5.20 ((discrete) prime event structure) A prime event structure is a triple $E = (Ev, \rightarrow^*, \#)$ consisting of

1. a set Ev of elements referred to as *events*,
2. a partial order called *causality* as a binary relation $\rightarrow^* \subseteq Ev \times Ev$, and
3. a symmetric and irreflexive binary relation $\# \subseteq Ev \times Ev$ referred to as *conflict*.

Two events $e, e' \in Ev$ are related by *immediate* causality, $e \rightarrow e'$, if there are no other event occurrences in between, i.e. if for all $e'' \in Ev$ holds $(e \rightarrow^* e'' \rightarrow^* e') \Rightarrow (e'' = e \vee e'' = e')$.

Conflict propagates over *causality*, i.e. $(e\#e' \wedge e' \rightarrow^* e'') \Rightarrow (e\#e'')$ holds for all $e, e', e'' \in Ev$.

If additionally for every $e \in Ev$ the *local configuration*

$$\downarrow e = \{e' \mid e' \in Ev, e' \rightarrow^* e\}$$

is finite, then E is a *discrete* prime event structure. □

From now on, we will simply use *event structure* to refer to a discrete prime event structure.

The demand for finite local configurations roots in every system being started at some point in time, limiting every event to have a finite set of preceding events on which it may be causally dependent. The axiom concerning conflict propagation states that two events which are in causal dependency on two conflicting events will themselves be in conflict with each other.

Two events are said to be *concurrent* if they are neither in conflict nor in causal relation.

Definition 5.21 (concurrency) Given an event structure $E = (Ev, \rightarrow^*, \#)$, two events $e, e' \in Ev$ are said to be concurrent, $e \, co \, e'$, if and only if

$$\neg\,(e \rightarrow^* e' \vee e' \rightarrow^* e \vee e\#e')$$

holds. □

In order to be able to define a run of an object, of a system, or—later on—of a module, we first have to introduce the concept of a configuration:

Definition 5.22 (configuration) For a given event structure $E = (Ev, \rightarrow^*, \#)$ and a sub-event-structure $C \subseteq E$, C is a configuration in E if and only if

1. it is *conflict free*, i.e. $\neg\,(e \# e')$ holds for all $e, e' \in C$, and
2. it is *downwards closed*, that is, for any $e \in C$ and any $e' \in Ev$, $e' \rightarrow^* e$ implies $e' \in C$.

□

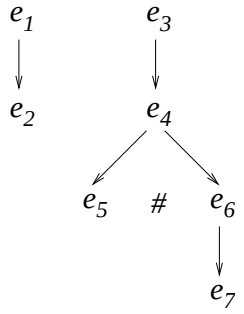
Note that the local configurations in definition 5.20 are in accordance with these criteria.

A run—also referred to as life cycle—can be defined as:

Definition 5.23 (run/life cycle) A *run* or *life cycle* in an event structure $E = (Ev, \rightarrow^*, \#)$ is a maximal configuration in E. □

Intuitively, a life cycle represents a complete run of an object, system, or module. The following example, a slightly modified version from [KF00], illustrates these concepts.

Example 5.24 (event structures, configurations, life cycles) Consider the event structure depicted by the following graphic, with conflict and immediate causality indicated by the corresponding symbols:



In order to keep examples simple, we refrain from explicitly representing the causal relation for reflexive pairs, i.e. $(e_i, e_i) \in \rightarrow^*$ or $\in \rightarrow$ will not be listed.

As e_5 and e_6 are in conflict with each other, an occurrence of e_5 excludes e_6 and vice versa. Since conflict propagates over causality, e_5 is also in conflict with e_7, because the latter is related to e_6 by (immediate) causality—as are e_1 and e_2, e_4 and e_5, e_4 and e_6, and e_3 and e_4.

Among others, events e_2 and e_5 are concurrent, as they are neither related by causality nor by conflict.

The local configuration of event e_6 is given by $\downarrow e_6 = \{e_3, e_4, e_6\}$. It is not maximal, and therefore not a life cycle. The same holds for $C_1 = \downarrow e_2 = \{e_1, e_2\}$. $C_2 = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ is not a configuration, as e_6 and e_5 are in conflict with

each other, and $C_3 = \{e_1, e_4, e_5\}$ is not a configuration either, as due to the missing e_3 it is not downwards closed. Finally, both $C_4 = \{e_1, e_2, e_3, e_4, e_5\}$ and $C_5 = \{e_1, e_2, e_3, e_4, e_6, e_7\}$ are maximal configurations and, thus, are both life cycles. □

As our intention is to interpret the objects of a system as sequential processes, we now define a sequential event structure. In doing so, we introduce a special initial event ε, which represents the minimal event of a sequential event structure with respect to the causality relation. With regard to the interpretation of object runs this event represents the state in which so far no other events have occurred.

Definition 5.25 (sequential event structure) A sequential event structure $E = (Ev, \rightarrow^*, \#)$ is an event structure which additionally meets the following criteria:

1. There exists a unique minimal event $\varepsilon \in Ev$.
2. Every local configuration $\downarrow e$ is totally ordered.
3. Any events not in causal relation are always in conflict with each other, i.e. $\forall\, e, e' \in Ev : e \# e' \Leftrightarrow \neg\, (e \rightarrow^* e' \vee e' \rightarrow^* e)$.

Events in $Ev_+ = Ev - \{\varepsilon\}$ are referred to as *proper* events. □

Sequential event structures can be viewed as a tree having the minimal event ε as its root. A life cycle in an event structure then corresponds to a path in this tree from the root to a leave (a *branch*), and the local configuration of an event e within such a run represents an object in a certain state, as it contains all the events which occurred prior to e.

According to definition 5.25 (3.), the conflict relation # is a derived concept. It is therefore sufficient to denote a sequential event structure by $E = (Ev, \rightarrow^*)$ instead of $E = (E, \rightarrow^*, \#)$, and in treating the causality relation $\rightarrow^*$ as the transitive and reflexive hull of $\rightarrow$ we may furthermore simply refer to it by $E = (Ev, \rightarrow)$.

Example 5.26 (sequential event structures) The event structure from example 5.24 is not sequential, but its substructures given by

$$\begin{aligned} E_1 &= (Ev_1, \rightarrow_1) = (\{e_1, e_2\}, \{(e_1, e_2)\}) \\ E_2 &= (Ev_2, \rightarrow_2) = (\{e_3, e_4, e_5, e_6, e_7\}, \{(e_3, e_4), (e_4, e_5), (e_4, e_6), (e_6, e_7)\}) \end{aligned}$$

can be transformed into sequential event structures by adding the minimal event to each of them and adjusting $\rightarrow$ accordingly, i.e.

$$\begin{aligned} E_1' &= (\{\varepsilon_1, e_1, e_2\}, \{(\varepsilon_1, e_1), (e_1, e_2)\}) \\ E_2' &= (\{\varepsilon_2, e_3, e_4, e_5, e_6, e_7\}, \{(\varepsilon_2, e_3), (e_3, e_4), (e_4, e_5), (e_4, e_6), (e_6, e_7)\}). \end{aligned}$$

□

After this more general introduction to event structures, we come back to instance signatures. For the interpretation of a given instance signature a *labelled* sequential event structure is used. In this process, a sequential event structure $E_i = (Ev_i, \rightarrow_i)$ is assigned to every object i of the instance structure. Here, the events from different structures are concurrent *per se*, and synchronisation happens by an event being part of several structures.

An event structure E for an instance signature $\Sigma_I = (Id, Ac)$ is the union of a family of sequential event structures, i.e. $E = \bigcup_{i \in Id}(Ev_i, \rightarrow_i)$. As the $E_i = (Ev_i, \rightarrow_i)$ are sequential, this Σ_I-event-structure E is sometimes referred to as a *locally* sequential event structure.

Definition 5.27 (Σ_I-event-structure) A Σ_I-event-structure $E = (Ev, \rightarrow)$ for an instance signature $\Sigma_I = (Id, Ac)$ is the union $\bigcup \vec{E}$ of an Id-indexed family of sequential event structures $\vec{E} = \{E_i\}_{i \in Id}$ for $E_i = (Ev_i, \rightarrow_i)$, where $\bigcup \vec{E} = (\bigcup_{i \in Id} Ev_i, \bigcup_{i \in Id} \rightarrow_i)$.

$Ev_+ = \bigcup \vec{Ev}_+$, $\vec{Ev}_+ = \{Ev_{i+}\}_{i \in Id} = \{Ev_i - \varepsilon_i\}_{i \in Id}$ is the *set* of proper events. □

Life cycles in a Σ_I-event-structure can be obtained analogously as the union of the life cycles for the respective sequential event structures. The condition of the slightly more general definition 5.23 is met automatically:

Definition 5.28 (distributed life cycle) Given a Σ_I-event-structure E, a distributed life cycle $L = (Lc, \rightarrow)$ in E is a sub-event-structure $L \subseteq E$, where L is the (component-wise) union of life cycles in the respective E_i, i.e. $L = \bigcup \vec{L} = \bigcup_{i \in Id} L_i \subseteq \bigcup_{i \in Id} E_i = \bigcup \vec{E}$ and $L_i \subseteq E_i$ holds for all $i \in Id$. □

In order to relate the events of an event structure to the actions of individual objects, a special labelling function is used which maps the events of a Σ_I-event-structure to the actions of the underlying instance signature Σ_I.

Definition 5.29 (Σ_I-event-structure labelling) Given a Σ_I-event-structure $E = (Ev, \rightarrow)$ for an instance signature $\Sigma = (Id, Ac)$, a Σ_I-event-structure labelling $\mu : Ev \rightarrow Ac$ over E is the union $\mu = \bigcup_{i \in Id} \mu_i$ of a family of mappings $\mu_i : Ev_i \rightarrow LAc_i \cup \{\{i.*\}\}, \mu_i(\varepsilon_i) = \{i.*\}$ with

$$\begin{aligned} \mu_j \cup \mu_k \;=\; & \{(e, \{l\}) \mid (e, l) \in \mu_j, e \in Ev_j, e \notin Ev_k\} \\ & \cup \{(e, \{l\}) \mid (e, l) \in \mu_k, e \in Ev_k, e \notin Ev_j\} \\ & \cup \{(e, \{l_j\} \cup \{l_k\}) \mid (e, l_j) \in \mu_j, (e, l_k) \in \mu_k, e \in Ev_j, e \in Ev_k\} \end{aligned}$$

which meets the following condition:

$$\forall\, e', e'' \in Ev_+ : (\exists\, e \in Ev_+ : e \rightarrow e' \wedge e \rightarrow e'' \wedge e' \neq e'') \Rightarrow \mu(e') \neq \mu(e'').$$

□

A Σ_I-event-structure labelling is a special type of a generic labelling function which, for a given event structure $E = (Ev, \rightarrow^*, \#)$ and an *arbitrary* set L, as a total function $l : Ev \rightarrow L$ maps each event from Ev into an element of the set L.

The codomain for the Σ_I-event-structure labelling μ—the set Ac—is the set of global actions determined in definition 5.15. Thus, every proper event of the Σ_I-event-structure is mapped to/labelled with such a global action. As such an action is a set of local actions, μ accordingly consists of a union of "local" mappings μ_i. A μ_i again maps a local event from Ev_{i+} to a local action LAc_i, the latter being a set of elementary action symbols. If the local event is an element of several of the Ev_i, then the set of all its images under the "local" mappings is used as the image under μ. Otherwise, it is the set containing the one local image. The minimal event ε_i is mapped to $*$ for all i. Communication between objects can now be represented by having the label of an event contain local actions from several objects.

Example 5.30 (Σ_I-event-structure, Σ_I-event-structure labelling) For some set of identities $Id = \{id1, id2\}$ and the corresponding sets of elementary actions $EAc_{id1} = \{ac^1_{id1}, ac^2_{id1}\}$ and $EAc_{id2} = \{ac^1_{id2}, ac^2_{id2}, ac^3_{id2}\}$, respectively, according to definition 5.15 we obtain the sets of local actions

$$\begin{aligned}
LAc_{id1} &= \left\{\{ac^1_{id1}\}, \{ac^2_{id1}\}, \{ac^1_{id1}, ac^2_{id1}\}\right\} \text{ and} \\
LAc_{id2} &= \Big\{\{ac^1_{id2}\}, \{ac^2_{id2}\}, \{ac^3_{id2}\}, \\
&\qquad \{ac^1_{id2}, ac^2_{id2}\}, \{ac^1_{id2}, ac^3_{id2}\}, \{ac^2_{id2}, ac^3_{id2}\}, \\
&\qquad \{ac^1_{id2}, ac^2_{id2}, ac^3_{id2}\}\Big\},
\end{aligned}$$

giving rise to the set of all local actions

$$\begin{aligned}
LAc = \Big\{&\{ac^1_{id1}\}, \{ac^2_{id1}\}, \{ac^1_{id1}, ac^2_{id1}\}, \\
&\{ac^1_{id2}\}, \{ac^2_{id2}\}, \{ac^3_{id2}\}, \\
&\{ac^1_{id2}, ac^2_{id2}\}, \{ac^1_{id2}, ac^3_{id2}\}, \{ac^2_{id2}, ac^3_{id2}\}, \\
&\{ac^1_{id2}, ac^2_{id2}, ac^3_{id2}\}\Big\},
\end{aligned}$$

from which in turn the set of global actions

$$\begin{aligned}
Ac = \Big\{&\Big\{\{ac^1_{id1}\}\Big\}, \ldots, \Big\{\{ac^1_{id2}, ac^2_{id2}, ac^3_{id3}\}\Big\}, \\
&\Big\{\{ac^1_{id1}\}, \{ac^2_{id1}\}\Big\}, \ldots, \Big\{\{ac^1_{id1}\}, \{ac^1_{id2}, ac^2_{id2}, ac^3_{id2}\}\Big\}, \\
&\Big\{\{ac^2_{id1}\}, \{ac^1_{id1}, ac^2_{id1}\}\Big\}, \ldots, \Big\{\{ac^2_{id1}\}, \{ac^1_{id2}, ac^2_{id2}, ac^3_{id2}\}\Big\}, \\
&\ldots, \\
&\Big\{\{ac^1_{id1}\}, \{ac^2_{id1}\}, \ldots, \{ac^1_{id2}, ac^2_{id2}, ac^3_{id2}\}\Big\}\Big\}
\end{aligned}$$

can be determined. If we assume two sequential event structures

$$\begin{aligned}
E_{id1} &= (\{\varepsilon_{id1}, a, e\}, \{(\varepsilon_{id1}, a), (a, e)\}) \text{ and} \\
E_{id2} &= (\{\varepsilon_{id2}, b, e, c\}, \{(\varepsilon_{id2}, b), (b, e), (e, c)\}),
\end{aligned}$$

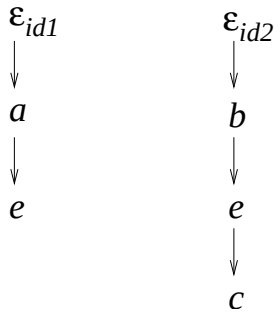


Figure 5.1: Graphical representation of two sequential event structures

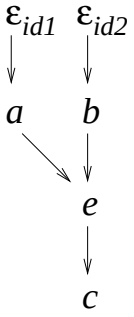


Figure 5.2: Graphical representation of the Σ_I-event-structure resulting for the sequential event structures illustrated in figure 5.1

graphically depicted in figure 5.1, then the Σ_I-event-structure resulting from them is

$$\begin{aligned} E &= (Ev, \rightarrow) = \bigcup \vec{E} = (\bigcup_{i \in Id} Ev_i, \bigcup_{i \in Id} \rightarrow_i) \\ &= (\{\varepsilon_{id1}, a, e, \varepsilon_{id2}, b, c\}, \{(\varepsilon_{id1}, a), (a, e), (\varepsilon_{id2}, b), (b, e), (e, c)\}), \end{aligned}$$

for which figure 5.2 shows a corresponding graphical representation.

Assuming further two "local" labelling functions

$$\begin{aligned} \mu_{id1} &= \{(\varepsilon_{id1}, \{id1.*\}), (a, \{ac^1_{id1}\}), (e, \{ac^2_{id1}\})\} \text{ and} \\ \mu_{id2} &= \{(\varepsilon_{id2}, \{id2.*\}), (b, \{ac^1_{id2}, ac^2_{id2}\}), (e, \{ac^1_{id2}\}), (c, \{ac^2_{id2}, ac^3_{id2}\})\} \end{aligned}$$

the resulting labelling function for E is

$$\begin{aligned} \mu = \Big\{ & (\varepsilon_{id1}, \{\{id1.*\}\}), (a, \Big\{\{ac^1_{id1}\}\Big\}), \\ & (e, \Big\{\{ac^2_{id1}\}, \{ac^1_{id2}\}\Big\}), \\ & (\varepsilon_{id2}, \{\{id2.*\}\}), (b, \Big\{\{ac^1_{id2}, ac^2_{id2}\}\Big\}), (c, \Big\{\{ac^2_{id2}, ac^3_{id2}\}\Big\})\Big\}. \end{aligned}$$

□

A Σ_I-event-structure E for a given instance signature Σ_I together with a labelling for E is referred to as a Σ_I-interpretation-framework, and a Σ_I-interpretation-structure denotes a labelled life cycle within this event structure.

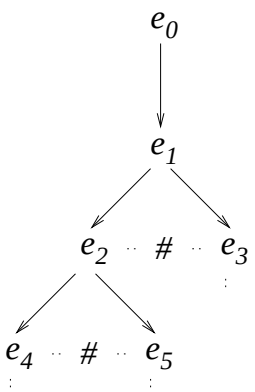
Definition 5.31 (Σ_I-interpretation-framework, Σ_I-interpretation-structure) An interpretation framework for an instance signature Σ_I is a labelled event structure, which is a pair $\overline{E} = (E, \mu)$ consisting of a Σ_I-event-structure E for the instance signature and a corresponding labelling function μ.

An interpretation structure in $\overline{E}$ is a labelled life cycle $\overline{L} = (L, \mu|_L)$, where $L = (Lc, \rightarrow)$ is a life cycle in E and $\mu|_L$ is the labelling function μ restricted to the events in Lc. □

In order to determine exactly one interpretation framework for a given instance signature whose events represent all possible occurrences of the corresponding actions, however, this definition is not sufficient. Intuitively, every proper event should represent the occurrence of an action, and should be marked accordingly. Furthermore, [KF00] states that a "labelling has to describe the occurrences of the actions [...], the current values of the attributes, and the enabled actions." Attention has to be paid not to synchronise arbitrary events, but only those which are not in conflict with each other. We refrain from discussing the technical details for the construction of such an interpretation framework, referring to [ES95] and [KF00] for further details, and instead provide an illustration of these concepts with regard to our example specification. Here, we already pick up the possibility to include attribute symbols in the labels as detailed in definition 5.52, since this inclusion will allow us to model locations at the formal level (cf. section 6.2.2, definition 6.50). Attributes and their values are then annotated as pairs of the form (a, b), with a being an attribute symbol of sort s and b an element in the carrier set of sort s,

Example 5.32 (Mobile-Agent-Based IR—interpretation framework) A labelled event structure can be derived for the instance signature sketched in example 5.19. In the following we present some parts of this resulting interpretation framework.

We first consider the sequential event structure characterising the behaviour of the object representing the administrator of the system. As in example 5.24, we use a graphical notation where the nodes of the graph represent events and the relations (immediate causality and conflict) are indicated by the same symbols as in the textual description. We therefore look at three possible life cycles for the object:

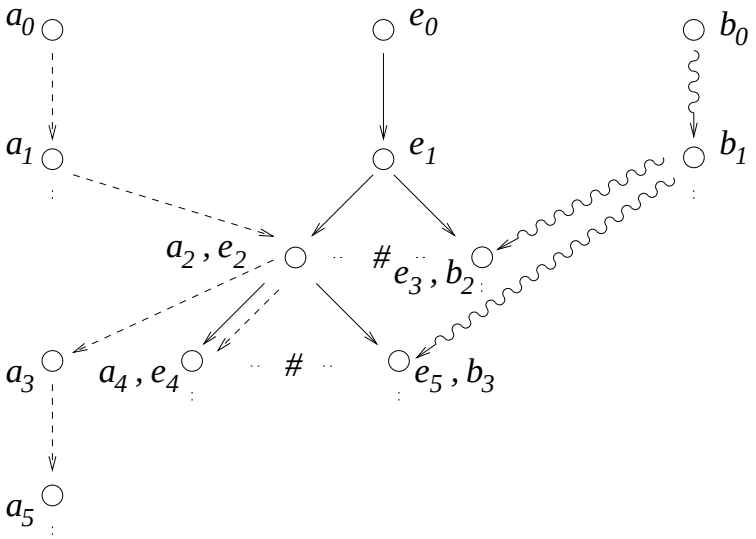


Let the labelling for the individual events be given by:

$$
\begin{aligned}
e_0 &\mapsto \{admin.*\} \\
e_1 &\mapsto \{admin.becomeAdmin()\} \\
e_2 &\mapsto \{admin.register(Alice)\} \\
e_3 &\mapsto \{admin.register(Bob)\} \\
e_4 &\mapsto \{admin.deregister(Alice)\} \\
e_5 &\mapsto \{admin.register(Bob)\}
\end{aligned}
$$

One possible life cycle is now that after being created (e_0) the object "admin" first registers "Alice" (e_1) as a user of the system and then does the same for "Bob" (e_3).

The following (concurrent) event structure contains the very same initial sequences for life cycles of the "admin" object, but additionally also one for a run of the object for user "Alice" and two for user "Bob":



In this last graphic, the causal relations belonging to the same object are indicated by using the same line style, where a different style is used for every object. Additionally, the events belonging to object "Alice" are marked by a_i and the ones for object "Bob" by b_i, those for the administrator object retaining the e_i marks. Instead of introducing another flag for shared events, those which occur in several structures are tagged with all the corresponding marks. A labelling may then be given as follows:

$$
\begin{aligned}
a_0 &\mapsto \{\{user(Alice).*, (user(Alice).name, x_0), \\
&\qquad (user(Alice).keywords, x_0), (user(Alice).targetList, x_0)\}\} \\
e_0 &\mapsto \{\{admin.*\}\} \\
b_0 &\mapsto \{\{user(Bob).*, (user(Bob).name, x_0), (user(Bob).keywords, x_0), \\
&\qquad (user(Bob).targetList, x_0)\}\}
\end{aligned}
$$

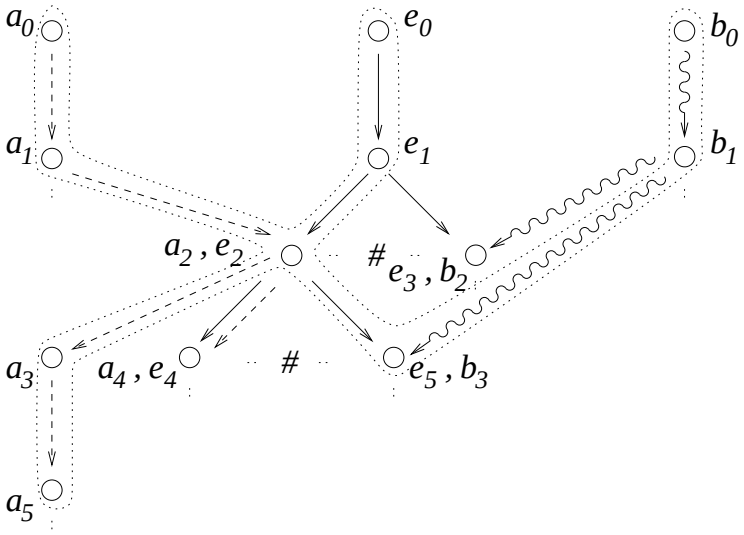
$$
\begin{aligned}
a_1 &\mapsto \{\{user(Alice).becomeAUser(x_1), (user(Alice).name, x_0),\\
&\qquad (user(Alice).keywords, x_0), (user(Alice).targetList, x_0)\}\}\\
e_1 &\mapsto \{\{admin.becomeAdmin()\}\}\\
b_1 &\mapsto \{\{user(Bob).becomeAUser(x_2), (user(Bob).name, x_0),\\
&\qquad (user(Bob).keywords, x_0), (user(Bob).targetList, x_0)\}\}\\
a_2, e_2 &\mapsto \{\{user(Alice).subscribe(), (user(Alice).name, x_1),\\
&\qquad (user(Alice).keywords, x_0), (user(Alice).targetList, x_0)\},\\
&\qquad \{admin.register(Alice)\}\}\\
e_3, b_2 &\mapsto \{\{user(Bob).subscribe(), (user(Bob).name, x_2),\\
&\qquad (user(Bob).keywords, x_0), (user(Bob).targetList, x_0)\},\\
&\qquad \{admin.register(Bob)\}\}\\
a_3 &\mapsto \{\{user(Alice).setKeywords(x_3), (user(Alice).name, x_1),\\
&\qquad (user(Alice).keywords, x_0), (user(Alice).targetList, x_0)\}\}\\
a_4, e_4 &\mapsto \{\{user(Alice).unsubscribe(), (user(Alice).name, x_1),\\
&\qquad (user(Alice).keywords, x_0), (user(Alice).targetList, x_0)\},\\
&\qquad \{admin.deregister(Alice)\}\}\\
e_5, b_3 &\mapsto \{\{user(Bob).subscribe(), (user(Bob).name, x_2),\\
&\qquad (user(Bob).keywords, x_0), (user(Bob).targetList, x_0)\},\\
&\qquad \{admin.register(Bob)\}\}\\
a_5 &\mapsto \{\{user(Alice).setTargets(x_4), (user(Alice).name, x_1),\\
&\qquad (user(Alice).keywords, x_3), (user(Alice).targetList, x_0)\}\},
\end{aligned}
$$

where x_0, x_1, and x_2 are constants of type string, x_3 is a constant of type set(string), and x_4 one of type list(string) with the following values:

$x_0 =$ “” (i.e. the empty string)
$x_1 =$ “Alice”
$x_2 =$ “Bob”
$x_3 = \{$“Mobility”, “Information Systems”$\}$
$x_4 = ($“litHost1”, “litHost5”, “litHost6”$)$

□

A distributed life cycle in this event structure is indicated in the following graphic by the enclosing dotted line.



The concepts presented in this section enable us to formalise the signature-related part of a TROLL specification. In the next section we will focus on the behaviour-related part.

5.1.2 Behaviour

In this section, we present a logic which can be used to express the behavioural part of a distributed object system—in particular TROLL specifications—in a formal way. It is a distributed linear temporal logic referred to as DTL (*Distributed Temporal Logic*) or D_0 (cf. e.g. [ES95, ECSD98, Ehr99, EC00]).

In the following, we first provide the DTL syntax. Based on this, we then introduce the term "system specification." Finally, we will use the labelled event structures discussed in the last section in order to define the semantics of DTL.

The fundamental idea of DTL is that every object i has a logic of its own, which consists of an internal logic—the so-called *Home Logic* H_i—and a communication logic C_i.

Definition 5.33 (object logic DTL—syntax) Given an extended data signature $\Sigma = (S, \Omega)$, a family $X = \{X_s\}_{s \in S}$ of sets of sort-indexed variables over Σ, and the set $T_\Sigma(X) = T_{\Sigma_D}(X) \cup T_{\Sigma_{Id}}(X) \cup T_{\Sigma_{At}}(X) \cup T_{\Sigma_{Ac}}(X)$ of terms over Σ, the syntax of DTL_Σ is defined as follows:

$$
\begin{array}{rcl}
\text{DTL}_\Sigma & ::= & \{\text{DTL}_i\}_{i \in T_{\Sigma_{Id}}(X)} \\
\text{DTL}_i & ::= & i.H_i \mid i.C_i \\
H_i & ::= & \text{ATOM}_i \mid \neg\,(H_i) \mid (H_i \vee H_i) \mid \exists\, x\,(H_i) \mid (H_i\, \mathcal{S}\, H_i) \mid (H_i\, \mathcal{U}\, H_i) \\
C_i & ::= & \odot T_{\Sigma_{Ac,i}}(X) \leftrightarrow j.(H_j) \text{ for } j \in T_{\Sigma_{Id}}, i \neq j \\
\text{ATOM} & ::= & true \mid T_{\Sigma_D}(X)\, \theta\, T_{\Sigma_D}(X) \mid \triangleright\, T_{\Sigma_{Ac,i}}(X) \mid \odot\, T_{\Sigma_{Ac,i}}(X)
\end{array}
$$

□

As already mentioned, every object i has its own local logic DTL_i, and DTL_Σ is the set of all local formulae constructible over the extended data signature Σ.

Atomic formulae of the internal logic H_i for object i are the boolean constant *true*, a comparison operator θ (e.g. $=, \neq, <, \geq, \ldots$) applied to two data terms, and the two state-dependent predicates $\triangleright$ (*enabled*) and $\odot$ (*occurred*) applied to action terms. Here, for an action term $\alpha = i.\omega(t_1, \ldots, t_n)$ the term $\triangleright\alpha$ states that the action α may happen next within the object i, and $\odot\alpha$ denotes that α has just occurred within i.

Using the logical connectives $\neg$ and $\vee$, the quantifier $\exists$, and the temporal operators $\mathcal{S}$ (*since*) and $\mathcal{U}$ (*until*), other formulae frequently used in various predicate and temporal logics can be constructed for H_i by applying the usual equivalences:

$$\begin{aligned}
false &\equiv (\neg\,(true)) \\
(\varphi \wedge \psi) &\equiv (\neg\,(\neg\,\varphi \vee \neg\,\psi)) \\
(\varphi \Rightarrow \psi) &\equiv (\neg\,\varphi \vee \psi) \\
(\varphi \Leftrightarrow \psi) &\equiv ((\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)) \\
(\forall\, x\, \varphi) &\equiv (\neg\,(\exists\, x\,(\neg\,\varphi))) \\
X\,\varphi &\equiv (false\,\mathcal{U}\,\varphi) \\
F\,\varphi &\equiv (\varphi \vee (true\,\mathcal{U}\,\varphi)) \\
G\,\varphi &\equiv (\neg\,(F(\neg\,\varphi))) \\
Y\,\varphi &\equiv (false\,\mathcal{S}\,\varphi) \\
P\,\varphi &\equiv (\varphi \vee (true\,\mathcal{S}\,\varphi)) \\
H\,\varphi &\equiv (\neg\,(P(\neg\,\varphi)))
\end{aligned}$$

$\varphi\,\mathcal{U}\,\psi$ states that φ will always be true from the next moment on until ψ becomes true for the next time; φ need not be true any more as soon as ψ holds, and ψ must eventually become true. The temporal operators derived from $\mathcal{U}$ denote *tomorrow* (X, *"next"*), *sometime in the future* (F), and *always in the future* (G).

Analogously, $\varphi\,\mathcal{S}\,\psi$ means that from a certain moment on φ was true until the previous moment since ψ became true the last time; it is not required that φ had already been true while ψ held, but ψ must have been true sometime. The temporal operators derived from $\mathcal{S}$ read *last* (Y, *yesterday*), sometime in the past (P), and *always in the past* (H).

The logic C_i is used to describe the communication with other objects from the point of view of the object i. Statements in C_i denote "if action α has just occurred, then it is admissible to conclude that action β has just happened within object j." The sign "$\leftrightarrow$" represents synchronous communication. It is in syntactical accordance with the module logic MDTL discussed below, which will be used to express the behavioural part of modularised object-oriented systems, i.e. of compound systems containing subsystems—in particular mobile or stationary entities.

The following example illustrates the construction of formulae:

Example 5.34 (Mobile-Agent-Based IR—formulae in DTL) For the extended data signature given in example 5.19 and a variable `s` of sort string, the following expressions are examples of syntactically correct formulae in DTL:

`user(s).(▷user(s).unsubscribe())`
`admin.(⊙admin.registerUser(s) ↔ user(s).(⊙user(s).subscribe()))`

The first formula states that the action `unsubscribe()` may happen next within object `user(s)`, while the second expresses that every time the action `registerUser` occurred within `admin`, also the action `subscribe` must have occurred within the `user`-object further identified by the value of the variable `s`. □

We now have all the prerequisites at hand to define our understanding of a system specification:

Definition 5.35 (system specification) A specification of a system is given by a pair $SysSpec = (\Sigma, \Phi)$, where Σ is an extended data signature and $\Phi \subseteq \text{DTL}_\Sigma$ is a set of formulae. These formulae are also referred to as *system axioms*. □

As already mentioned, labelled event structures can be used as the foundation for interpreting formulae developed in accordance with definition 5.33. More precisely, for a given extended data signature Σ, the formulae constructed in DTL_Σ are interpreted over an induced instance signature $\Sigma_I = (Id, Ac)$ and an interpretation structure $\overline{L} = (L, \mu|_L)$—as defined in the last section—in an interpretation framework $\overline{E} = (E, \mu)$ derived from Σ_I. Here, satisfaction of those formulae is considered for a local configuration $\downarrow$ in L with regard to a given assignment ρ. Accordingly, formulae are therefore interpreted in a state-dependent way.

In the next definition, we make use of the following abbreviations: $[\![i]\!]$ denotes the interpretation $[\![i]\!]^\rho_{A(\Sigma)}$ of an identity term $i \in T_{\Sigma_{Id}}(X)$, $[\![\alpha]\!]$ the interpretation $[\![\alpha]\!]^\rho_{A(\Sigma)}$ of an action term $\alpha \in T_{\Sigma_{Ac}}$, and $[\![t]\!]$ the interpretation $[\![t]\!]^\rho_{A(\Sigma)}$ of a data term $t \in T_\Sigma$.

Definition 5.36 (Satisfaction of DTL formulae) Assume given an extended data signature $\Sigma = (S, \Omega)$, a Σ-algebra $A(\Sigma) = (A(S), A(\Omega))$, and the instance signature $\Sigma_I = (Id, Ac)$ induced by Σ and $A(\Sigma)$. Furthermore, let $\overline{E} = (E, \mu)$ denote a labelled event structure, $\overline{L} = (L, \mu|_L)$ a labelled life cycle in $\overline{E}$, and α an action term from $T_{\Sigma_{Ac}}(X)$, then satisfaction of a formula in a local configuration $\downarrow e$ in $\overline{L}$ with respect to a given assignment ρ from the point of view of an object identified by $i \in T_{\Sigma_{Id}}(X)$ is defined inductively as follows:

1. $\overline{L}, \downarrow e, \rho \models i.(t_1 \theta t_2)$ iff $[\![t_1]\!] \theta [\![t_2]\!]$.
2. $\overline{L}, \downarrow e, \rho \models i.(\odot\alpha)$ iff $e \in Lc_{[\![i]\!]}$ and there is a local action $\alpha^{local} \in \mu(e)$ such that $[\![\alpha]\!] \in \alpha^{local}$ holds.
3. $\overline{L}, \downarrow e, \rho \models i.(\triangleright\alpha)$ iff $e \in Lc_{[\![i]\!]}$ and there is an event $e' \in Ev_i$ with $e \rightarrow_{[\![i]\!]} e'$ such that there exists a local action $\alpha^{local} \in \mu(e')$ and $[\![\alpha]\!] \in \alpha^{local}$ holds.

4. $\overline{L}, \downarrow e, \rho \models i.(\neg\, \varphi)$ iff $e \in Lc_{[\![i]\!]}$ and $\overline{L}, \downarrow e, \rho \models i.(\varphi)$ does *not* hold.

5. $\overline{L}, \downarrow e, \rho \models i.(\varphi \vee \psi)$ iff $e \in Lc_{[\![i]\!]}$ and $\overline{L}, \downarrow e, \rho \models i.(\varphi)$ or $\overline{L}, \downarrow e, \rho \models i.(\psi)$ holds.

6. $\overline{L}, \downarrow e, \rho \models i.(\exists x \varphi)$ iff $e \in Lc_{[\![i]\!]}$ and $\overline{L}, \downarrow e, \rho' \models i.(\varphi)$ holds for an x-equivalent assignment ρ'.

7. $\overline{L}, \downarrow e, \rho \models i.(\psi\, \mathcal{S}\, \varphi)$ iff $e \in Lc_{[\![i]\!]}$ and there is an event $e' \in Ev_i$ with $e' \rightarrow^+ e$ such that $\overline{L}, \downarrow e', \rho \models i.(\varphi)$ holds and there is a finite set ("path") $\{e_1, \ldots, e_n\} \subseteq Ev_i$ with $e' = e_1 \rightarrow \ldots \rightarrow e_n = e$ where $\overline{L}, \downarrow e_p, \rho \models i.(\psi)$ holds for every $e_p, 1 < p < n$.

8. $\overline{L}, \downarrow e, \rho \models i.(\varphi\, \mathcal{U}\, \psi)$ iff $e \in Lc_{[\![i]\!]}$ and there is an event $e' \in Ev_i$ with $e \rightarrow^+ e'$ such that $\overline{L}, \downarrow e', \rho \models i.(\psi)$ holds and there is a finite set ("path") $\{e_1, \ldots, e_n\} \subseteq Ev_i$ with $e = e_1 \rightarrow \ldots \rightarrow e_n = e'$ where $\overline{L}, \downarrow e_p, \rho \models i.(\varphi)$ holds for every $e_p, 1 < p < n$.

9. $\overline{L}, \downarrow e, \rho \models i.(\odot\alpha_1 \leftrightarrow j.(\varphi))$ iff $i \neq j, e \in Lc_{[\![i]\!]}, e \in Lc_{[\![j]\!]}$ and $\overline{L}, \downarrow e, \rho \models i.(\odot\alpha_1)$ implies $\overline{L}, \downarrow e, \rho \models j.(\varphi)$.

□

According to 1., *data* terms (only) are interpreted in a state-independent way. The second rule says that an elementary action denoted by the action term α occurred for a local configuration of event e within the context of the object identified by the identity term i, provided e is an event of object i and the label of e contains a local action α^{local} comprising α. An action may occur next if there is a directly successional event in which the action occurred (3.). The event representing the subsequent state does not have to be part of the life cycle in focus. Rules 4, 5, and 6 determine the interpretation of logical connectives and quantifiers in the usual way, considering an assignment ρ' to be x-equivalent to the assignment ρ if $\rho(y) = \rho'(y)$ holds for all $y \neq x$. 7. deals with the interpretation of the temporal operator $\mathcal{S}$, stating that "ψ held since φ" if for a set of events there is one e' genuinely preceding e where φ was true, and in all subsequent events until the immediate predecessor of e ψ held. Rule 8 gives a similar meaning in the opposite direction for the "until" operator $\mathcal{U}$. Finally, 9. describes the interpretation of synchronous communication between two different objects. The corresponding formula is satisfied if and only if the occurrence of the action term within the calling object implies satisfaction of the formula for the object being called.

Having considered satisfaction of a formula with regard to a local configuration $\downarrow e$, i.e. a certain state, we now turn to satisfaction of formulae in life cycles and interpretation frameworks:

Definition 5.37 (satisfaction) Given a labelled event structure $\overline{E} = (E, \mu)$ as an interpretation framework for an instance signature Σ_I and a labelled distributed life cycle $\overline{L} = (L, \mu|_L), L = (Lc, \rightarrow)$, as an interpretation structure in $\overline{E}$, satisfaction of a local formula $i.(\varphi)$ is defined by

1. $\overline{L} \models i.(\varphi)$ iff $\forall e \in Lc$ and all assignments ρ holds $\overline{L}, \downarrow e, \rho \models i.(\varphi)$,
2. $\overline{E} \models i.(\varphi)$ iff for all $\overline{L} \in \overline{E}$ holds $\overline{L} \models i.(\varphi)$.

□

A distributed life cycle satisfies a local formula $i.(\varphi)$ if and only if for any assignment the formula is satisfied in every local configuration of the life cycle, and an interpretation framework satisfies a formula if and only if every life cycle of the framework satisfies it.

An event structure which satisfies all the axioms of a system specification $SysSpec = (\Sigma, \Phi)$ is called a *model of SysSpec,* and the set of all these event structures is referred to as a *model class.* The distributed life cycles in a model form interpretation structures, where every life cycle represents one run of the system.

With these definitions as the basis, we now turn to the specification of systems containing subsystems, applying the module concepts developed in [KF00] and [Eck01] for structuring large specifications. In the next chapter, we will then use the hierarchical structures as a means to reflect locations, and on top of this develop constructs to express mobility and changing connectivity.

The following section is structured like this one: the first subsection considers the signature-related part of modules, which are the units used for structuring a specification. A module consists of a kernel comprising one or more object classes, corresponding instance declarations and relations and interactions between those instances, and it may import other modules or export certain functionality. Modules which neither import other modules nor export something correspond to the object systems discussed so far. Accordingly, the Module Distributed Temporal Logic MDTL discussed in subsection 5.2.2 used to formulate axioms regarding the behaviour of modules carries forward basic ideas developed for DTL.

5.2 Subsystem Specification

The focus of the previous section was on object systems, which we considered to contain a number of concurrent objects, each of them again denoting a set of possible sequential life cycles. We now turn to systems consisting of a number of concurrent modules, each of them denoting a set of possible distributed life cycles. The definitions we provide will be in close correspondence to the ideas developed previously, and will enable us to describe modular object-oriented systems in a formal way.

The simplest type of a module is a basic module, consisting of a module kernel (or just kernel for short) and an export part. The kernel may comprise one or more

object classes, a number of instance declarations of objects from these classes, and the definition of relationships and interactions between them. The export part consists of a—possibly empty—set of export declarations. A formal description of a kernel signature is given by an extended data signature, which in contrast to those in section 5.1 contains additional module sorts and operation symbols. The export part describes those items of the kernel which should be visible from outside, and accordingly also constitute signatures. For these signatures, suitable inclusion morphisms are required relating them to respective parts of the kernel. The signature of a basic module therefore consists of a kernel signature, the signatures from the export part, and the corresponding inclusion morphisms.

Basic modules are said to be *closed*, *open*, or *completely open* if their export part is empty, non-empty but also not exporting the entire kernel, or non-empty and exporting the entire kernel, respectively. The object systems discussed in the previous section therefore correspond to closed basic modules.

An export declaration also determines a basic module, which matches a part of the kernel. These completely open basic modules derived from export declarations are referred to as *views* or *view modules*.

In contrast to basic modules complex modules contain other modules, which themselves may again be complex. The complex modules discussed in [KF00] and [Eck01] consist of a body, an export and import part and also a parameter part, the latter meant to support reuse and application specific adaptation. However, as our concern is merely to use hierarchical structures as a means to reflect location, and as these structures already result from import and export relations, in the following we neglect the issues regarding parameterisation. The import part consists of a finite set of basic modules, the export part is structured exactly as described above, and the body corresponds to a basic module.

In order to be able to formalise complex modules, first extended signatures of kernels will be defined containing additional module sorts and operators. Next, based on this, module signatures can be determined consisting of an extended kernel signature, the signature of a basic module representing the body, a set of signatures for the imported view module, and a set of export signatures.

Since basic modules correspond to general modules with an empty import and export part, they, too, can be described by module signatures. A complex module will have a non-empty import part, but need not have a body.

For the specification of a module's axioms, the Modular Distributed Temporal Logic MDTL can be used, which applies the ideas developed for objects in DTL to modules: every module of a system has a logic associated with it consisting of an internal or *Home Logic* and a communication logic. Just like DTL, MDTL is interpreted over labelled event structures.

5.2.1 Signatures

As already mentioned, in this subsection the signature-related part of module specifications coarsely outlined above will be defined. We start by considering basic modules consisting of a kernel and an export part. The signature of such

a kernel corresponds to an extended data signature augmented by additional module sorts and operators.

Definition 5.38 (kernel signature) The signature of a module kernel $\Sigma_K = (S, \Omega)$ is an extended data signature as described in definition 5.8, which additionally contains the following sorts and operators:

1. The set of sorts S is extended by the sorts $S_M = S_M^e \cup S_M^+$. The sorts in S_M^+ are referred to as *internal module sorts* and the sorts in S_M^e as *export module sorts*. S_M is a union of module sorts with $S_M^e \cap S_M^+ = \{\alpha\}$, where α is called the *local module sort*. Hence, $S = S_D \cup S_O^i \cup S_O^{at} \cup S_O^{ac} \cup S_M$ holds for the module kernel signature $\Sigma_K = (S, \Omega)$.

2. The family of sets of operators is extended by Ω_M, leading to $\Omega = \Omega_D \cup \Omega_O^i \cup \Omega_O^{at} \cup \Omega^{ac} \cup \Omega_M$, in such a way that for every $m \in S_M$ a unique instance symbol exists in $\Omega_{M;\epsilon,m}$.

□

The additional operation symbols are instance symbols for modules. Analogously to data signatures, module kernel signatures are interpreted over Σ-algebras.

With definition 5.3 and 5.11 we have shown how to construct terms from a given (extended) data signature and a family of sets of sort-indexed variables. Regarding kernel signatures, we can obtain terms in much the same way, only additionally deriving instance terms for modules:

Definition 5.39 (module instance terms) Let $\Sigma_K = (S, \Omega)$ be a kernel signature and $X = \{X_s\}_{s \in S}$ be a family of sets of sort-indexed variables, then

$$T_{\Sigma_M}(X) = \{T_{\Sigma_{M,s}}(X)\}_{s \in S_M}$$

denotes the family of sets of module-sort-indexed module instance terms, or module terms for short, obtained in the same way as data terms and identity terms. □

As the only module operations available are constants, a module instance term will always be given by a constant of a given sort $m \in S_M$.

With module instance terms at hand, we can now also give a first definition of terms derivable from a kernel signature:

Definition 5.40 (Σ_K-Terms) Given a kernel signature $\Sigma_K = (S, \Omega)$ and a family $X = \{X_s\}_{s \in S}$ of sets of sort-indexed variables, the family $T_\Sigma(X)$ of sets derivable from the extended data signature Σ contained within Σ_K is extended by $T_{\Sigma_M}(X)$, yielding

$$\begin{aligned} T_{\Sigma_K}(X) &= T_\Sigma(X) \cup T_{\Sigma_M}(X) \\ &= T_{\Sigma_D}(X) \cup T_{\Sigma_{Id}}(X) \cup T_{\Sigma_{At}}(X) \cup T_{\Sigma_{Ac}}(X) \cup T_{\Sigma_M}(X) \end{aligned}$$

as the family of sets of terms derivable from the kernel signature. □

It is possible to define structure-preserving transformations (morphisms) between kernel signatures, and, based on this, export signatures can be introduced:

Definition 5.41 (export signatures) $E = (\Sigma_{K_2}, \mu)$ is an export signature for a signature Σ_{K_1} of a module kernel if there exists an inclusion morphism μ from Σ_{K_2} to Σ_{K_1}—i.e., the elements from Σ_{K_2} can be embedded into Σ_{K_1}—and the local module sorts α_1 and α_2 from Σ_{K_1} and Σ_{K_2}, respectively, meet the following conditions:

1. $\alpha_2 \in S^e_{M_1}$,
2. $\alpha_1 \in S^+_{M_2}$,
3. $S^e_{M_2} = \{\alpha_2\}$,
4. $\alpha_2 = \alpha_1$ if and only if $\Sigma_{K_2} = \Sigma_{K_1}$.

Furthermore, for any two export signatures E_1 and E_2 over Σ_K with local module sorts β_1 and β_2, respectively, $E_1 \neq E_2$ if and only if $\beta_1 \neq \beta_2$ has to hold. □

Using kernel signatures and export signatures, signatures of basic modules can now be defined, where the inclusion morphisms embed the export signatures into the module kernel signatures:

Definition 5.42 (basic module signature) The signature of a basic module is a pair $\Theta = (\Sigma_K, Exp)$, where Σ_K is the signature of a module kernel and Exp is a set of different export signatures over Σ_K such that either Exp is empty or that $Exp = \{E_1, \ldots, E_n\}$, with n being the maximal number of export signatures definable over Σ_K. □

The maximal number of export signatures which can be defined over the module kernel signature is equal to the number of elements in $S^e_M \setminus \{\alpha\}$, provided this number is greater than 1. The underlying idea for this is that either nothing is exported at all, or the following rule is respected: If α is the only module sort in the set of export module sorts S^e_M, then the complete signature of the module kernel is exported; otherwise for every module sort contained in S^e_M except α, Exp comprises a corresponding export signature.

An export declaration determines a completely open basic module, which usually corresponds to a part of the kernel of the underlying module. In other words, if $\Theta = (\Sigma_K, Exp)$ is a basic module signature and $E_i = (\Sigma_K, \mu_i)$ is an export signature from Exp, then $\Theta_i = (\Sigma_{K_i}, \{(\Sigma_{K_i}, id)\})$ can be derived from E_i in a unique way. Additionally $\Theta_i = \Theta$ holds provided $\Sigma_{K_i} = \Sigma_K$. A basic module signature derived from an export signature is an example of a signature whose sets of module sorts are not restricted to the local module sort and the export module sorts, as it also contains a reference to underlying module signature: $S_{M_i} = \{\alpha_i, \alpha\}$.

Based on the definitions and explanations given so far, we can now define signatures of body and view modules, which then form a part of a general module signature:

Definition 5.43 (body module signature) A basic module signature $\Theta = (\Sigma_K, Exp)$ is a body module signature if $Exp = \{(\Sigma, id)\}$ and S_M is a singleton. □

Definition 5.44 (view module signature) Given a basic module signature $\Theta = (\Sigma_K, Exp)$, Θ_v is a view module signature of Θ if and only if there exists an E_v in Exp such that Θ_v is the basic module signature induced by E_v, or Θ_v is isomorphic to some Θ_w which has been determined by an export signature in Exp. □

A view module signature of a basic module signature Θ is therefore a signature that has been determined by one of Θ's export signatures or corresponds to a renaming of such a signature.

While a basic module consists of a kernel and an export part, a general module (which may be complex, i.e. contain other modules) also comprises an import part and a body. Accordingly, the kernel signature has to be extended by corresponding module sorts for the additional parts:

Definition 5.45 (extended kernel signature) An extended kernel signature is a (simple) kernel signature $\Sigma_K = (S, \Omega)$ according to definition 5.38 which additionally meets the following conditions: The set S_M^+ from $S_M = S_M^e \cup S_M^+$ is extended in such a way that $S_M^+ = S_M^i \cup S_M^\times$ for disjoint sets S_M^i and $S_M^\times$ of sorts, and the local module sort α is an element of $S_M^\times$. Furthermore, $S^\times = S_M^b \cup S_M^o$ has to hold with $S_M^b \cap S_M^o = \{\alpha\}$ and $S_M^b = \{\alpha, \beta\}$, where β is referred to as the module body sort. □

This definition states that a new set of module sorts is introduced for each the import part (S_M^i) and the module body (S_M^b), where the latter contains the module body sort as well as the local module sort. [Eck01] and [KF00] also introduce a parameter part S_M^p as another disjoint subset of S_M^+, but as parameterisation of modules is of no concern to this thesis, we refrain from carrying this item along here. If the parameter part should be required in future work, it can be included in a straight forward way.

Terms over an extended kernel signature are constructed in the same way as terms over simple kernel signatures (cf. definition 5.40), with the addition that for the former the family $T_{\Sigma_K}(X)$ of sets of module-sort-indexed module instance terms now also includes instance terms for the import modules and the body module. We denote these instance terms by Mod_Σ:

Definition 5.46 (module instance terms for import and body modules) Let $S^{ib} = S_M^i \cup S_M^b$. Mod_Σ denotes the S^{ib}-indexed subfamily of sets of import and body module terms, i.e.

$$Mod_\Sigma = \bigcup_{s \in S_M^i \cup S_M^b} T_{\Sigma_{M,s}}.$$

□

Export signatures can be given for extended kernel signatures much like for simple kernel signatures:

Definition 5.47 (export signatures for extended kernel signatures) Let Σ_{K_1} denote an extended kernel signature, Σ_{K_2} a (simple) kernel signature, α_1 and α_2 the local module sorts of Σ_{K_1} and Σ_{K_2}, respectively, and μ an inclusion morphism from Σ_{K_2} to Σ_{K_1}. $E = (\Sigma_{K_2}, \mu)$ is an export signature over Σ_{K_1} if and only if the conditions 1. to 4. from definition 5.41 are met. □

Here, the only difference with regard to definition 5.41 is that Σ_{K_1} is usually an extended kernel signature as opposed to the exported signature, which is always a simple kernel signature. Accordingly, the signature derivable from an export signature will always be a basic module signature, and view module signatures will therefore always be basic module signatures. Hence, an export signature can only comprise the entire underlying signature—i.e. $\Sigma_{K_1} = \Sigma_{K_2}$—if Σ_{K_1} is a simple kernel signature.

We are now able to define a module signature in general:

Definition 5.48 (module signature) A module signature is a quadruple $\Theta = (\Sigma_K, \Sigma_{K_{bod}}, Imp, Exp)$, where Σ_K is an extended kernel signature, $\Sigma_{K_{bod}}$ is a body module signature with local module sort *bod*, $Imp = \{\Sigma_{K_{i1}}, \ldots, \Sigma_{K_{im}}\}$ is a finite, possibly empty set of import view module signatures and Exp is a set of different export signatures over Σ_K such that either $Exp = \{\}$ or $Exp = \{E_1, \ldots, E_n\}$ with n being the maximal number of export signatures definable over Σ, if Θ meets the following conditions:

1. The import view module signatures are view modules according to definition 5.44.
2. $bod \in S^b_M$ and there exists an inclusion morphism μ_b from $\Sigma_{K_{bod}}$ to Σ_K.
3. $S^i_M = \{i1, \ldots, im\}$ and there exists an inclusion morphism μ_{ij} from $\Sigma_{K_{ij}}$ to Σ_K for every $1 \leq j \leq m$.
4. $m \notin S_{M_{i1}} \cup \ldots \cup S_{M_{i1}}$ holds for every $m \in S^e_M$.
5. For α denoting the local module sort of Σ_K, $bod = \alpha$ holds if and only if Σ_K is a (simple) kernel signature.
6. The extended data signature contained within the kernel signature Σ_K results from the union of the extended data signatures for the import view module signatures and the body module signature.

□

In order to relate module signatures to event structures, the intermediate step of applying instance signatures as a bridge between extended data signatures and event structures used in section 5.1 is not required here, as multiple instantiation of modules is not permitted and an analogy to the concept of object class and

instance does not exist. Accordingly, module signatures can be directly assigned to the event structures defined previously, the relation again being established via a suitable labelling function.

The labelled event structures for basic modules are developed analogously to those of object systems, i.e. first sequential event structures for the individual objects are constructed and then assembled to concurrent units. For technical details regarding this construction we once more refer to [KF00]. Intuitively, again events get labelled with (global) actions having just occurred or being enabled, and the current values of the attributes.

With the concepts presented in this section, we are now able to formally describe the signature-related part of modular TROLL specifications and can turn to considering their behaviour.

5.2.2 Behaviour

In this section, we present a logic which can be used to express the behavioural part of modular system specifications in a formal way. It is the Module Distributed Temporal Logic (MDTL) from [KF00], which extends the Distributed Temporal Logic (DTL) from section 5.1.2

The main difference of MDTL with respect to DTL is that in MDTL every *module* has a logic of its own. However, as the modular approach also permits to regard individual objects as modules, the ideas of DTL can also be expressed in MDTL, and furthermore the latter additionally provides the possibility to postulate axioms at a superordinate level—the module level.

Another difference is that MDTL provides a means to express concurrency directly via a dedicated operator (*concurrency operator*). In DTL, this is neither possible nor required, as objects are considered to be sequential, having no internal concurrency. For modules, however, this is usually the case, and, accordingly, the associated logic should provide the possibility to represent concurrency within formulae.

Similarly to DTL, MDTL consists of two parts: a communication logic, which allows to express the interaction between modules, and the internal logic—again referred to as *Home Logic*—providing means to make assertions about a module itself.

In this thesis, we use a slightly modified version of the logic, similar to the one presented in [Eck01], which restricts the communication logic to the constructs related to the expression of synchronous communication (while the version in [KF00] also allows to describe asynchronous sending and receiving of messages):

Definition 5.49 (Module Distributed Temporal Logic MDTL) Let a module signature $\Theta = (\Sigma_K, \Sigma_{K_{bod}}, Imp, Exp)$ be given, with α denoting the local module sort of Σ_K. Let $l \in T_{\Sigma_{M,\alpha}}$ be the local module term of Θ, let $\beta \in S^i_M \cup S^b_M$ and $\Sigma_{K_\beta} = (S_\beta, \Omega_\beta)$ be the (extended) kernel signature within Θ with local module sort β. Finally, let $X = \{X_s\}_{s \in S}$ be a family of sort-indexed variables, $x \in X_s$, and let the terms $T_{\Sigma_K}(X)$ be constructed over Σ_{K_β}. The

syntax of MDTL_Θ is defined as follows:

$$\begin{aligned}
\text{MDTL}_\Theta &::= \text{MDTL}^l \mid l.H_l \\
\text{MDTL}^l &::= \{\text{MDTL}^l_m\}_{m \in Mod_{\Sigma,\beta}} \\
\text{MDTL}^l_m &::= m.H_m \mid m.C^l_m \\
H_m &::= \text{ATOM}_m \mid \neg\,(H_m) \mid (H_m \Rightarrow H_m) \mid \exists\, x\,(H_m) \mid \\
&\quad (H_m\, \mathcal{U}_\forall\, H_m) \mid (H_m\, \mathcal{U}_\exists\, H_m) \mid (H_m\, \mathcal{S}\, H_m) \mid \Delta(H_m) \\
C^l_m &::= \odot T_{\Sigma_{Ac}}(X) \leftrightarrow k.(H_k) \text{ for } k \in Mod_\Sigma, k \neq m, k \neq l \\
\text{ATOM} &::= true \mid T_{\Sigma_D}(X)\, \theta\, T_{\Sigma_D}(X) \mid \triangleright\, T_{\Sigma_{Ac}}(X) \mid \odot\, T_{\Sigma_{Ac}}(X)
\end{aligned}$$

□

The slight modification with regard to the versions of this logic in [Eck01] or [KF00] is that we do not require $m \neq l$ for $\text{MDTL}^l ::= \{\text{MDTL}^l_m\}_{m \in Mod_{\Sigma,\beta}}$, as our concern is to use the module hierarchies in order to provide means to make locations explicit, instead of using it for encapsulation and hiding of internal details, which is a focus in the former works.

Every module Θ has a module logic MDTL_Θ associated with it. The first production reflects the two possible views on modules considered in [Eck01] and [KF00]: if a complex module Θ should be regarded as a collection of basic modules, then MDTL^l should to be used to express its properties. However, if Θ should be considered as a basic module or a complex module containing a collection of objects, then H^l should be applied, where l denotes the local module term (cf. definition 5.39) for Θ of sort α.

MDTL^l assigns an individual local logic MDLT^l_m to every basic module from Θ, where m is the corresponding identity term. This local logic reflects the structure of DTL, as it consists of a communication logic C^l_m and an internal logic H_m (the *Home Logic*).

The internal logic H is a first-order temporal logic, providing the additional concurrency operator Δ mentioned above. This operator permits, e.g., to formulate statements from the point of view of a complex module about the concurrent behaviour of basic modules contained in it. The atomic formulae of H are structured just like atomic formulae in DTL. Formulae in H can be constructed by successively applying the connectives $\neg$ and $\Rightarrow$ and the quantifier $\exists$ known from conventional predicate logics, the temporal operators $\mathcal{U}$ (*until*) and $\mathcal{S}$ (*since*), and the concurrency operator Δ to atomic formulae. Within the temporal operators, [KF00] distinguishes between a *for all until* $\mathcal{U}_\forall$ and an *exists until* $\mathcal{U}_\exists$ in order to reflect the branching-time nature of the temporal logic.

The communication logic C^l_m is used to describe interactions between objects of different modules comprised by Θ, thus expressing their communication. Here, similarly to DTL, statements are formulated from the perspective of a certain module. Communication within a module is expressed using the internal logic H.

Depending on which of the two ways of looking at a complex module is chosen, the same property can be expressed by different formulae. Both [KF00]

and [Eck01] discuss these possibilities in depth and provide rules for translating formulae from one logic into the other, for example:

Definition 5.50 (transformation of DTL formulae into MDTL formulae) Let the specification ψ of a basic subsystem be given and let Φ_ψ denote the set of all the behaviour axioms derivable from ψ. Let ω be the specification of a complex subsystem, id the instance term of ψ in ω, and Φ_ω the set of all the behaviour axioms derivable from ω, then

- if $o.(\varphi) \in \Phi_\psi$ and $\varphi \in H_o$, then $id.(\varphi) \in \Phi_\omega$,
- if $o.(\varphi) \in \Phi_\psi$ and $\varphi = \varphi_1 \leftrightarrow j.(\varphi_2) \in C_o$, then $id.(\varphi_1 \Rightarrow \varphi_2) \in \Phi_\omega$

holds for all DTL formulae from Φ_ψ. □

Just like for DTL, the logical constant $false$ and other connectives like $\wedge, \vee$, or $\Leftrightarrow$ frequently used in propositional calculi can be expressed in terms of $\neg$ and $\Rightarrow$ by applying the following equivalences:

$$\begin{aligned} false &\equiv (\neg\,(true)) \\ (\varphi \wedge \psi) &\equiv (\neg\,(\varphi \Rightarrow (\neg\,\psi))) \\ (\varphi \vee \psi) &\equiv ((\neg\,\varphi) \Rightarrow \psi) \\ (\varphi \Leftrightarrow \psi) &\equiv ((\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)) \\ (\forall\, x\, \varphi) &\equiv (\neg\,(\exists\, x\,(\neg\,\varphi))) \end{aligned}$$

Similarly, other common temporal operators like X (*next*), F (*sometime in the future*), G (*always in the future*), Y (*yesterday*), P (*sometime in the past*), and H (*always in the past*) can be derived from $\mathcal{U}$ and $\mathcal{S}$ using the following equivalences:

$$\begin{aligned} X\,\varphi &\equiv (false\,\mathcal{U}\,\varphi) \\ F\,\varphi &\equiv (\varphi \vee (true\,\mathcal{U}\,\varphi)) \\ G\,\varphi &\equiv (\neg\,(F(\neg\,\varphi))) \\ Y\,\varphi &\equiv (false\,\mathcal{S}\,\varphi) \\ P\,\varphi &\equiv (\varphi \vee (true\,\mathcal{S}\,\varphi)) \\ H\,\varphi &\equiv (\neg\,(P(\neg\,\varphi))) \end{aligned}$$

We now have all the prerequisites at hand to define our understanding of a module specification:

Definition 5.51 (module specification) Let Θ denote a module signature for an extended kernel signature Σ and local module sort α. Furthermore, $l \in T_{\Sigma_{M,\alpha}}$ be the identity term of Θ, then a module specification is a pair $ModSpec = (\Theta, \Phi)$, where the axioms Φ of the module are given by a set of MDTL_Θ formulae. □

Quite like for DTL formulae, labelled event structures provide the basis for interpreting MDTL formulae. However, this time we explicitly mention the possibility to include attributes and their values in the labels of the event structure for a module:

Definition 5.52 (module labelled event structure) Assume given a module signature Θ for an extended kernel signature Σ, an event structure $E = (Ev, \rightarrow^*, \#)$, a Σ-algebra $A(\Sigma) = (A(S), A(\Omega))$ over Σ, and an interpretation $[\![]\!]$ for a variable assignment $\rho : \{\rho_s : X_s \rightarrow A_s\}_{s \in S}$. Let $\mathbf{Ac} = [\![T_{\Sigma_{Ac}}]\!]$ and $\mathbf{At} = [\![T_{\Sigma_{At}}]\!]$. A module labelled event structure for Θ under $[\![]\!]$ is given by $M_\Theta[\![]\!] = (E, \mu)$, where $\mu : Ev \rightarrow 2^{\mathbf{Ac} \cup (\mathbf{At}_s \times A_s)}$ with $s \in S^i$. □

In this definition, an event is mapped into a set containing action symbols or pairs of the form (a, b), where a is an attribute symbol—i.e. the interpretation of a closed attribute term—of sort s and b is an element from the carrier set for sort s.

Satisfaction of MDTL formulae is defined similarly to the one of DTL formulae, namely in that a formula of MDTL_Θ is satisfied by a labelled event structure E_Θ if it is satisfied for every event and every variable assignment; for the formal details we refer to [KF00]. A labelled event structure E_Θ is said to be a model of a module specification $ModSpec = (\Theta, \Phi)$, if it satisfies every formula $\varphi \in \Phi$ in E_Θ.

We now have gone through all the mathematical means required in order to describe TROLL and our extensions for modelling mobility-related issues in a formal way, and will turn to this in the next chapter.

Chapter 6

Mobile Troll

In this chapter, we are going to present some language constructs which can be used in the design of complex systems containing stationary and mobile parts. These constructs will be given as an extension to the Troll specification language. We will therefore start with an introduction of the elementary language concepts, and turn to our extensions in section 6.2. As we provide larger parts of the Troll grammar in section 6.1, we also repeat the syntax of the mobility-related extensions from section 4.2 in section 6.2. The syntax is always given using *Extended Backus Naur Form* (EBNF), i.e.

- alternatives (which are always exclusive) are separated by a vertical bar (|),
- optional details are included in square brackets ([,]),
- elements enclosed in between curly braces ({, }) have arbitrary cardinality, i.e. may occur none, one, or several times,
- terminal symbols are printed in **bold** font or are enclosed in single quotes (‘,’), and
- non-terminal symbols are written between angular brackets (<, >).

The complete grammar is given in appendix B.

6.1 Basic Language Concepts

The top level of a system specification is denoted by **object system**. It may contain either elementary language constructs referred to as *specItem* from Troll_3, or subsystems, which in this thesis are identified as *partSpec.*

Syntax

<*systemSpec*> ::= **object system** <*ident*>
(<*partSpec*> { ‘;’ <*partSpec*> } ‘;’ |
<*specItem*> { ‘;’ <*specItem*> } ‘;’)

end.

<*partSpec*> ::= <*statSpec*> | <*statInstanceDecl*> |
<*mobiSpec*> | <*mobiInstanceDecl*> |
<*dataTypeSpec*> | <*behaviorSpec*>

Subsystems will be discussed in section 6.2. The focus of this section is on the basic language concepts. These elementary concepts allow to specify complex data structures, object classes, declarations of instances of these classes, or the global behaviour between them.

Syntax

<*specItem*> ::= <*dataTypeSpec*> | <*objectClassSpec*> |
<*instanceDecl*> | <*behaviorSpec*>

TROLL provides a couple of predefined elementary data types. In addition, it offers the possibility to construct new complex data types from predefined ones. The latter consist of the data sorts and operators usually associated with them; the type constructors comprise constructors for sorts and generic operators. In the following, we assume that the names of the data types are reused as the data sorts. $ds(\psi)$ denotes the set of all data sorts of a subsystem specification ψ.

Object classes describe the potential objects of the system by means of structure, interface, and behaviour specifications. We refrain from discussing structuring mechanisms like specialisation or component relationships, as they are not relevant with regard to this thesis; details on it can be found in [Har97]. The local interface (local signature) consists of attribute and action declarations, the behaviour is given by operation definitions and integrity constraints.

Syntax

<*objectClassSpec*> ::= **object class** <*ident*> [<*specialization*>]
[**components** <*componentDecl*> {';' <*componentDecl*> } ';']
[<*signatureDecl*>]
[<*behaviorDef*>]
end

<*signatureDecl*> ::= [**attributes** <*attributeDecl*> {';' <*attributeDecl*> } ';']
[**actions** <*actionDecl*> {';' <*actionDecl*> } ';']

<*behaviorDef*> ::= [**behavior** <*operationDef*> {';' <*operationDef*> } ';']
[<*constraintDef*>**;**]

Object classes correspond to object sorts of class signatures introduced in definition 5.6. $oc(\psi)$ denotes the set of all class identifiers of a subsystem specification ψ.

Definition 6.1 (object sorts from classes) Let $oc(\psi)$ denote the set of all class identifiers of a subsystem specification ψ, then $S_O = oc(\psi)$ holds for the class signature $\Sigma_C(\psi) = (S_O, I, AT, AC)$ derivable from ψ. □

Note that this definition will later on have to be refined in order to take the object sorts into account which are induced by interface objects.

In order to illustrate this and the following definitions, we make use of the sample application discussed in section 4.4 where, for the time being, we restrict ourselves to the specification of the mobile subsystem `RetrievalUnit`.

Example 6.2 (object sorts from classes) For our sample application of mobile-agent-based information/document retrieval (IR), the specification of the mobile subsystem `RetrievalUnit` comprises the `RETRIEVER` object class, which means that

$$oc(\texttt{RetrievalUnit}) = \{\texttt{RETRIEVER}\}$$

holds at the formal level. □

6.1.1 Actions

The keyword **actions** marks the beginning of the section in an object class signature which contains the declarations of the actions provided by the instances of the respective class. Every action has an identifying name, which has to be unique throughout the object class specification. Additionally, it may have a list of parameters. Within this optional list, a distinction is made between input and output parameters, the latter being marked by a preceding exclamation mark "!". Every parameter is of a certain type and should have an identifying name.

Actions which should only be visible to an object itself are marked by the keyword **hidden**. Private functions not accessible from any other object can be declared this way. Two types of actions are explicitly marked: constructors (also referred to as "birth actions") are indicated by a preceding "∗", and destructors (also known as "death actions") by a leading "+".

Syntax

<*actionDecl*> ::= [∗|+] <*actionSignature*> [**hidden**]
<*actionSignature*> ::= <*ident*> [(<*parameter*>{',' <*parameter*> })]

<parameter> ::= [!]<field>
<field> ::= [<ident> ':'] <type>

The action alphabet of an object class can now be defined as follows:

Definition 6.3 (action declarations) An action declaration is given by a triple consisting of a sort-index identifier $id_{s^*}, s^* \subseteq ds(\psi)$ and the two sets $para_{in}(id_{s^*})$ and $para_{out}(id_{s^*})$, where s^* represents a (possibly empty) sequence $s_1, \ldots, s_n$ of sorts, $s^* \in ds(\psi)$ for $s_i \in ds(\psi), i \in \{1, \ldots, n\}$. $para_{in}(id_{s^*})$ is the set of the position indications for the input parameters, $para_{out}(id_{s^*})$ the one for the output parameters—i.e., those marked by a preceding "!".

$Act(c)$ denotes the set of action declarations for the object class specification c. This set is partitioned into three disjoint subsets:

- $Act_{birth}(c) \subseteq Act(c)$ denotes the set consisting of the birth actions (constructors) marked by "$*$",
- $Act_{death}(c) \subseteq Act(c)$ denotes the set consisting of the death actions (destructors) marked by "+", and
- $Act_{update}(c) \subseteq Act(c) - (Act_{birth}(c) \cup Act_{death}(c))$ denotes the set consisting of updating actions (modifiers).

□

Regarding subsystem specifications without interface objects, the corresponding class signature contains action symbols for all the action declarations of the subsystem's object classes.

Definition 6.4 (action symbols of a class signature) For the class signature $\Sigma_c(\psi) = (S_O, I, AT, AC)$ induced by a subsystem specification ψ holds:

$$\text{if } (a_{s^*}, para_{in}(a_{s^*}), para_{out}(a_{s^*})) \in Act(c) \text{ for } c \in oc(\psi), \text{ then } a : s^* \to c \in AC.$$

In the following, we may also write $a_{s^*} \in Act(c)$ as an abbreviation for $(a_{s^*}, para_{in}(a_{s^*}), para_{out}(a_{s^*})) \in Act(c)$ □

In order to be able to provide the local semantics of these actions, the prerequisites for their execution have to be defined. Furthermore, the relationship between input and output parameters has to be given.

Under the assumption that an object is alive if and only if it has been created by a birth action of the corresponding class sometime in the past, and a death action so far has not occurred for this object, these prerequisites can be formulated as follows: a birth action may only occur if the object is not alive, while a death action may only take place for those who are. If an update action has just occurred, then the object must be alive.

The output parameters should be functionally dependent on the input parameters and the current state in order to guarantee a deterministic behaviour of a system.

A formal definition of these rules is given in [Har97]. There, first a special predicate is introduced in order to define the liveness of objects, which is then used to formalise frame rules for the occurrence of actions. The relationship between input and output parameters is also characterised by such a frame rule. These rules impose axioms on the instances of a system, restricting the possible runs of its objects. They form elements of the set Φ of axioms for a subsystem specification ψ.

We refrain from repeating these frame rules and instead illustrated definition 6.4 and also provide some axioms resulting for the object class `RETRIEVER` from the issues discussed so far:

Example 6.5 (action alphabet and axioms for `RETRIEVER`) For the object class `RETRIEVER`, the following sets of actions can be derived from the example specification:

$$\begin{aligned}
Act_{birth}(\texttt{RETRIEVER}) &= \{(\texttt{becomeRetriever}_\epsilon,\\
&\quad para_{in}(\texttt{becomeRetriever}_\epsilon),\\
&\quad para_{out}(\texttt{becomeRetriever}_\epsilon))\}\\
Act_{update}(\texttt{RETRIEVER}) &= \{(\texttt{launchRetrieval}_{\texttt{set(string)},\texttt{list(string)},\texttt{set(litInfo)}},\\
&\quad para_{in}(\texttt{launchRetrieval}_{\texttt{set(string)},\texttt{list(string)},\texttt{set(litInfo)}}),\\
&\quad para_{out}(\texttt{launchRetrieval}_{\texttt{set(string)},\texttt{list(string)},\texttt{set(litInfo)}})),\\
&\quad \ldots,\\
&\quad (\texttt{continueRetrieval}_\epsilon,\\
&\quad para_{in}(\texttt{continueRetrieval}_\epsilon),\\
&\quad para_{out}(\texttt{continueRetrieval}_\epsilon))\}\\
Act_{death}(\texttt{RETRIEVER}) &= \{(\texttt{cease}_\epsilon,\\
&\quad para_{in}(\texttt{cease}_\epsilon),\\
&\quad para_{out}(\texttt{cease}_\epsilon))\}
\end{aligned}$$

with $para_{in}(\texttt{becomeRetriever}_\epsilon) = para_{out}(\texttt{becomeRetriever}_\epsilon) = para_{in}(\texttt{cease}_\epsilon) = para_{out}(\texttt{cease}_\epsilon) = \{\}$ and

$$\begin{aligned}
para_{in}(\texttt{launchRetrieval}_{\texttt{set(string)},\texttt{list(string)},\texttt{set(litInfo)}}) &= \{1,2\},\\
para_{out}(\texttt{launchRetrieval}_{\texttt{set(string)},\texttt{list(string)},\texttt{set(litInfo)}}) &= \{3\},\\
&\vdots\\
para_{in}(\texttt{continueRetrieval}_\epsilon) &= \{\}, \text{ and}\\
para_{out}(\texttt{continueRetrieval}_\epsilon) &= \{\}
\end{aligned}$$

Furthermore, the following axioms have to hold for all `RETRIEVER` objects r:

$$r.(r.alive \Leftrightarrow P\,(\odot r.\texttt{becomeRetriever}()) \wedge \neg \odot\, r.\texttt{cease}())$$

$r.(\triangleright r.\texttt{becomeRetriever}() \Rightarrow \neg\, r.alive)$

$r.(\triangleright r.\texttt{cease}() \Rightarrow r.alive)$

$r.(\odot r.\texttt{launchRetrieval}(k,t,a) \Rightarrow r.alive)$

$r.(\odot r.\texttt{continueRetrieval}() \Rightarrow r.alive)$

where $r \in X_{\text{RETRIEVER}}$, $k \in X_{\text{set(string)}}$, $t \in X_{\text{list(string)}}$ and $a \in X_{\text{set(litInfo)}}$.

For the objects of the RETRIEVER class, these axioms formalise the above-mentioned prerequisites for the execution of the actions. □

The repeated occurrence of object identities in formulae like those in the previous example (there: r) is due to the identifier being used as a prefix of action terms as well as of DTL formulae (cf. definition 5.11 (4.) and 5.33, respectively). This use is redundant here, but allows a straight forward transformation of DTL formulae into those of MDTL, cf. definition 5.50.

6.1.2 Attributes

Attributes are used to model the observable properties of objects. Together with the actions discussed previously, they form the local signature of an object class. In TROLL, the section containing the declarations of attributes is introduced by the keyword **attributes**.

The declaration of an attribute consists of an identifier, which has to be unique throughout the object class, and the type for it, separated from the former by a colon. Like actions, attributes can be marked as **hidden**, rendering them visible only to the object containing them. Attributes marked as **constant** can never change their value, while **optional** attributes are not required to have a value from the beginning of the object's life time. For attributes marked as **derived** also a rule has to be provided on how to determine its value from those of other attributes. Finally, values to be assigned to attributes at the time of an object's creation can be given following the keyword **initialized**. However, these values have to be constants.

Syntax

<*attributeDecl*>	::=	<*variable*> [<*attributDesc*> {',' <*attributeDesc*> }]
<*attributeDesc*>	::=	**hidden** | **constant** | **optional** |
		derived <*dataterm*> | **initialized** <*constTerm*>
<*variable*>	::=	<*ident*> ':' <*type*>

In analogy to the action alphabet we can now define the attribute alphabet of an object class, already making use of data terms which will be introduced in the following subsection.

Definition 6.6 (attribute declaration) An attribute declaration is given by a sort-indexed identifier $id_s, s \in ds(\psi)$, and, in case of a derived attribute, an additional data term t of sort s.

$Attr(c)$ denotes the set of all attribute declarations for a given object class c. For this set, the following subsets are defined:

1. the set of constant attributes: $Attr_{const}(c)$,
2. the set of optional attributes: $Attr_{opt}(c)$,
3. the set of derived attributes: $Attr_{der}(c)$,
4. the set of initialised attributes: $Attr_{init}(c)$, and
5. the set of hidden attributes: $Attr_{hidden}(c)$.

□

Furthermore, constraints can be made in order to prevent unreasonable combinations of attribute descriptors, like $(Attr_{const}(c) \cup Attr_{init}(c)) \cap Attr_{opt}(c) = \{\}$, which states that constant and initialised attributes cannot be optional at the same time. However, as it is not the topic of this thesis, we refrain from investigating this aspect any further.

The attribute alphabet just defined provides the basis for formulating attribute terms:

Definition 6.7 (attribute terms) The set of attribute terms for an object class c is defined by:

if $a_s \in Attr(c)$, then a is an attribute term of sort s and $free(a) = \{a\}$.

□

Every attribute term of sort s is at the same time also a data term of sort s, cf. definition 6.10 (11.) below.

As already mentioned in relation to definition 5.6 and example 5.7, like earlier works we continue to introduce additional action symbols which allow us to describe the access to attributes in terms of read and write operations (cf. e.g. [Har97, Eck01]).

Definition 6.8 (attributes as action symbols of a class signature) Given a class signature $\Sigma_C(\psi) = (S_O, I, AT, AC)$ induced by a subsystem specification ψ,

- if $a_s \in Attr(c) - Attr_{der}(c)$ for $c \in oc(\psi)$, then $\{a_{write} : s \to c, a_{read} : s \to c\} \subset AC$ and
- if $a_s \in Attr_{der}(c)$ for $c \in oc(\psi)$, then $a_{read} : s \to c \in AC$

hold. □

Just like for actions, frame rules can be formulated for attributes in order to formally define the intended semantics. For details, we refer to [Har97] and [Eck01], as this aspect is not in the main focus of this thesis. Instead, we merely give an example of the action symbols derived from attributes of the example specification:

Example 6.9 (attributes as action symbols) According to definition 6.8, for the object class `RETRIEVER` the following action symbols are derived corresponding to its attributes:

$$\begin{aligned}
&\texttt{homeLoc}_{read} : \mathsf{string} \rightarrow \texttt{RETRIEVER},\\
&\texttt{homeLoc}_{write} : \mathsf{string} \rightarrow \texttt{RETRIEVER},\\
&\quad\vdots\\
&\texttt{litrInfo}_{read} : \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{RETRIEVER},\\
&\texttt{litrInfo}_{write} : \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{RETRIEVER}.
\end{aligned}$$

□

6.1.3 Terms, Declarations, Propositions

In this section data terms, range declarations, and propositions will be defined syntactically as well as semantically. As fundamental constructs of a specification language data terms describe simple and complex values. Examples of simple data terms are constants and variables, but also the attribute terms already introduced in definition 6.7. In TROLL, complex data terms can be constructed from simpler data terms using prefix and infix operators as well as by providing conditional and query terms. Furthermore, it is possible to create terms matching user-defined data types by means of data term constructors, or to refer to subterms of complex terms.

Syntax

<*dataTerm*> ::= <*ident*> | <*constTerm*> | (<*dataTerm*>) | ... |
sublist(<*dataTerm*> ',' <*dataTerm*> '..' <*dataTerm*>) |
<*prefixOp1*>(<*dataTerm*>) |
<*prefixOp2*>(<*dataTerm*> ',' <*dataTerm*>) |
<*dataTerm*> <*infixOp*> <*dataTerm*> |
<*constructor*> | <*condTerm*> | <*selectTerm*>

<*condTerm*> ::= '[' <*pFormula*> '?' <*dataTerm*> ':' <*dataTerm*> ']'

<*selectTerm*> ::= **select** <*dataTerm*> **from** <*rangeDecl*>
[**where** <*pFormula*>]

As on the one hand conditional terms and query/select terms may contain propositions, and the latter may also comprise declarations, but on the other hand propositions as well as declarations may be constructed on top of data terms, we obtain a recursive structure instead of a hierarchical one. The following definition of data terms therefore already refers to the one of range declarations and propositions provided thereafter. Both definitions make use the sets $free$ and $decl$ of variables occurring free or being introduced by means of a declaration, respectively.

Definition 6.10 (data terms for a subsystem specification) The set of data terms for a subsystem specification ψ and a set of variables X is defined by:

1. If t is a constant of sort s, then t is a data term of sort s and $free(t) = \{\}$.
2. If v is a variable of sort s, then v is a data term of sort s and $free(v) = \{v\}$.
3. If t is a data term of sort s, then (t) is a data term of sort s and $free((t)) = free(t)$.
4. If t, t_1, t_2 are data terms of sorts s, s_1, s_2, respectively, then $\mathsf{sublist}(t, t_1..t_2)$ is a data term of sort s and $free(\mathsf{sublist}(t, t_1..t_2)) = free(t) \cup free(t_1) \cup free(t_2)$.
5. If t_1 is a data term of sort s_1 and $\circ$ is a unary prefix operator $\circ : s_1 \to s$, then $\circ(t_1)$ is a data term of sort s and $free(\circ(t_1)) = free(t_1)$.
6. If t_1, t_2 are data terms of sorts s_1, s_2, respectively, and if $\circ$ is a binary prefix operator $\circ : s_1 \times s_2 \to s$, then $\circ(t_1, t_2)$ is a data term of sort s and $free(\circ(t_1, t_2)) = free(t_1) \cup free(t_2)$.
7. If t_1, t_2 are data terms of sorts s_1, s_2, respectively, and if $\circ$ is a binary infix operator $\circ : s_1 \times s_2 \to s$, then $(t_1 \circ t_2)$ is a data term of sort s and $free(t_1 \circ t_2) = free(t_1) \cup free(t_2)$.
8. If $t_1, \ldots, t_n$ are data terms of sorts $s_1, \ldots, s_n$, respectively, and if $\circ$ is a constructor of sort s, then $\circ(t_1, \ldots, t_n)$ is a data term of sort s and $free(\circ(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} free(t_i)$.
9. If t_1 and t_2 are both data terms of sorts s, and if c is a proposition, then $c?t_1 : t_2$ is a data term of sort s and $free(c?t_1 : t_2) = free(c) \cup free(t_1) \cup free(t_2)$.
10. If t is a data term of sort s, c is a proposition, and d is a range declaration, then $\mathsf{select}\ t\ \mathsf{from}\ d\ \mathsf{where}\ c$ is a data term of sort $\mathsf{bag}(s)$ and $free(\mathsf{select}\ t\ \mathsf{from}\ d\ \mathsf{where}\ c) = free(t) \cup free(d) \cup free(c) \setminus decl(d)$.
11. If t is an attribute term of sort s, then t is a data term of sort s.

□

Range declarations are required in formulating query terms. They obey the following syntax:

Syntax

<rangeDecl> ::= *<ident>* **in** *<dataTerm>*{ ',' *<ident>* **in** *<dataTerm>* }

With such a statement an implicitly declared variable is bound to a set of values, the set being determined by a suitable data term. The second item in the following definition guarantees that variables are declared only once.

Definition 6.11 (range declarations) The set of range declarations for a subsystem specification ψ and a set X of variables is defined by:

1. If v is a variable of sort s and t is a set-valued data term of sort $\mathsf{set}(s)$, then $v : t$ is a range declaration and $free(v : t) = free(t)$ and $decl(v : t) = \{v\}$.
2. If d' is a range declaration, v a variable of sort $s, v \notin decl(d')$, and if t is a set-valued data term of sort $\mathsf{set}(s)$, then $d', v : t$ is a range declaration and $free(d', v : t) = free(d') \cup free(t)$ and $decl(d', v : t) = decl(d') \cup \{v\}$.

□

As opposed to formulae of the object logic, TROLL formulae are referred to as "propositions." The main difference is that propositions can be used in order to specify state-dependent statements for objects directly, which is not possible for formulae of the logic. This is mostly due to the data terms over which the formulae are constructed. In particular, in TROLL attribute terms can be used (cf. definitions 6.7 and 6.10), which allow direct access to the state of an object.

However, except for the structure of data terms and the fact that quantification is only possible over finite ranges, the structure of TROLL propositions corresponds to the one known from predicate logics:

Syntax

<boolOp> ::= **or** | **and** | **implies** | **xor**

<pFormula> ::= **not** *<pFormula>* | *<pFormula>* *<boolOp>* *<pFormula>* |
<dataTerm> | **all** *<rangeDecl>*(*<pFormula>*) |
(*<pFormula>*) | **any** *<rangeDecl>*(*<pFormula>*)

Formally, the set of propositions and the variables occurring free in them is defined as follows:

Definition 6.12 (propositions) The set of TROLL propositions for a subsystem specification ψ and a set X of variables is defined by:

1. If t is a TROLL data term of sort bool, then t is a TROLL proposition.
2. If c is a TROLL proposition, then not c and (c) are TROLL propositions and $free(\mathsf{not}\ c) = free(c)$ and $free((c)) = free(c)$.
3. If c_1, c_2 are TROLL propositions and $\circ$ is a boolean operator, then $c_1 \circ c_2$ is a TROLL proposition and $free(c_1 \circ c_2) = free(c_1) \cup free(c_2)$.
4. If d is a range declaration, c is a TROLL proposition, and if $\circ \in \{\mathsf{any}, \mathsf{all}\}$, then $\circ\, d\, (c)$ is a TROLL proposition and $free(\circ\, d\, (c)) = (free(c) \cup free(d)) - decl(d)$.

□

The definition of the formal semantics for the TROLL terms, range declarations, and propositions introduced above can be given by providing transformation rules mapping them onto terms, declarations and formulae of the object logic. Due to the already mentioned differences between TROLL data terms and those of the module logic, a one-to-one incorporation of the data terms into the object logic is not possible.

The following definition 6.13 is concerned with mapping propositions into the object logic. The image of non-atomic propositions without range declarations consists of the images of the atoms, related by the logical connectives from the object logic corresponding to those of the TROLL formula. In mapping propositions given by a boolean data term, the images of the constituent terms have to be considered, and for propositions containing quantifiers the mapping should preserve the range declarations.

The identity term o passed along in the translation allows to identify the object for which the proposition, term, or range declaration was given, and the resulting formula can therefore be assigned to it.

Definition 6.13 (mapping propositions) For a proposition φ, an identity term $o \in T_{\Sigma_{Id}}(X)$, and a set X of variables, the mapping $\mathcal{P}_{o,X}(\varphi)$ to a formula of the object logic DTL_Σ is defined as follows:

1. If t is a term of sort s, then
   $\mathcal{P}_{o,X}(t) \equiv \exists\, v\, \mathcal{T}_{o,X}(v = t) \wedge v = \mathsf{true}$.
2. If φ and ψ are propositions, then
   (a) $\mathcal{P}_{o,X}(\varphi\ \mathsf{or}\ \psi) \equiv \mathcal{P}_{o,X}(\varphi) \vee \mathcal{P}_{o,X}(\psi)$,
   (b) $\mathcal{P}_{o,X}(\varphi\ \mathsf{and}\ \psi) \equiv \mathcal{P}_{o,X}(\varphi) \wedge \mathcal{P}_{o,X}(\psi)$,
   (c) $\mathcal{P}_{o,X}(\varphi\ \mathsf{implies}\ \psi) \equiv \mathcal{P}_{o,X}(\varphi) \Rightarrow \mathcal{P}_{o,X}(\psi)$,
   (d) $\mathcal{P}_{o,X}(\varphi\ \mathsf{xor}\ \psi) \equiv (\neg\, \mathcal{P}_{o,X}(\varphi) \wedge \mathcal{P}_{o,X}(\psi)) \vee (\mathcal{P}_{o,X}(\varphi) \wedge \neg\, \mathcal{P}_{o,X}(\psi))$,
   (e) $\mathcal{P}_{o,X}(\mathsf{not}\ \varphi) \equiv \neg\, \mathcal{P}_{o,X}(\varphi)$, and
   (f) $\mathcal{P}_{o,X}((\varphi)) \equiv (\mathcal{P}_{o,X}(\varphi))$.

3. If t_i are data terms of sorts $\mathsf{set}(s_i)$ for $i \in \{1, \ldots, n\}$, if φ is a proposition, and if $v_1 \,\mathsf{in}\, t_1, \ldots, v_n \,\mathsf{in}\, t_n$ is a range declaration, then

   (a) $\mathcal{P}_{o,X}(\mathsf{any}\, v_1 \,\mathsf{in}\, t_1, \ldots, v_n \,\mathsf{in}\, t_n\, (\varphi)) \equiv$
   $\exists\, v_1, v_2, \ldots, v_n$
   $(\mathcal{D}_{o,X}(v_1 \,\mathsf{in}\, t_1) \wedge \mathcal{D}_{o,X\cup\{v_1\}}(v_2 \,\mathsf{in}\, t_2) \wedge \ldots \wedge \mathcal{D}_{o,X\cup\{v_1,\ldots,v_{n-1}\}}(v_n \,\mathsf{in}\, t_n)$
   $\wedge(\mathcal{P}_{o,X\cup\{v_1,\ldots,v_n\}}(\varphi)))$ and

   (b) $\mathcal{P}_{o,X}(\mathsf{all}\, v_1 \,\mathsf{in}\, t_1, \ldots, v_n \,\mathsf{in}\, t_n\, (\varphi)) \equiv \mathcal{P}_{o,X}(\mathsf{any}\, v_1 \,\mathsf{in}\, t_1, \ldots, v_n \,\mathsf{in}\, t_n\, (\mathsf{not}\, \varphi))$

□

A range declaration is mapped to a formula of the object logic by declaring a set-valued, existentially quantified variable and equating this variable with the finite range determined by the data term. It is required that the additional variable is an element of the set mentioned:

Definition 6.14 (mapping range declarations) For a range declaration $v \,\mathsf{in}\, t$, an identity term $o \in T_{\Sigma_{Id}}(X)$, and a set X of variables, the mapping $\mathcal{D}_{o,X}(v \,\mathsf{in}\, t)$ to a formula of the object logic $\textsc{Dtl}_\Sigma$ is defined by:

$$\mathcal{D}_{o,X}(v \,\mathsf{in}\, t) \equiv \exists\, m\, (\mathcal{T}_{o,X}(m = t) \wedge v \in m)$$

where t is a data term, m is a variable of sort $\mathsf{set}(s)$, and v is a variable of sort s. □

Constants and variables as well as prefix and infix operators applied to them can be incorporated into the logic on a one-to-one basis (cf. definition 5.3); the same holds for sort constructors and their associated generic operators. As neither conditional terms nor query terms nor attribute terms have a corresponding counterpart in the object logic, translating them is only possible using the following trick: conditional terms and query terms are transformed into boolean terms by equating them with variables. This way, a mapping to formulae containing only data terms which comply to definition 5.3 can be used instead of one to data terms of the object logic. Attribute terms are mapped to read actions for the corresponding attributes. The following definition provides such a mapping for boolean terms $v = t$ to a formula of the object logic.

Definition 6.15 (mapping data terms) For a proposition $v = t$, an identity term $o \in T_{\Sigma_{Id}}(X)$, and a set X of variables, the mapping $\mathcal{T}_{o,X}(v = t)$ to a formula of the object logic $\textsc{Dtl}_\Sigma$ is defined by:

1. $\mathcal{T}_{o,X}(v = t) \equiv v = t$, if t is a constant data term.

2. (a) $\mathcal{T}_{o,X}(v = x) \equiv v = x$, if $x \in X$.

   (b) $\mathcal{T}_{o,X}(v = x) \equiv \triangleright o.x_{read}(v)$, if x is an attribute term and $x \notin X$.

3. $\mathcal{T}_{o,X}(v = \mathsf{sublist}(t_1, t_2..t_3)) \equiv$
$\exists\, v_1, v_2, v_3$
$(\mathcal{T}_{o,X}(v_1 = t_1) \wedge \mathcal{T}_{o,X}(v_2 = t_2) \wedge \mathcal{T}_{o,X}(v_3 = t_3) \wedge (v = \mathsf{sublist}(v_1, t_2..t_3)))$
with $v_i \in X_{s_i}$, if t_i is a data term of sort s_i for $i = 1, 2, 3$.

4. (a) $\mathcal{T}_{o,X}(v = \circ(t_1)) \equiv \exists\, v_1\, (\mathcal{T}_{o,X}(v_1 = t_1) \wedge (v = \circ(v_1)))$ with $v_1 \in X_{s_1}$, if $\circ$ is a unary prefix operator and t_1 is a data term of sort s_1.

   (b) $\mathcal{T}_{o,X}(v = \circ(t_1, t_2)) \equiv$
   $\exists\, v_1, v_2\, (\mathcal{T}_{o,X}(v_1 = t_1) \wedge \mathcal{T}_{o,X}(v_2 = t_2) \wedge (v = \circ(v_1, v_2)))$
   with $v_1 \in X_{s_1}, v_2 \in X_{s_2}$, if $\circ$ is a binary prefix operator and t_1 and t_2 are data terms of sort s_1 and s_2, respectively.

   (c) $\mathcal{T}_{o,X}(v = t_1 \circ t_2) \equiv$
   $\exists\, v_1, v_2\, (\mathcal{T}_{o,X}(v_1 = t_1) \wedge \mathcal{T}_{o,X}(v_2 = t_2) \wedge (v = v_1 \circ v_2))$
   with $v_1 \in X_{s_1}, v_2 \in X_{s_2}$, if $\circ$ is an infix operator and t_1 and t_2 are data terms of sort s_1 and s_2, respectively.

   (d) $\mathcal{T}_{o,X}(v = \circ(t_1, \ldots, t_n)) \equiv$
   $\exists\, v_1, \ldots, v_n\, (\mathcal{T}_{o,X}(v_1 = t_1) \wedge \ldots \wedge \mathcal{T}_{o,X}(v_n = t_n) \wedge (v = \circ(v_1, \ldots, v_n)))$
   with $v_i \in X_{s_i}$, if $\circ$ is a constructor and t_i is a data term of sort s_i for $i = 1, \ldots, n$.

5. $\mathcal{T}_{o,X}(v = \varphi?t_1 : t_2) \equiv (\mathcal{P}_{o,X}(\varphi) \Rightarrow \mathcal{T}_{o,X}(v = t_1)) \wedge (\neg\mathcal{P}_{o,X}(\varphi) \Rightarrow \mathcal{T}_{o,X}(v = t_2))$
with $v \in X_s$, if φ is a proposition and t_1 and t_2 are data terms of sorts s_1 and s_2, respectively.

6. $\mathcal{T}_{o,X}(v = \mathsf{select}\ t\ \mathsf{from}\ v_1\ \mathsf{in}\ t_1, \ldots, v_n\ \mathsf{in}\ t_n\ \mathsf{where}\ \varphi) \equiv$
$\exists\, m\, (\forall\, v_1, \ldots, v_n (\mathcal{D}_{o,X}(v_1\ \mathsf{in}\ t_1) \wedge \ldots \wedge \mathcal{D}_{o,X\cup\{v_1,\ldots,v_{n-1}\}}(v_n\ \mathsf{in}\ t_n) \Rightarrow$
$(\mathcal{P}_{o,X\cup\{v_1,\ldots,v_n\}}(\varphi) \Rightarrow$
$(\exists\, e\, \mathcal{T}_{o,X\cup\{v_1,\ldots,v_n\}}(e = m) \wedge m(v_1, \ldots, v_n) = e)) \wedge$
$(\neg\, \mathcal{P}_{o,X\cup\{v_1,\ldots,v_n\}}(\varphi) \Rightarrow$
$(\neg\, \mathsf{def}(m(v_1, \ldots, v_n))))) \wedge$
$\forall\, v'_1, \ldots, v'_n (\neg\, \mathcal{D}_{o,X}(v'_1\ \mathsf{in}\ t_1) \vee \ldots \vee \neg\, \mathcal{D}_{o,X\cup\{v'_1,\ldots,v'_{n-1}\}}(v'_n\ \mathsf{in}\ t_n) \Rightarrow$
$(\neg\, \mathsf{def}(m(v_1, \ldots, v_n)))) \wedge$
$v = \mathsf{rng}(m))$,
where the v_i are variables of sorts s_i, m is a variable of sort $\mathsf{map}(\mathsf{record}(s_1, \ldots, s_n))$, e is a variable and t a term of sort s, and the t_i are data terms of sorts $\mathsf{set}(s_i)$.

   The operator rng returns the range/counterdomain of a mapping and the predicate def states whether the mapping is defined for the values given or not.

□

Examples 6.22 and 6.53 illustrate the use of these definitions by providing a translation of an operation definition into a formula of the object logic. Further examples can be found in [Har97] and [Eck01].

6.1.4 Operations

While the signature of the specification of an object class describes the objects' interface, the behaviour part details the objects' reaction to calls to the actions declared in its interface. Additionally, restrictions on the permissible behaviour can be given by specifying initialisation conditions and invariants, cf. e.g. [Har97]. However, as the latter are not relevant to this thesis, we will confine ourselves to the behaviour specification.

An operation definition is structured as follows: it starts with an action term indicating the action for which the behaviour specification is given. Next, an optional precondition can be stated, syntactically introduced by the keyword **onlyIf**, with a TROLL proposition following it. The action may only take place if the proposition evaluates to true. Prior to the actual specification of the effects of the action as the final part in the form of action rules, local variables can be declared.

Syntax

<*operationDef*>	::=	<*actionTerm*> [**onlyIf** <*pFormula*>] [<*variableDecl*>] [**do** <*actionRule*> **od**]
<*actionTerm*>	::=	<*ident*>[(<*ident*>{ ',' <*ident*> })]
<*variableDecl*>	::=	**var** <*variable*> { ',' <*variable*>}

In the complete grammar given in appendix B, the structure of action terms is somewhat more complicated due to the fact that it also considers component relationships, which we can neglect with regard to this thesis.

Let us now discuss the individual parts of an operation definition, which are all optional except for the action term.

The formal parameters of the action declaration to which the action term refers to are replaced within the latter by pairwise disjoint identifiers for variables. These identifiers denote an implicit variable declaration, local to the operation definition. The sorts for the variables are derived from the corresponding parameters; input and output variables are distinguished.

Definition 6.16 (action terms of an object class) The set of action terms of an object class c is defined by:

> if $a_{s_1,\ldots,s_n} \in Act(c)$ and $v_i \in X_{s_i}, i \in \{1,\ldots,n\}$, with $v_i \neq v_j$ for $i \neq j$, then $a(v_1,\ldots,v_n)$ is an action term and $var_{in}(a(v_1,\ldots,v_n)) = \{v_j \mid j \in para_{in}(a_{s^*})\}$ is the set of its input variables and $var_{out}(a(v_1,\ldots,v_n)) = \{v_j \mid j \in para_{out}(a_{s^*})\}$ is the set of its output variables.

□

The structure of an operation definition can now be described formally:

Definition 6.17 (operation definition) Let ψ denote a subsystem specification and $c \in oc(\psi)$, then $(\alpha, \varphi, var_{local}, ax)$ is an operation definition provided

1. α is an action term of c,
2. φ is a proposition with $free(\varphi) \subseteq var_{in}(\alpha) \cup Attr(c)$,
3. var_{local} is set of sort-indexed identifiers id_s with $var_{local} \cap (var_{in}(\alpha) \cup var_{out}(\alpha)) = \{\}$, and
4. ax is an action rule such that
   $free(ax) \subseteq Attr(c) \cup var_{in}(\alpha) \cup var_{out}(\alpha) \cup var_{local}$,
   $evaluated(ax) \subseteq (Attr(c) - Attr_{der}(c)) \cup var_{in}(\alpha) \cup var_{out}(\alpha) \cup var_{local}$

holds. □

As it must be possible to evaluate the precondition prior to the possible execution of the action, no references to the next state may be made from it: definition 6.17 ensures this by allowing only input variables and attributes to be used within the proposition φ, but no output variables or locally defined ones. The identifiers of the variables declared within the operation definition have to be different from those implicitly declared.

The above definition states only coarsely which variables and attributes of an action rule may be effected by the occurrence of an action. As TROLL does not allow the specification of variables which are both input and output variables at the same time, these effected items are all the variables defined locally, all the output variables, and all attributes except the derived ones, as the latter cannot be assigned any values. The set $evaluated(ax)$ of variables and attributes potentially modified by the action rule ax is restricted further by the following definitions of so-called action rule primitives and call terms, but prior to those definitions the formal semantics of a precondition should be given:

Definition 6.18 (semantics of preconditions) Given an operation definition $(\alpha, \varphi, var_{local}, ax)$ of an object class specification $c \in oc(\psi)$, Φ_ψ contains the following formula of the object logic:

$$o.(\triangleright\, o.\alpha \Rightarrow \mathcal{P}_{o,var_{in}(\alpha)}(\varphi)),$$

where $o \in X_{c^i}$. □

In this definition, c^i denotes the sorts of object identities pertaining to the class c, and X_{c^i} the set of variables of this sort.

The condition formalised in this definition expresses that an action may only occur if the formula $\mathcal{P}_{o,var_{in}(\alpha)}(\varphi)$ of the object logic corresponding to the specified

precondition φ is met. Note, however, that the frame rules mentioned with regard to definition 6.4 also have to be true.

An action rule details which state an object has to be in immediately after the action has occurred. Hence, an action rule can be considered a postcondition. In the simple case, it consists of an assignment of a value to a variable or attribute (*valuation*), or of a call to another action (*call term*). However, it may also describe the multiple (*repetitive rule*) or conditional (*conditional rule*) execution of an action rule. Finally, it may also consist of a list of action rules.

Neither the repetitive execution of action rules nor the composition of several rules to a list imposes any ordering on the execution sequence. Conceptionally, a (complex) action rule is always performed within a single step; it merely states which prerequisites have to hold prior to execution and which conditions it will establish.

Syntax

<*actionRule*> ::= <*valuation*> | <*callTerm*> |
<*repetitiveRule*> | <*conditionalRule*> |
<*actionRule*> { ',' <*actionRule*> }

<*valuation*> ::= <*assignTerm*> ':=' <*dataTerm*>

<*callTerm*> ::= <*ident*> [(<*dataTerm*> { ',' <*dataTerm*> })]

<*repetitiveRule*> ::= **forEach** <*rangeDecl*> **do** <*actionRule*> **od**

<*conditionalRule*> ::= **if** <*pFormula*> **then** <*actionRule*>
[**else** <*actionRule*>] **fi**

In this thesis, an assignment term (*assign term*) is used only in conjunction with attributes and variables. However, it can also be used to express the assignment of values to an attribute of a component object, to an individual field of a record structure, or to an item of a list.

Action calls are special action terms—referred to as *call terms* in the following—which use arbitrary data terms instead of input variables. Their output variables are replaced by variables (of matching sorts) which are defined in the calling operation, either as local or as output variables. Just like for action terms, we confine our discussion to simple call terms, i.e. those not including component relations.

Definition 6.19 (call terms) The set of call terms of an object class c is defined by:

> If $a_{s_1,\ldots,s_n} \in Act(c)$, if t_i are data terms sorts s_i for $i \in \{1,\ldots,n\}$, if $t_i = v_i \in X_{s_i}$ for $i \in para_{out}(a_{s_1,\ldots,s_n})$, and if furthermore $v_i \in var_{loc} \cup var_{out}$, then $a(t_1,\ldots,t_n)$ is a call term and

$free(a(t_1, \ldots, t_n)) = \bigcup_{i \in \{1,\ldots,n\}} free(t_i)$, $evaluated(a(t_1, \ldots, t_n)) = \{v_i \mid i \in para_{out}(a_{s_1,\ldots,s_n})\}$.

□

With this definition, we are now able to define action rules formally:

Definition 6.20 (action rules) Let α denote an action term of an object class specification c and var_{local} a set of local variables. The set of action rules for α and var_{local} is then defined by:

1. If v is a variable and t a data term of sort s, then the assignment $v := t$ is an action rule with $evaluated(v := t) = \{v\}$ and $free(v := t) = free(t)$.

2. If β is a call term, then the action call β is an action rule.

3. If ax' is an action rule and d is a declaration for which $decl(d) \cap evaluated(ax') = \{\}$ holds, then **forEach** d **do** ax' **od** is an action rule with $evaluated($**forEach** d **do** ax' **od**$) = evaluated(ax')$ and $free($**forEach** d **do** ax' **od**$) = free(d) \cup (free(ax') - decl(d))$.

4. If ax_1 and ax_2 are action rules and p is a proposition, then **if** p **then** ax_1 **else** ax_2 **fi** is a conditional action rule with $evaluated($**if** p **then** ax_1 **else** ax_2 **fi**$) = evaluated(ax_1) \cup evaluated(ax_2)$ and $free($**if** p **then** ax_1 **else** ax_2 **fi**$) = free(p) \cup free(ax_1) \cup free(ax_2)$.

5. If ax_i are action rules for $i \in \{1, \ldots, n\}$, then $ax_1, \ldots, ax_n$ is an action rule with $free(ax_1, \ldots, ax_n) = \bigcup_{i \in \{1,\ldots,n\}} free(ax_i)$ and $evaluated(ax_1, \ldots, ax_n) = \bigcup_{i \in \{1,\ldots,n\}} evaluated(ax_i)$.

□

Together with definition 6.17 (4.), the last item here guarantees that assignments to input parameters are not possible, as only local variables, output variables, and non-derived attributes may appear on the left hand side of such a statement.

An action call according to the second clause represents the joint execution of called and calling action. The calling action provides the values for the input parameters and suitable variables for the output parameters. The action being called then determines the values for the output parameters by using the input parameters and attribute values. In the joint execution these values are then bound to the variables of the calling action.

The third item ensures that in a term of the form **forEach** d **do** ax' **od** no values may be assigned to any variable possibly declared. All in all, the set of attributes and variables which potentially may get modified by complex action rules therefore consists of the corresponding sets of its constituent parts.

In order to be able to provide the formal semantics of action rules, the data terms and proposition on top of which they are built have to be mapped to

formulae of the object logic. In this mapping care has to be taken that all terms which either occur as input parameters or appear on the right hand side of assignments are evaluated in the state directly preceding the action execution. The following definition is in accordance with this demand.

Definition 6.21 (semantics of action rules) Let $(\alpha, \varphi, var_{local}, ax)$ with $var_{local} = \{v_1, \ldots, v_n\}$ be an operation definition of an object class c, then Θ_ψ contains the following axiom:

$$o.(\odot\, o.\alpha \Rightarrow \exists\, v_1, \ldots, v_n\, \mathcal{A}_{o,var_{in}(\alpha)\cup var_{out}(\alpha)\cup var_{local}}(ax)),$$

with $v_j \in X_{s_j}$ for $j \in \{1, \ldots, n\}$ and $o \in X_{c^i}$, where the mapping $\mathcal{A}_{o,X}(ax)$ with an identity term $o \in T_{\Sigma_{Id}}(X)$ and a set of variables X is defined inductively as follows:

1. (a) $\mathcal{A}_{o,X}(v := t) \equiv \exists\, v'\, \mathsf{Y}(\mathcal{T}_{o,X}(v' = t)) \wedge v = v'$, if $v \in X$ and
   (b) $\mathcal{A}_{o,X}(v := t) \equiv \exists\, v'\, \mathsf{Y}(\mathcal{T}_{o,X}(v' = t)) \wedge \odot\, o.v_{write}(v'))$ otherwise

2. $\mathcal{A}_{o,X}(b(t_1, \ldots, t_m)) \equiv$
   $\exists\, v_1, \ldots, v_m\, \mathsf{Y}(\mathcal{T}_{o,X}(v_1 = t_1)) \wedge \ldots \wedge \mathsf{Y}(\mathcal{T}_{o,X}(v_m = t_m)) \wedge \odot\, o.b(v_1, \ldots, v_m)$,
   if $b_{s^*} \in Act(c)$

3. $\mathcal{A}_{o,X}(\mathsf{forEach}\ v_1\ \mathsf{in}\ t_1,\ \ldots,\ v_m\ \mathsf{in}\ t_m\ \mathsf{do}\ ax'\ \mathsf{od}) \equiv$
   $\forall\ v_1, v_2, \ldots, v_m\ \ \mathsf{Y}(\mathcal{D}_{o,X}(v_1\ \mathsf{in}\ t_1)) \wedge \mathsf{Y}(\mathcal{D}_{o,X\cup\{v_1\}}(v_2\ \mathsf{in}\ t_2)) \wedge \ldots \wedge$
   $\mathsf{Y}(\mathcal{D}_{o,X\cup\{v_1,\ldots v_{m-1}\}}(v_m\ \mathsf{in}\ t_m)) \Rightarrow \mathcal{A}_{o,X\cup\{v_1,\ldots,v_m\}}(ax')$

4. $\mathcal{A}_{o,X}(\mathsf{if}\ p\ \mathsf{then}\ ax_1\ \mathsf{else}\ ax_2\ \mathsf{fi}) \equiv$
   $(\mathsf{Y}(\mathcal{P}_{o,X}(p)) \Rightarrow \mathcal{A}_{o,X}(ax_1)) \wedge (\neg\mathsf{Y}(\mathcal{P}_{o,X}(p)) \Rightarrow \mathcal{A}_{o,X}(ax_2))$

5. $\mathcal{A}_{o,X}(ax_1, \ldots, ax_m) \equiv \mathcal{A}_{o,X}(ax_1) \wedge \ldots \wedge \mathcal{A}_{o,X}(ax_m)$

□

In order to formalise an assignment, an auxiliary variable v' is used, which according to the mapping $\mathcal{T}_{o,X}$ introduced in definition 6.15 is bound to the value which the term t had in the last state. If the assignment is one to a variable, then the value of this variable in the current state has to meet the one of the auxiliary variable. In an assignment to an attribute, however, the definition states that the value of v' must have been used in the write action which just should have occurred for the attribute.

As already mentioned, in an action call the calling action and the one being called have to take place at the same time. Here, the latter must have occurred with its parameters set to the values determined by evaluating the terms in the previous state. This issue, too, can be expressed using auxiliary variables bound to the values which the terms had in the preceding state, again utilising the mapping $\mathcal{T}_{o,X}$ defined in definition 6.15. If the calling action cannot take place—maybe because its precondition does not hold—then the called action does not occur either, applying an "all-or-nothing principle."

In a repetitive execution, the action rule has to hold for any valid assignment to the quantified variables. Once again, the assignments used in the previous state are the ones which have to be considered.

For a conditional action rule, its proposition p is analysed in the preceding state and provided it evaluates to *true*, the action rule given in the then clause will be considered; otherwise the one in else clause will be evaluated.

For a compound action rule the conjunct of its constituents is considered. This clearly illustrates the abovementioned point that a complex action rule does *not* represent the sequence of its internal action rules: the occurrence of a complex action rule merely states that all its constituents must have happened, too.

The following example illustrates the mapping of operation definitions to axioms of a subsystem specification according to these definitions:

Example 6.22 (definition of operations) The `setKeywords` action of the `USER` class (cf. section 4.4, p. 38) specifies that a user of the system for mobile-agent-based information retrieval should have the possibility to set the keywords that documents he is looking for should contain. Its operation definition contains neither a precondition nor the declaration of local variables, but only the action term and the action rule consisting of a single assignment to an attribute:

```
setKeywords(keyw) do keywords:= keyw od;
```

For all `USER` objects usr, this operation definition is mapped to

$$usr.(\odot usr\,.\texttt{setKeywords(keyw)} \Rightarrow \mathcal{A}_{usr,\{\texttt{keyw}\}}\ (\texttt{keywords:=keyw})),$$

where by definition 6.21, 1. (b) we have

$$\mathcal{A}_{usr,\{\texttt{keyw}\}}\ (\texttt{keywords:=keyw}) \equiv \\ \exists\, k'\, \mathsf{Y}(\mathcal{T}_{usr,\{\texttt{keyw}\}}\ (k' = \texttt{keyw})) \wedge \odot usr\,.\texttt{keywords}_{write}(k'),$$

and by definition 6.15, 2. (a)

$$\mathcal{T}_{usr,\{\texttt{keyw}\}}\ (k' = \texttt{keyw}) \equiv k' = \texttt{keyw},$$

which all in all leads to

$$usr.(\odot usr\,.\texttt{setKeywords(keyw)} \Rightarrow \\ \exists\, k'\, \mathsf{Y}(k' = \texttt{keyw}) \wedge \odot usr\,.\texttt{keywords}_{write}(k')).$$

□

This is of course a fairly simple example. More complex cases for TROLL$_3$ can be found in [Har97], and we are going to provide some larger examples in conjunction with the mobility-related language constructs which we will turn to in the following section.

6.2 Stationary and Mobile Subsystems

[Eck01] introduces subsystems as a means to structure large specifications. Subsystems group classes and objects with related functionality into logical units. A difference is made between basic and complex subsystems in the way that complex subsystems may contain other—basic or complex—subsystems, while basic or simple subsystems may only comprise object classes and their instances. A specification in this approach thereby creates a hierarchy with the object system as the root, the complex subsystems as the inner nodes, and the basic subsystems as the leaf nodes.

As the concept of a hierarchy has also been used in several specification languages for mobile systems in order to express locations in relative terms (cf. section 3, e.g. table 3.1), it seems reasonable to build upon this similarity and to extend the constructs developed in [Eck01] suitably.

In our "mobile TROLL," the following three productions together replace the <*subsystemSpec*> production from [Eck01], and allow for the specification of stationary and mobile subsystem, respectively, of a compound system:

Syntax

<*statSpec*> ::= **stationary** <*ident*>
(<*statSpec*> | <*mobiSpec*> | <*subSpec*>)
{ ';' <*statSpec*> | <*mobiSpec*> | <*subSpec*> |
<*statInstanceDecl*> | <*mobiInstanceDecl*> }
[**onEntry** { <*linkDef*> ';' } [<*actionTerm*>] **end**]
[**onExit** { <*unlinkDef*> ';' } [<*actionTerm*>] **end**]
end_stationary

<*mobiSpec*> ::= **mobile** <*ident*>
(<*mobiSpec*> | <*subSpec*>)
{ ';' <*mobiSpec*> | <*subSpec*> |<*mobiInstanceDecl*> }
[**onEntering** { <*linkDef*> ';' } [<*actionTerm*>] **end**]
[**onExiting** { <*unlinkDef*> ';' } [<*actionTerm*>] **end**]
end_mobile

<*subSpec*> ::= [<*offerSpec*> { ';' <*offerSpec*> }]
[<*requestSpec*> { ';' <*requestSpec*> }]
[<*specItem*> { ';' <*specItem*> } ';']

With these productions, we can now on the one hand distinguish which parts of a complex system are mobile and which are not, and on the other hand we can still group related functionality together—something which can not be expressed by the formal approaches discussed in chapter 3.

Mobile subsystems represent the units of migration. Together with the nesting facility, it is then possible to specify that a mobile agent will migrate together with all the objects it contains, that a mobile hardware device moves together with all the subsystems it comprises, or that mobile software moves within mobile hardware. Nesting also applies to stationary entities. Note, however, that while a stationary unit may contain other mobile and stationary subsystems in addition to elementary classes and objects, mobile entities may only contain the latter and other mobile units, but no stationaries. This should seem reasonable, since any item with a built-in fixed unit can itself not be mobile. However, connections between mobile and stationary entities are possible and can be described in the form of communication links.

After introducing in the following some terminology needed in general, we will discuss basic (mobile or stationary) subsystems in the next subsection and turn to complex units in subsection 6.2.2.

We use $subSpec(\xi)$ to stand for the set of names of all subsystems contained within a complex subsystem ξ, and $mSubSpec(\xi)$ and $sSubSpec(\xi)$ to denote the set of names of all its mobile and stationary subsystems, respectively. We assume $mSubSpec(\xi) \cup sSubSpec(\xi) = subSpec(\xi)$ and $mSubSpec(\xi) \cap sSubSpec(\xi) = \{\}$.

In order to declare an instance of a mobile or stationary subsystem, an identifier has to be provided following the keyword **mobiles** or **stationary**, respectively. This identifier has to be unique throughout the complex subsystem in which the instance is declared. The type of the instance then has to be indicated after a separating colon:

Syntax

<*statInstanceDecl*> ::= **stationaries** <*ident*> **:** <*ident*>
<*mobiInstanceDecl*> ::= **mobiles** <*ident*> **:** <*ident*>

At the formal level, a declaration of a mobile or stationary subsystem is defined as follows:

Definition 6.23 (declaration of (instances) of mobile/stationary subsystems) A declaration of an instance of a mobile or stationary subsystem is given by a sort-index identifier $id_{su}, su \in mSubSpec(\xi)$, or $id_{su}, su \in sSubSpec(\xi)$, respectively.

Within the superordinate complex subsystem ξ, the identifiers of declarations of different mobile or stationary subsystem have to be pairwise disjoint. $subDecl(\xi)$ denotes the set of all subsystem declarations within the complex subsystem ξ, and $mSubDecl(\xi)$ and $sSubDecl(\xi)$ are used to refer to the sets of declarations of its mobile and stationary subsystems, respectively. We assume $mSubDecl(\xi) \cup sSubDecl(\xi) = subDecl(\xi)$ and $mSubDecl(\xi) \cap sSubDecl(\xi) = \{\}$. □

6.2.1 Basic Stationary and Mobile Subsystems

With the syntactical and semantical definition of object classes from section 6.1 at hand, we can now turn to simple or basic mobile and stationary subsystems, i.e. those not containing other subsystems but only conventional TROLL specifications.

We will have to consider the specification of interface objects, the declaration of internal objects, and the specification of interactions between objects. Together with the operation definitions local to the objects, the latter describes the behaviour of a subsystem.

In addition to the objects discussed so far, subsystems may also contain special interface objects, which are used to describe the services a subsystem offers to or expects to be offered by other subsystems. The former are referred to as offer-objects, the latter as request-objects.

Syntax

<*offerSpec*> ::= **offer_object** <*ident*>
actions <*actionDecl*> { ';' <*actionDecl*> } ';'
[<*behaviorSpec*>]
end

<*requestSpec*> ::= **request_object** <*ident*>
actions <*actionDecl*> { ';' <*actionDecl*> } ';'
[<*behaviorSpec*>]
end

In contrast to the conventional objects discussed earlier, [Eck01] does not permit having several interface objects of the same "type." It is possible to specify a number of request-objects and offer-objects for the same subsystem, even to have several with the same behaviour and signature, but multiple instantiation is not allowed. An indication of an object class as a kind of "object generator" is therefore not required; interface objects are specified directly (as objects), identified by a unique name.

A specification of an interface object implicitly constitutes an object sort. Definition 6.1 can therefore be extended to:

Definition 6.24 (object sorts from classes and interface objects) Let $ic(\psi)$ denote the set of all identifiers of interface objects and $oc(\psi)$ the set of all class identifiers of a subsystem specification ψ, then

$$S_O = oc(\psi) \cup ic(\psi), oc(\psi) \cap ic(\psi) = \{\}$$

holds for the set of object sorts S_O of the class signature $\Sigma_C(\psi) = (S_O, I, AT, AC)$ derived from ψ. □

Every object class c induces a special data sort $|c|$, referred to as "identification sort." The values of an identification sort represent references to objects of class c. The references have to be unique throughout a subsystem. [Har97] describes how to construct these identification sorts; with regard to this thesis it suffices to take their existence for granted.

In addition to the interface objects declared explicitly for subsystems according to [Eck01], a special location-reflecting interface object is implied by and associated with every mobile or stationary subsystem at the formal level. There is always exactly one location-reflecting interface object per subsystem, mirroring the fact that a stationary or mobile unit can never be in more than one place at a time. Different from the interface objects in [Eck01], a location-reflecting interface object has one attribute, which is used to maintain the information about a unit's location. These special interface objects should all have the same name, and this should also hold for their dedicated attribute, which for this thesis we assume to be *loc* and λ, respectively.

With the help of the attribute λ, the location-reflecting interface object can provide a string designating the path from the object system to the subsystem enclosing the one which the object itself is part of, naming all other enclosing subsystems from the root to one immediately surrounding it in descending order. The evaluation of these attributes is done recursively over the location hierarchy, which is why the corresponding definition (6.50) is postponed until section 6.2.2, "Complex Stationary and Mobile Subsystems."

With regard to chapter 3, note that while our TROLL extension provides means to express location in relative terms, at the formal level we turn to the absolute characterisation (cf. e.g. table 3.1). A welcome side effect of this is that we therefore entirely comply with the Module Labelled Event Structure developed in [KF00], and hence can use this as the formal semantics for our constructs, cf. definition 5.52.

Example 6.25 (object sorts) The mobile subsystem `RetrievalUnit` contains the object class `RETRIEVER` and the interface objects `searchFacility`, `tripPlanner`, `searchEngine`. Furthermore, at the formal level the special location-reflecting interface object *loc* is also associated with it, leading to

$$S_O = \{\texttt{RETRIEVER, searchFacility, tripPlanner, searchEngine,}loc\}$$

for $\Sigma_C(\text{RetrievalUnit}) = (S_O, I, AT, AC)$. □

The set of possible instances for the specified object classes is given by the object declarations. At system run-time, concrete instances can be created using special constructor methods, sometimes also referred to as "birth actions." In TROLL, the internal objects of a subsystem are declared after the keyword **objects** by providing a (possibly parameterised) identifier, which has to be unique within the subsystem, followed by the name of the class the object should be an instance of:

Syntax

<*instanceDecl*> ::= **objects** <*ident*>[(<*field*>)] ':' <*ident*>
<*field*> ::= [<*ident*> ':'] <*type*>

At the formal level, object declarations can be defined as follows:

Definition 6.26 (object declaration) An object declaration is given by a sort-indexed identifier $id_{s,c}, s \in ds(\psi) \cup \{\epsilon\}, c \in oc(\psi)$.

Within the subsystem specification ψ, the identifiers of the different object declarations have to be pairwise disjoint. $obj(\psi)$ denotes the set of all object declarations within the subsystem specification ψ. □

Example 6.27 (set of object declarations) The subsystem `RetrievalUnit` of the example specification contains the object class `RETRIEVER`, for which a corresponding instance is declared by `objects retriever: RETRIEVER`. As no other declarations are made within this subsystem,

$$obj(\texttt{RetrievalUnit}) = \{\texttt{retriever}_{\epsilon,\texttt{RETRIEVER}}\}$$

holds for it. □

Just like for the common interface objects of [Eck01], multiple instantiation is not available for location-reflecting interface objects, and they, too, can be specified directly as objects provided they are given a unique name. Such a specification also implies an object sort (as described with regard to definition 6.24), leading to

Definition 6.28 (declaration of interface objects) A declaration of an interface object is given by a sort-indexed identifier $id_{\epsilon,id}$, where $id \in ic(\psi)$. Additionally, a special location-reflecting interface object declared by $loc_{\epsilon,loc}$ is implied by and associated with every mobile or stationary subsystem. Within a subsystem specification ψ the identifiers of interface objects and object declarations have to be pairwise disjoint.

The set of all interface objects of a subsystem specification ψ is denoted by $sobj(\psi)$, consisting of

- the set of offer-objects $sobj_{\mathsf{Offer}}(\psi)$,
- the set of request-objects $sobj_{\mathsf{Request}}(\psi)$, and
- the set containing the location-reflecting interface object of ψ, $sobj_{\mathsf{Location}}(\psi)$,

where

$$\begin{aligned} sobj_{\mathsf{Offer}}(\psi) \cap sobj_{\mathsf{Request}}(\psi) &= \{\}, \\ sobj_{\mathsf{Request}}(\psi) \cap sobj_{\mathsf{Location}}(\psi) &= \{\}, \\ sobj_{\mathsf{Location}}(\psi) \cap sobj_{\mathsf{Offer}}(\psi) &= \{\}, \text{ and} \\ sobj_{\mathsf{Offer}}(\psi) \cup sobj_{\mathsf{Request}}(\psi) \cup sobj_{\mathsf{Location}}(\psi) &= sobj(\psi) \end{aligned}$$

□

As identifiers of interface objects cannot be parameterised, the value for the first sort indicated here will always be ϵ, as opposed to the one in the declaration of conventional objects (cf. definition 6.26). The subscript for the second sort, which in definition 6.26 identifies the object class, could also be dropped, since the identifiers of the interface objects represent both the sort and the instance at the same time. However, in order to simplify subsequent definitions we stick to the indexing method for identifiers introduced in definition 6.26.

The following example illustrates definition 6.28:

Example 6.29 (declaration of interface objects) The mobile subsystem `RetrievalUnit` contains two offer-objects and one request-object:

```
mobile RetrievalUnit
  offer_object searchFacility
    actions
      searchKeywords(keywords: set(string),
                     homeAndFirstTarget: list(string),
                     ! resultSet: set(litInfo));
  end;
  request_object tripPlanner
    actions
      getNextTarget(someLoc: string, ! nextLoc: string);
  end;
  request_object searchEngine
    actions
      searchKeywords(keywords: set(string),
                     ! resultSet: set(litInfo));
  end;
  ...
end_mobile;
```

We therefore obtain

$$\begin{aligned} sobj(\texttt{RetrievalUnit}) = \{&\texttt{searchFacility}_{\epsilon,\texttt{searchFacility}}, \\ &\texttt{tripPlanner}_{\epsilon,\texttt{tripPlanner}}, \\ &\texttt{searchEngine}_{\epsilon,\texttt{searchEngine}}, \\ &loc_{\epsilon,loc}\} \text{ where} \end{aligned}$$

$$
\begin{aligned}
sobj_{\mathsf{Offer}}(\texttt{RetrievalUnit}) &= \{\texttt{searchFacility}_{\epsilon,\texttt{searchFacility}}\},\\
sobj_{\mathsf{Request}}(\texttt{RetrievalUnit}) &= \{\texttt{tripPlanner}_{\epsilon,\texttt{tripPlanner}},\\
&\quad\ \texttt{searchEngine}_{\epsilon,\texttt{searchEngine}}\}\text{, and}\\
sobj_{\mathsf{Location}}(\texttt{RetrievalUnit}) &= \{loc_{\epsilon,loc}\}
\end{aligned}
$$

at the formal level. □

As explained in [Eck01], action declarations of interface objects should differ in no way from those of object classes. This also extends to the implied location-reflecting interface objects from mobile and stationary subsystems, which—as the characteristic feature separating them from "plain" interface objects—also have one single attribute λ. Hence, $Act(c)$ and $Attr(c)$ should from now on denote, respectively, the set of all action declarations and the set of all attribute declarations for a given specification of an object class or a (location-reflecting) interface object c.

Example 6.30 (action alphabets for interface objects) For the interface object `searchEngine` of the mobile subsystem `RetrievalUnit`, the following sets of attributes and actions can be derived from the example specification:

$$
\begin{aligned}
Attr(\texttt{searchEngine}) &= \{\}\\
Act(\texttt{searchEngine}) &= Act_{update}(\texttt{searchEngine})\\
&= \{(\texttt{searchKeywords}_{\mathsf{set(string),\ set(litInfo)}},\\
&\quad\ para_{in}(\texttt{searchKeywords}_{\mathsf{set(string),\ set(litInfo)}}),\\
&\quad\ para_{out}(\texttt{searchKeywords}_{\mathsf{set(string),\ set(litInfo)}}))\}
\end{aligned}
$$

with

$$
\begin{aligned}
para_{in}(\texttt{searchKeywords}_{\mathsf{set(string),\ set(litInfo)}}) &= \{1\}\\
para_{out}(\texttt{searchKeywords}_{\mathsf{set(string),\ set(litInfo)}})) &= \{2\}.
\end{aligned}
$$

□

Note, that for the location-reflecting interface object *loc* implied by and associated with the mobile subsystem we can derive $Attr(loc) = \{\lambda_{\mathsf{string}}\}$, but no $Act(loc)$, as we do not have any *action* declaration related to this object. Nevertheless, according to definition 6.8 for a class signature we would obtain corresponding action symbols from this *attribute* declaration.

Having defined the explicit declaration of objects within the body of a subsystem on the one hand and the implicit declaration of interface objects on the other, we are now able to determine the family I of sets of instance symbols of the induced class signature as the union of the set of object declarations and the set of interface object declarations:

Definition 6.31 (instance symbols) For the class signature $\Sigma_C(\psi) = (S_O, I, AT, AC)$ induced by the subsystem specification ψ holds

$$id : s \to c \in I$$

if and only if $id_{s,c} \in obj(\psi) \cup sobj(\psi), s \in ds(\psi) \cup \{\epsilon\}, c \in oc(\psi)$. □

Example 6.32 (instance symbols for a class signature) Using definition 6.31, the instance symbols for the class signature $\Sigma_C(\texttt{RetrievalUnit}) = (S_O, I, AT, AC)$ induced by the subsystem specification `RetrievalUnit` are:

$$\begin{aligned} I \ = \{&\texttt{searchFacility} : \rightarrow \texttt{searchFacility}, \\ &\texttt{tripPlanner} : \rightarrow \texttt{tripPlanner}, \\ &\texttt{searchEngine} : \rightarrow \texttt{searchEngine}, \\ &loc : \rightarrow loc, \\ &\texttt{retriever} : \rightarrow \texttt{RETRIEVER}\} \end{aligned}$$

□

The family AT of sets of attribute symbols and the family AC of sets of action symbols are obtained in a similar way:

Definition 6.33 (attribute symbols and action symbols) For the class signature $\Sigma_C(\psi) = (S_O, I, AT, AC)$ induced by the subsystem specification ψ holds:

1. if $a_s \in Attr(c)$ for $c \in oc(\psi) \cup ic(\psi)$, then $a : c \rightarrow s \in AT$.
2. if $a_{s^*} \in Act(c)$ for $c \in oc(\psi) \cup ic(\psi)$, then $a : s^* \rightarrow c \in AC$.

□

Example 6.34 (class signatures for interface objects) Similar to [Eck01, pp. 121, 131, and 29] and using the notation applied there, for

$$sobj_{\mathsf{Offer}}(\texttt{RetrievalUnit}) = \{\texttt{searchFacility}_{\epsilon,\texttt{searchFacility}}\}$$

we obtain

$$\begin{aligned} \Sigma_C(\texttt{searchFacility}) \ &= (S_O, I, AT, AC) \\ &= (\{\texttt{searchFacility}\}, \\ &\quad \{\texttt{searchFacility} : \rightarrow \texttt{searchFacility}\}, \\ &\quad \{\}, \\ &\quad \{\texttt{searchKeywords} : \mathsf{set(string)} \times \mathsf{list(string)} \\ &\quad \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchFacility}\}), \end{aligned}$$

and, analogously,

$$\begin{aligned} \Sigma_C(loc) \ &= (S_O, I, AT, AC) \\ &= (\{loc\}, \\ &\quad \{loc : \rightarrow loc\}, \\ &\quad \{\lambda_{\mathsf{string}} : loc \rightarrow \mathsf{string}\}, \\ &\quad \{\lambda_{read} : \mathsf{string} \rightarrow loc, \lambda_{write} : \mathsf{string} \rightarrow loc\}) \end{aligned}$$

for $sobj_{\mathsf{Location}}(\texttt{RetrievalUnit}) = \{loc_{\epsilon,loc}\}$. □

The latter can be generalised to any specification of a mobile or stationary subsystem:

Definition 6.35 (class signature of location-reflecting interface objects) The class signature for a location-reflecting interface object $sobj_{\mathsf{Location}}(\psi) = \{loc_{\epsilon,loc}\}$ implied by and associated with every mobile or stationary subsystem ψ is given by

$$\begin{aligned}\Sigma_C(loc) &= (S_O, I, AT, AC)\\ &= (\{loc\},\\ &\quad \{loc :\rightarrow loc\},\\ &\quad \{\lambda_{\mathsf{string}} : loc \rightarrow \mathsf{string}\},\\ &\quad \{\lambda_{read} : \mathsf{string} \rightarrow loc, \lambda_{write} : \mathsf{string} \rightarrow loc\})\end{aligned}$$

for a mobile subsystem and by

$$\begin{aligned}\Sigma_C(loc) &= (S_O, I, A)\\ &= (\{loc\},\\ &\quad \{loc :\rightarrow loc\},\\ &\quad \{\lambda_{\mathsf{string}} : loc \rightarrow \mathsf{string}\},\\ &\quad \{\lambda_{read} : \mathsf{string} \rightarrow loc\})\end{aligned}$$

for a stationary subsystem. □

In order to be able to describe the interactions between several objects and/or interface objects within the body of a subsystem, we need a way to express which instances are accessed. Object terms provide such means. In TROLL, they obey the following syntax:

Syntax

<objectTerm> ::= *<ident>*[(*<dataTerm>*)]

As already mentioned in the remarks to definition 6.24, any object class c implies an identification sort $|c|$, whose values represent references to objects of class c. The references have to be unique throughout a subsystem. The following definition describes object terms at the formal level, and these formal object terms correspond to those references. Their sorts are therefore given by the sort $|c|$ of the respective class c.

Definition 6.36 (object terms) The set of object terms for a subsystem specification ψ is defined as follows:

1. If $id_{s,c} \in obj(\psi)$ and t is a data term of sort s, then $id(t)$ is an object term of sort $|c|$ in ψ.

2. If $id_{\epsilon,c} \in obj(\psi)$, then id is an object term of sort $|c|$ in ψ.
3. If $id_{\epsilon,id} \in sobj(\psi)$, then id is an object term of sort $|id|$ in ψ.
4. Nothing else is an object term.

□

Object terms constructed on the basis of object declarations can therefore be parameterised, but this is not required. Object terms constructed on the basis of interface objects, however, can never be parameterised. The object terms introduced here directly correspond to the identity terms over an extended data signature from definition 5.11.

Example 6.37 (object terms) Examples of object terms with regard to the specification of the subsystem `RetrievalUnit` are

- `retriever` for the instance of the `RETRIEVER` class,
- `searchEngine` for the interface object `searchEngine`, and
- *loc* for the implied location-reflecting interface object *loc*.

□

With the means at hand to specify object classes and interface objects, to declare instances, and to construct object terms in order to refer to these instances and those of interface objects, we can now turn to the specification of interactions. We will start with interactions between internal objects of a subsystem, including its interface objects. This type of interaction is called "subsystem internal interaction."

All the instances within a subsystem—those explicitly declared for object classes as well as the implicitly declared interface objects—are initially assumed to be completely independent of each other. From the local point of view, the actions taking place within every object occur sequentially, but with regard to the other objects they are completely concurrent. However, in the underlying model every object corresponds to a sequential event structure, and the interactions—represented by the joint occurrence of local actions—synchronise these objects.

With regard to a subsystem, interactions form global behaviour rules, which together with those local to an object collectively describe the behaviour of the unit. In the chapter on theoretical foundations, in definition 5.15 the set Ac of global actions was introduced as $Ac \subseteq 2^{LAc} - \{\}$ with the additional restriction that every object may participate in a global action with at most one local action, LAc denoting the set of all local actions of all objects. With the help of the global behaviour specifications of a TROLL specification, the set Ac of all global actions gets restricted to those global actions explicitly given. Syntactically, in TROLL these behaviour specifications are given by additional operation definitions at the subsystem level for the actions of chosen objects.

Syntax

<*behaviorSpec*> ::= **behavior** <*operationDef*> { ';'<*operationDef*> } ';'
end

The structure of an operation definition at subsystem level corresponds to the one for object classes, comprising an obligatory action term and a precondition, a set of variable declarations, and an action rule, the latter three all being optional.

Syntax

<*operationDef*> ::= <*actionTerm*>
[**onlyIf** <*pFormula*>]
[<*variableDecl*>]
[**do** <*actionRule*> **od**]

The action term identifies the action for which the behaviour specification is given. Concerning an action term at the subsystem level, first the reference to the object has to be given and then an action term according to definition 6.16 needs to be constructed locally to the class of the identified object or to the interface object:

Definition 6.38 (action terms of subsystems) The set of action terms of a subsystem specification ψ is defined by:

1. If α is an action term of object class c, $id_{s,c} \in obj(\psi)$, and v is a variable of sort s, then $id(v).\alpha$ is an action term and $var_{in}(id(v).\alpha) = \{v\} \cup var_{in}(\alpha)$ and $var_{out}(id(v).\alpha) = var_{out}(\alpha)$.

2. If α is an action term of object class c, $id_{\epsilon,c} \in obj(\psi)$, then $id.\alpha$ is an action term and $var_{in}(id.\alpha) = var_{in}(\alpha)$ and $var_{out}(id.\alpha) = var_{out}(\alpha)$.

3. If $id_{\epsilon,id} \in sobj(\psi)$ and if α is an action term of the interface object id, then $id.\alpha$ is an action term and $var_{in}(id.\alpha) = var_{in}(\alpha)$ and $var_{out}(id.\alpha) = var_{out}(\alpha)$.

4. Nothing else is an action term of ψ.

□

In contrast to behaviour specifications made locally to a class, which hold for all its instances, a behavioural rule given at the subsystem level refers only to the object whose identity can be derived from the object term forming part of

the "subsystem action term." Much like this definition extends definition 6.16, [Eck01] also provides modified versions of definition 6.17, 6.18, 6.19, 6.20, and 6.21, but as they are not relevant for the rest of this thesis we refrain from providing corresponding counterparts here.

As the signature of a basic module consists of a kernel signature and a set of export signatures, accordingly first these parts have to be derived from the subsystem specification. Among others, a kernel signature has to contain a local module sort. Since every subsystem should correspond to a module, the subsystem's name will be used as the local module sort. Every interface object is represented by an export module, and, analogously, using the interface object's name as the local module sort seems obvious.

Definition 6.39 (local module sorts) Let su denote a given subsystem specification and $ic(su) = \{ids_1, \ldots, ids_n\}$ the set of all identifiers of su's interface objects (including the location-reflecting one).

1. $\alpha = su$ holds for the local module sort α of the module kernel signature derivable from su.

2. $\alpha_i = ids_i, i = 1, \ldots, n$, holds for the local module sorts of the module kernel signatures derivable from the interface objects.

□

Based on this, the set of internal module sorts can be constructed and the set of export module sorts be determined:

Definition 6.40 (internal module sorts and export module sorts) Let a subsystem specification ψ with local module sort α and the set $ic(\psi) = \{ids_1, \ldots, ids_n\}$ of all interface objects (including the location-reflecting one) of ψ with local module sorts $\alpha_1, \ldots, \alpha_n$, respectively, be given.

1. $S^+_M = \{\alpha\}$ and $S^e_M = \{\alpha, \alpha_1, \ldots, \alpha_n\}$ hold for the set S^+_M of internal module sorts and the set S^e_M of export module sorts of the module kernel signature $\Sigma_K(\psi) = (S, \Omega)$ derivable from the subsystem specification ψ.

2. $S^+_{M_i} = \{\alpha, \alpha_i\}$ and $S^e_{M_i} = \{\alpha_i\}, i = 1, \ldots, n$, hold for the set $S^+_{M_i}$ of internal module sorts and the sets $S^e_{M_i}$ of export module sorts of the module kernel signatures Σ_{K_i} derivable from the interface objects $ids_1, \ldots, ids_n$.

□

One part of an export signature is given by a module kernel signature, which can be derived from an interface object by constructing the class signature for this object and choosing the module sorts as described above. Furthermore, an export signature requires an inclusion morphism μ embedding the module kernel signature Σ_{K_2} of the interface object into the module kernel signature Σ_{K_1} of the subsystem. As due to the definitions given so far $\Sigma_{K_2} \subseteq \Sigma_{K_1}$ always holds, such

an inclusion morphism is always given. Export signatures and also signatures of basic modules (apart from instance symbols) can therefore be derived from modular TROLL specifications. Before turning to complex subsystems and complex modules, let us illustrate these definitions by means of an example:

Example 6.41 (subsystems as basic modules) We consider the specification of the mobile subsystem `RetrievalUnit` of the example specification. Analogously to [Eck01], assume given some suitable data signature $\Sigma_D = (S_D, \Omega_D)$, which contains at least basic data sorts like string, nat, int, etc., but also the data sort `litInfo` (cf. section 4.4), whose structure here will not be analysed any further.

From example 6.25 we know that the set of object sorts is given by

$$S_O = \{\texttt{RETRIEVER}, \texttt{searchFacility}, \texttt{tripPlanner}, \texttt{searchEngine}, loc\},$$

giving rise to

$$\begin{aligned} I \ = \{&\texttt{retriever} : \rightarrow \texttt{RETRIEVER},\\ &\texttt{searchFacility} : \rightarrow \texttt{searchFacility},\\ &\texttt{tripPlanner} : \rightarrow \texttt{tripPlanner},\\ &\texttt{searchEngine} : \rightarrow \texttt{searchEngine},\\ &loc : \rightarrow loc\} \end{aligned}$$

and

$$\begin{aligned} AT \ = \{&\texttt{homeLoc} : \texttt{RETRIEVER} \rightarrow \mathsf{string},\\ &\ldots,\\ &\texttt{litrInfo} : \texttt{RETRIEVER} \rightarrow \mathsf{set}(\texttt{litInfo}),\\ &\lambda_{\mathsf{string}} : loc \rightarrow \mathsf{string}\}\\ AC \ = \{&\texttt{becomeRetriever} : \rightarrow \texttt{RETRIEVER},\\ &\ldots,\\ &\texttt{cease} : \rightarrow \texttt{RETRIEVER},\\ &\texttt{homeLoc}_{read} : \mathsf{string} \rightarrow \texttt{RETRIEVER},\\ &\ldots,\\ &\texttt{litrInfo}_{write} : \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{RETRIEVER},\\ &\texttt{searchKeywords} :\\ &\mathsf{set}(\mathsf{string}) \times \mathsf{list}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchFacility},\\ &\texttt{getNextTarget} : \mathsf{string} \times \mathsf{string} \rightarrow \texttt{tripPlanner},\\ &\texttt{searchKeywords} : \mathsf{set}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchEngine},\\ &\lambda_{read} : \mathsf{string} \rightarrow loc,\\ &\lambda_{write} : \mathsf{string} \rightarrow loc\} \end{aligned}$$

From this class signature of the mobile subsystem `RetrievalUnit`, $\Sigma_C(\texttt{RetrievalUnit}) = (S_O, I, AT, AC)$, the following parts of the module kernel

signature Σ_K can be derived:

$$
\begin{aligned}
S_O^i &= \{\texttt{RETRIEVER}^i, \texttt{searchFacility}^i, \texttt{tripPlanner}^i, \texttt{searchEngine}^i, loc^i\},\\
S_O^{at} &= \{\texttt{RETRIEVER}^{at}, \texttt{searchFacility}^{at}, \texttt{tripPlanner}^{at}, \texttt{searchEngine}^{at}, loc^{at}\},\\
S_O^{ac} &= \{\texttt{RETRIEVER}^{ac}, \texttt{searchFacility}^{ac}, \texttt{tripPlanner}^{ac}, \texttt{searchEngine}^{ac}, loc^{ac}\},\\
\Omega_O^i &= \{\texttt{retriever} : \rightarrow \texttt{RETRIEVER}^i, \texttt{searchFacility} : \rightarrow \texttt{searchFacility}^i,\\
&\quad \texttt{tripPlanner} : \rightarrow \texttt{tripPlanner}^i, \texttt{searchEngine} : \rightarrow \texttt{searchEngine}^i,\\
&\quad loc : \rightarrow loc^i\}\\
\Omega_O^{at} &= \{\texttt{homeLoc} : \texttt{RETRIEVER}^i \times \texttt{RETRIEVER}^{at} \rightarrow \mathsf{string},\\
&\quad \ldots,\\
&\quad \texttt{litrInfo} : \texttt{RETRIEVER}^i \times \texttt{RETRIEVER}^{at} \rightarrow \mathsf{set}(\texttt{litInfo}),\\
&\quad \lambda_{\mathsf{string}} : loc^i \times loc^{at} \rightarrow \mathsf{string}\},\\
\Omega_O^{ac} &= \{\texttt{becomeRetriever} : \texttt{RETRIEVER}^i \rightarrow \texttt{RETRIEVER}^{ac},\\
&\quad \ldots,\\
&\quad \texttt{searchKeywords} :\\
&\quad \texttt{searchEngine}^i \times \mathsf{set}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchEngine}^{ac},\\
&\quad \lambda_{read} : loc^i \times \mathsf{string} \rightarrow loc^{ac},\\
&\quad \lambda_{write} : loc^i \times \mathsf{string} \rightarrow loc^{ac}\}
\end{aligned}
$$

Furthermore, from definition 6.39 and with $ic(\texttt{RetrievalUnit}) = \{\texttt{searchFacility}, \texttt{tripPlanner}, \texttt{searchEngine}, loc\}$ as the set of all identifiers of interface objects for the specification of the mobile subsystem we obtain

1. $\alpha = \texttt{RetrievalUnit}$ and

2. $\alpha_1 = \texttt{searchFacility}, \alpha_2 = \texttt{tripPlanner}, \alpha_3 = \texttt{searchEngine}$, and $\alpha_4 = loc$.

With definition 6.40 (1.) this gives

$$S_M^e = \{\texttt{RetrievalUnit, searchFacility, tripPlanner, searchEngine}, loc\}$$

and $S_M^+ = \{\texttt{RetrievalUnit}\}$, making $\alpha \in S_M^e \cap S_M^+ = \{\texttt{RetrievalUnit}\}$ the local module sort. □

All that is required of Ω_M due to definition 5.38 is that for every $m \in S_M$ a unique instance symbol exists in $\Omega_{M;\epsilon,m}$. However, as we intend a close correspondence between (sub-) system specifications and signatures, we determine the additional operation symbols in Ω_M in a way that relates them to identifiers used in the declaration of (sub-) systems and their interface objects, which already brings us to complex subsystems.

6.2.2 Complex Stationary and Mobile Subsystems

So far, our focus was on specifications of simple subsystem, i.e. those not containing other subsystems but only classes, objects, data types, etc. We now turn to complex subsystems, which do contain other subsystems. The corresponding TROLL syntax and the formal definition for the declaration of mobile and stationary subsystems were already given in the beginning of section 6.2, and we can therefore directly move on to how to determine instance symbols and instance terms for subsystems in a formal way:

Definition 6.42 (instance symbols and instance terms for subsystems) Let the declaration of an instance $id_{su} \in subDecl(\xi)$ of a mobile or stationary subsystem be given. According to definition 6.39, the local module sort for the kernel signature derived from su is also denoted by su. The corresponding instance symbol for the module is then defined as

$$id :\rightarrow su, id \in \Omega_{M;\epsilon,su}.$$

Let $ic(su) = \{loc, ids_1, \ldots, ids_n\}$ denote the set of su's interface objects, with local module sorts $loc, ids_1, \ldots, ids_n$. The instance symbols for the basic modules derivable from the interface objects are defined as

$$id.loc :\rightarrow loc \text{ and } id.ids_i :\rightarrow ids_i$$

for $id \in \Omega_{M;\epsilon,su}$ and $ids_i \in \Omega_{M;\epsilon,ids_i}, i = 1, \ldots, n$.

If $\omega :\rightarrow \alpha, \alpha \in S_M$, is an instance symbol from $\Omega_{M;\epsilon,\alpha}$, then ω is an instance term of sort α. □

According to this definition, the identifier used in the declaration of an instance of a subsystem is reused as the instance symbol in the signature corresponding to this subsystem. As such a declaration is always made outside the scope of the subsystem specification, this definition can not be used at the topmost level, the system specification. The following definition copes with this case:

Definition 6.43 (module sorts for object systems) Let a TROLL specification with the identifier id be given. The local module sort is defined as os, the instance symbol as $id :\rightarrow os, id \in \Omega_{M;\epsilon,os}$ and the instance term as id. □

Example 6.44 (subsystems as basic modules ctd.) Using definition 6.42, from

$$S^e_M \cup S^+_M = \{\texttt{RetrievalUnit, searchFacility, tripPlanner,}\\ \texttt{searchEngine}, loc\}$$

and

```
mobiles retrievalUnit:  RetrievalUnit
```

in `stationary LiteratureSystem` we obtain

$$\begin{aligned}
\Omega_{M;\epsilon,\texttt{RetrievalUnit}} &= \{\texttt{retrievalUnit}\},\\
\Omega_{M;\epsilon,\texttt{searchFacility}} &= \{\texttt{retrievalUnit.searchFacility}\},\\
&\vdots\\
\Omega_{M;\epsilon,loc} &= \{\texttt{retrievalUnit}.loc\}.
\end{aligned}$$

and hence

$$\begin{aligned}
\Omega_M = \{&\texttt{retrievalUnit} : \rightarrow \texttt{RetrievalUnit},\\
&\texttt{retrievalUnit.searchFacility} : \rightarrow \texttt{searchFacility},\\
&\texttt{retrievalUnit.tripPlanner} : \rightarrow \texttt{tripPlanner},\\
&\texttt{retrievalUnit.searchEngine} : \rightarrow \texttt{searchEngine},\\
&\texttt{retrievalUnit}.loc : \rightarrow loc\}
\end{aligned}$$

All in all, the module kernel signature is therefore

$$\Sigma_K = (S, \Omega) = (S_D \cup S_O^i \cup S_O^a \cup S_M, \Omega_D \cup \Omega_O^i \cup \Omega_O^a \cup \Omega_M).$$

In order to determine the signature of the basic module $\Theta = (\Sigma_K, Exp)$ corresponding to the mobile subsystem `RetrievalUnit`, all that remains to be fixed is the set of export signatures Exp. As the subsystem contains three interface objects, and a mobile subsystem also implies and is associated with a location-reflecting interface object loc, this set is given by $Exp = \{E_1, E_2, E_3, E_4\}$. We illustrate how to determine these export signatures for the example of the location-reflecting interface object and the interface object `searchFacility`

According to definition 6.35, the class signature for the location-reflecting interface object implied by and associated with the mobile subsystem `RetrievalUnit` is given by

$$\begin{aligned}
\Sigma_C(loc) &= (S_O, I, AT, AC)\\
&= (\{loc\},\\
&\quad \{loc : \rightarrow loc\},\\
&\quad \{\lambda_{\mathsf{string}} : loc \rightarrow \mathsf{string}\},\\
&\quad \{\lambda_{read} : \mathsf{string} \rightarrow loc, \lambda_{write} : \mathsf{string} \rightarrow loc\}),
\end{aligned}$$

from which the extended data signature given by

$$\begin{aligned}
S_{O_4}^i &= \{loc^i\},\\
S_{O_4}^{at} &= \{loc^{at}\},\\
S_{O_4}^{ac} &= \{loc^{ac}\},\\
\Omega_{O_4}^i &= \{loc : \rightarrow loc^i\}\\
\Omega_{O_4}^{at} &= \{\lambda_{\mathsf{string}} : loc^i \times loc^{at} \rightarrow \mathsf{string}\}\\
\Omega_{O_4}^{ac} &= \{\lambda_{read} : loc^i \times \mathsf{string} \rightarrow loc^{ac}, \lambda_{write} : loc^i \times \mathsf{string} \rightarrow loc^{ac}\}
\end{aligned}$$

and some suitable S_{D_4} ($= S_D$) and Ω_{D_4} ($= \Omega_D$) as outlined in the beginning of this example can be derived. With definition 6.40 (2.), the required module sorts for the module kernel signature are $S^+_{M_4} = \{\texttt{RetrievalUnit}, loc\}$ and $S^e_{M_4} = \{loc\}$, making $\alpha_{M_4} = loc$ the local module sort. $\Omega_{M_4} = \{\texttt{RetrievalUnit}, loc\}$ follows from $S_{M_4} = S^e_{M_4} \cup S^+_{M_4}$. Since $\Sigma_{K_4} \subseteq \Sigma_K$, also the required inclusion morphism μ_4 exists and hence $E_4 = (\Sigma_{M_4}, \mu_4)$ derived from loc is one of the required export signatures:

1. $loc \in \{\texttt{RetrievalUnit}, \texttt{searchFacility}, \texttt{tripPlanner}, \texttt{searchEngine}, loc\}$,
2. $\texttt{RetrievalUnit} \in \{\texttt{RetrievalUnit}, loc\}$,
3. $\{loc\} = \{loc\}$, and
4. $loc \neq \texttt{RetrievalUnit}$ and $\Sigma_{M_4} \neq \Sigma_K$

all hold.

Analogously, for

$$\begin{aligned}\Sigma_C(\texttt{searchFacility}) &= (S_O, I, AT, AC)\\ &= (\{\texttt{searchFacility}\},\\ &\quad \{\texttt{searchFacility} : \rightarrow \texttt{searchFacility}\},\\ &\quad \{\},\\ &\quad \{\texttt{searchKeywords} : \mathsf{set}(\mathsf{string}) \times \mathsf{list}(\mathsf{string})\\ &\quad \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchFacility}\})\end{aligned}$$

we obtain

$$\begin{aligned}S^i_{O_1} &= \{\texttt{searchFacility}^i\},\\ S^{at}_{O_1} &= \{\texttt{searchFacility}^{at}\},\\ S^{ac}_{O_1} &= \{\texttt{searchFacility}^{ac}\},\\ \Omega^i_{O_1} &= \{\texttt{searchFacility} : \rightarrow \texttt{searchFacility}^i\}\\ \Omega^{at}_{O_1} &= \{\}\\ \Omega^{ac}_{O_1} &= \{\texttt{searchKeywords} : \texttt{searchFacility}^i \times \mathsf{set}(\mathsf{string}) \times \mathsf{list}(\mathsf{string})\\ &\quad \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchFacility}^{ac}\}.\end{aligned}$$

With $S^+_{M_1} = \{\texttt{RetrievalUnit}, \texttt{searchFacility}\}$ and $S^e_{M_1} = \{\texttt{searchFacility}\}$, which makes $\alpha_{M_1} = \{\texttt{searchFacility}\}$ the local module sort, and $\Omega_{M_1} = \{\texttt{RetrievalUnit, searchFacility}\}$ this yields

$$\begin{aligned}E_1 &= (\Sigma_{K_1}, \mu_1)\\ &= ((S_{D_1} \cup S^i_{O_1} \cup S^a_{O_1} \cup S^e_{M_1} \cup S^+_{M_1}, S_{\Omega_1} \cup \Omega^i_{O_1} \cup \Omega^a_{O_1} \cup \Omega_{M_1}), \mu_1)\end{aligned}$$

as the export signature derived from `searchFacility`. □

Compared to basic modules, which rely on a kernel signature, the syntactical foundation for complex subsystems are module signatures requiring an extended kernel signature, as they permit to make statements about the module body and imported modules in addition to export signatures. These extensions, however, only effect the complex subsystems themselves. Their interface objects are described by simple kernel signatures as usual. At the formal level, the internal subsystems are represented by importing the basic module signatures derivable from their interface objects into the module representing the superordinate system.

Definition 6.45 (module sorts) Let a subsystem specification ω be given with local module sort α and $ic(\omega) = \{ids_1, \ldots, ids_n\}$ the set of all identifiers of interface objects of ω with local module sorts $\alpha_1, \ldots, \alpha_n$. Let $subSpec(\omega) = \{\omega_1, \ldots, \omega_m\}$ denote the set of internal subsystems of ω with $\{\omega_{j_1}, \ldots, \omega_{j_g}\} = mSubSpec(\omega) \subseteq subSpec(\omega)$ and $\{\omega_{l_1}, \ldots, \omega_{l_h}\} = sSubSpec(\omega) \subseteq subSpec(\omega)$, $mSubSpec(\omega) \cap sSubSpec(\omega) = \{\}$, the sets of its mobile and stationary internal subsystems, respectively. Let $ic(\omega_i) = \{ids_{i,1}, \ldots, ids_{i,k_i}\}$ be the set of all interface objects of the internal subsystem $\omega_i \in subSpec(\omega), i = 1, \ldots, m$, with local module sorts $\gamma_{i,1}, \ldots, \gamma_{i,k_i}$. The following holds for the set of module sorts $S_M = S_M^e \cup S_M^+ = S_M^e \cup S_M^i \cup S_M^\times$ of the extended kernel signature $\Sigma_K(\omega) = (S, \Omega)$ derivable from ω:

1. $S_M^e = \{\alpha, \alpha_1, \ldots, \alpha_n\}$,
2. $S_M^i = \{\omega_1.\gamma_{1,1}, \ldots, \omega_1.\gamma_{1,k_1}, \ldots, \omega_m.\gamma_{m,k_m}, \omega_{j_1}, \ldots, \omega_{j_g}, \omega_{l_1}, \ldots, \omega_{l_h}\}$,
3. $S_M^\times = S_M^b \cup S_M^o$ where $S_M^b = \{\alpha, \beta\}$ and $S_M^o = \{\alpha\}$, with β as the local module sort of the body module.

□

The set of export module sorts S_M^e again contains the local module sort α, but additionally also the local module sorts of all the export signatures derived from the interface objects.

The second item defines the set of import module sorts. It consists of all the local module sorts of the export signatures derived from the interface objects of the internal subsystems which are direct child nodes of the complex subsystem under consideration. In contrast to [Eck01], for mobile and stationary subsystems in addition to the local module sorts of their interface objects also their own local module sorts are imported into the superordinate system, as the module instance terms derivable from them will be required for formulating axioms (cf. e.g. definition 6.56). Here we see that the aims for using modules and mobile and stationary units are rather oppositional, even though we use similar means to express them: while modules are used to hide internal details, mobile and stationary components allow to make location explicit.

The set of internal module sorts $S_M^\times$ is the union of the sets S_M^b and S_M^o, where S_M^o's only element is the local module sort α of the complex module examined, but

S^b_M additionally contains the local module sort of the body module signature. As we require a complex subsystem to have no objects other than interface objects, a body module signature may only provide information on data type specifications. It therefore follows that the corresponding signature has to be a simple data signature, of the form $\Sigma_\beta = (S_D, \Omega_D)$.

According to definition 5.38, kernel signatures consist of extended data signatures augmented by special module sorts and module instance symbols. Definition 6.45 determines these module sorts, and definition 6.42 explains how to obtain the corresponding instance symbols. Regarding the sorts of the imported module sorts, it seems reasonable to simply take over the corresponding instance symbols given by the respective export signatures.

In order to be able to formalise the signature part of a specification of a complex subsystem completely, we finally need to characterise the structure of the extended data signature itself. For a simple subsystem, the extended data signature was directly given by a specification in TROLL$_3$, possibly extended with interface objects.

In order to determine the extended data signature of a complex subsystem, however, for the interface objects of this subsystem first corresponding class signatures have to be provided from which in a second step matching extended data signatures can then be constructed (cf. e.g. example 6.41). Together with the extended data signatures for the body module signature and the import module signatures derived from the interface objects of the internal subsystems this forms the extended data signature of the complex subsystem. The class signatures for the interface objects are obtained as explained above (cf. definitions 6.28, 6.31, and 6.33), but this time the sorts and operation symbols are prefixed with the local module sort of the subsystem comprising the corresponding interface object (using definition 6.39 (1.)) or its instance symbol, respectively. This is due to the problem that identifiers for interface objects have to be unique only within one subsystem, but not throughout the entire system; the same identifier can be used in several subsystems.

Example 6.46 (subsystems as complex modules I) For the specification ω of the stationary subsystem denoted by `LiteratureHost1` ($= su$) the local module sort according to definition 6.39 (1.) is $\alpha = su =$ `LiteratureHost1`. With

- ic(`LiteratureHost1`) = {`searchF`, loc}
- $subSpec$(`LiteratureHost1`) = {`Reception`}
- $mSubSpec$(`LiteratureHost1`) = {}
- $sSubSpec$(`LiteratureHost1`) = {`Reception`}
- ic(`Reception`) = {`searchFacility`, loc}

we therefore obtain

1. S^e_M = {`LiteratureHost1, searchF`, loc}

2. $S^i_M = \{\texttt{Reception.searchFacility, Reception.}loc, \texttt{Reception}\}$
3. $S^\times_M = \{\texttt{LiteratureHost1}, \beta\}$

according to definition 6.45. According to definition 6.42, the corresponding module instance symbols are

$$\begin{aligned}\Omega_M \;= \{&\texttt{litHost1} : \rightarrow \texttt{LiteratureHost1},\\ &\texttt{litHost1.searchF} : \rightarrow \texttt{searchF},\\ &\texttt{litHost1.}loc : \rightarrow loc,\\ &\texttt{reception.searchFacility} : \rightarrow \texttt{searchFacility},\\ &\texttt{reception.}loc : \rightarrow loc,\\ &\texttt{reception} : \rightarrow \texttt{Reception}\\ &\texttt{bod} : \rightarrow bod\},\end{aligned}$$

taking over those instance symbols used in the export signatures of imported module signatures. □

Example 6.47 (subsystems as complex modules II) Similarly, for the specification ω of the stationary subsystem denoted by `LiteratureSystem` the local module sort is $\alpha = \texttt{LiteratureSystem}$. With

- $ic(\texttt{LiteratureSystem}) = \{\texttt{searchEng}, loc\}$
- $subSpec(\texttt{LiteratureSystem}) =$ $\{\texttt{ManagingUnit, RetrievalUnit, LiteratureHost1}\}$
- $mSubSpec(\texttt{LiteratureSystem}) = \{\texttt{RetrievalUnit}\}$
- $sSubSpec(\texttt{LiteratureSystem}) = \{\texttt{ManagingUnit, LiteratureHost1}\}$
- $ic(\texttt{ManagingUnit}) = \{\texttt{searchFacility, tripPlanner, searchEngine}, loc\}$
- $ic(\texttt{RetrievalUnit}) = \{\texttt{searchFacility, tripPlanner, searchEngine}, loc\}$
- $ic(\texttt{LiteratureHost1}) = \{\texttt{searchF}, loc\}$

we get

1. $S^e_M = \{\texttt{LiteratureSystem, searchEng}, loc\}$
2. $S^i_M = \{\texttt{ManagingUnit.searchFacility}, \ldots, \texttt{ManagingUnit.}loc,$ $\texttt{RetrievalUnit.searchFacility}, \ldots, \texttt{RetrievalUnit.}loc,$ $\texttt{LiteratureHost1.searchF}, \texttt{LiteratureHost1.}loc,$ $\texttt{ManagingUnit, RetrievalUnit, LiteratureHost1}\}$
3. $S^\times_M = S^b_M \cup S^o_M = \{\texttt{LiteratureSystem}, \beta\} \cup \{\texttt{LiteratureSystem}\}$

as the set of module sorts and

$$\begin{aligned}\Omega_M = \{&\texttt{litSys} : \rightarrow \texttt{LiteratureSystem},\\ &\texttt{litSys.searchEng} : \rightarrow \texttt{searchEng}, \texttt{litSys}.loc : \rightarrow loc,\\ &\texttt{managingUnit.searchFacility} : \rightarrow \texttt{searchFacility}, \ldots,\\ &\texttt{managingUnit}.loc : \rightarrow loc, \texttt{managingUnit} : \rightarrow \texttt{ManagingUnit},\\ &\texttt{retrievalUnit.searchFacility} : \rightarrow \texttt{searchFacility}, \ldots,\\ &\texttt{retrievalUnit}.loc : \rightarrow loc, \texttt{retrievalUnit} : \rightarrow \texttt{RetrievalUnit},\\ &\texttt{litHost1.searchF} : \rightarrow \texttt{searchF}, \texttt{litHost1}.loc : \rightarrow loc,\\ &\texttt{litHost1} : \rightarrow \texttt{LiteratureHost1},\\ &\texttt{bod} : \rightarrow bod\}\end{aligned}$$

as the corresponding module instance symbols. □

The complete signatures corresponding to the last two examples are given in appendix A.2 and A.3, respectively.

Having introduced complex mobile and stationary subsystems formally as well as in our TROLL extension, we can now discuss how movement can be expressed. As a prerequisite, however, we first have to define action terms for complex subsystems and the initial values for the attributes of the location-reflecting interface objects or the basic modules corresponding to them.

Definition 6.48 (action terms for complex subsystems) The set of action terms of a complex subsystem ξ is defined by

1. If $ida_{su} \in subDecl(\xi), idb_{\epsilon,idb} \in sobj(su)$, and if α is an action term of the interface object idb belonging to the subsystem su, then $ida.idb.\alpha$ is an action term and $var_{in}(ida.idb.\alpha) = var_{in}(idb.\alpha)$ and $var_{out}(ida.idb.\alpha) = var_{out}(idb.\alpha)$.

2. Nothing else is an action term of ξ.

□

Example 6.49 (action terms for complex subsystems) The stationary subsystem `LiteratureSystem` of the sample specification contains the subsystem `RetrievalUnit`, formally denoted by su = `RetrievalUnit` $\in$ $subSpec$(`LiteratureSystem`). Due to the declaration `mobile retrievalUnit: RetrievalUnit`, $\texttt{retrievalUnit}_{\texttt{RetrievalUnit}} \in subDecl(\texttt{LiteratureSystem})$ is the corresponding instance declaration. According to example 6.29 we have $\texttt{searchEngine}_{\epsilon,\texttt{searchEngine}} \in sobj(\texttt{RetrievalUnit})$, and from example 6.30 using definition 6.16 we determine `searchKeywords` as an action term for this interface object. Hence,

```
retrievalUnit.searchEngine.searchKeywords(keywords, resultSet)
```

is an action term of `LiteratureSystem`. □

In the beginning of subsection 6.2.1, we mentioned that in a specification in "mobile TROLL" the location of any unit is given in relative terms by its position within the location hierarchy, and that at the formal level this information is provided by the value of the attribute of the location-reflecting interface objects (cf. page 103). From definition 6.35 we know that this attribute is read-only for stationary subsystems, but readable and writable for mobile ones—mirroring that only the latter can change location.

Of course, in order to be of any use read-only attributes have to be initialised properly:

Definition 6.50 (frame rule for mobile and stationary subsystems) The attribute λ of the location-reflecting basic module *loc* associated with an instance id_{su} of a mobile or stationary subsystem, or system specification is initialised with the corresponding instance symbol *id* as detailed in definition 6.42 or 6.43, respectively.

Read access to this attribute returns this value prefixed by a "/" and the value read in the same way from the attribute λ of the location-reflecting basic module associated with the enclosing (sub-) system. □

This definition can now for instance be used to express locations, as the following example illustrates:

Example 6.51 (initial values for location-reflecting basic modules) In the initial situation of the example specification, the following formulae hold:

- $\exists\, l_1 \triangleright \texttt{litSys}.loc.\lambda_{read}(l_1) \wedge l_1 =$ "`/DocumentRetrieval/litSys`",
- $\exists\, l_2 \triangleright \texttt{litHost1}.loc.\lambda_{read}(l_2)$
  $\wedge\, l_2 =$ "`/DocumentRetrieval/litSys/litHost1`",
- $\exists\, l_3 \triangleright \texttt{reception}.loc.\lambda_{read}(l_3)$
  $\wedge\, l_3 =$ "`/DocumentRetrieval/litSys/litHost1/reception`",
- $\exists\, l_4 \triangleright \texttt{retrievalUnit}.loc.\lambda_{read}(l_4)$
  $\wedge\, l_4 =$ "`/DocumentRetrieval/litSys/retrievalUnit`",

etc. □

In our "mobile TROLL," the following production has been added as another type of action rule:

Syntax

<*movement*> ::= **if** <*pFormula*> **moveTo** <*target*>
<*target*> ::= <*ident*> | { '/' <*ident*> } '/' <*ident*>

i.e. the syntax from page 96 is refined to

$$\begin{aligned} \textit{<actionRule>} \;::=\; & \textit{<valuation>} \mid \textit{<movement>} \mid \textit{<callTerm>} \mid \\ & \textit{<repetitiveRule>} \mid \textit{<conditionalRule>} \mid \\ & \textit{<actionRule>} \; \{ \text{`,'} \; \textit{<actionRule>} \; \} \end{aligned}$$

The *movement* primitive can be used for an object contained within a mobile entity to specify another (mobile or stationary) subsystem as the *target* into which the unit enclosing this object should migrate if the proposition can be evaluated to true. With the following definition we see that this primitive can be used only for objects contained within a mobile unit, as according to definition 6.35 only these are associated with a location-reflecting basic module containing a writable attribute λ.

Definition 6.52 (semantics of *movement* action rule) Let $(\alpha, \varphi, var_{local}, ax)$ with $var_{local} = \{v_1, \ldots, v_n\}$ be an operation definition of an object class c. The mapping $\mathcal{A}_{o,X}(ax)$ with an identity term $o \in T_{\Sigma_{Id}}(X)$ and a set of variables X from definition 6.21 is refined by:

1. $\mathcal{A}_{o,X}(\mathsf{if}\ p\ \mathsf{moveTo}\ t) \equiv (\mathsf{Y}(\mathcal{P}_{o,X}(p)) \Rightarrow \mathsf{X}(\odot loc.\lambda_{write}(t)))$, if t is a constant data term,

2. $\mathcal{A}_{o,X}(\mathsf{if}\ p\ \mathsf{moveTo}\ t) \equiv (\mathsf{Y}(\mathcal{P}_{o,X}(p)) \Rightarrow \exists\, v\ (v = t \wedge \mathsf{X}(\odot loc.\lambda_{write}(v))))$, if $t \in X$, and

3. $\mathcal{A}_{o,X}(\mathsf{if}\ p\ \mathsf{moveTo}\ t) \equiv (\mathsf{Y}(\mathcal{P}_{o,X}(p)) \Rightarrow$
   $\exists\, v\ (\mathsf{Y}(\triangleright o.t_{read}(v)) \wedge \mathsf{X}(\odot loc.\lambda_{write}(v))))$, if t is an attribute term and $t \notin X$

□

As according to definition 6.50 read access to the attribute λ of a location-reflecting basic module is performed in a recursive way up the subsystem hierarchy, the change to a specific λ is "propagated" to any subsystem included, and, hence, a mobile subsystem migrates with all the units it comprises.

A *movement* primitive can be specified for several objects directly contained within the same subsystem. However, it then has to be assured that always at most one of the corresponding propositions can be evaluated to true, as one subsystem may of course only move to one other location at a time.

Example 6.53 (operation definition—movement) Treating the mobile subsystem `RetrievalUnit` as a system specification, its operation definition

```
continueRetrieval()
  var newLiterature: set(litInfo),
      nxtLocBuf: string,
      nxxtTargetBuf: string
```

```
do
  litrInfo:= litrInfo + newLiterature,
  searchForKeywords(keywords, newLiterature),
  currentLoc:=nxtLocBuf,
  nextTarget:=nxxtTargetBuf,
  determineNextTarget(currentLoc, nxtLocBuf),
  determineNextTarget(nextTarget, nxxtTargetBuf),
  if (currentLoc # homeLoc) moveTo nextTarget
od;
```

gives rise to the following axiom for the `retriever` object (in the following abbreviated by r):

$$
\begin{array}{l}
r.(\odot r.\texttt{continueRetrieval()} \Rightarrow \\
\quad \exists\, \texttt{newLiterature, nxtLocBuf, nxxtTargetBuf} \\
\qquad \mathcal{A}_{r,X}(\texttt{litrInfo:= litrInfo + newLiterature}) \\
\qquad \wedge\, \mathcal{A}_{r,X}(\texttt{searchForKeywords(keywords, newLiterature)}) \\
\qquad \wedge\, \mathcal{A}_{r,X}(\texttt{currentLoc:=nxtLocBuf}) \wedge\ \mathcal{A}_{r,X}(\texttt{nextTarget:=nxxtTargetBuf}) \\
\qquad \wedge\, \mathcal{A}_{r,X}(\texttt{determineNextTarget(currentLoc, nxtLocBuf)}) \\
\qquad \wedge\, \mathcal{A}_{r,X}(\texttt{determineNextTarget(nextTarget, nxxtTargetBuf)}) \\
\qquad \wedge\, \mathcal{A}_{r,X}(\texttt{if (currentLoc \# homeLoc) moveTo nextTarget}) \\
)
\end{array}
$$

with $X = \{$newLiterature, nxtLocBuf, nxxtTargetBuf$\}$ and (equivalences are annotated by definitions applied)

1. $\mathcal{A}_{r,X}(\texttt{litrInfo:= litrInfo + newLiterature}) \overset{6.21,1.(b)}{\equiv}$
   $\exists v_{11}\, \mathsf{Y}(\mathcal{T}_{r,X}(v_{11} = \texttt{litrInfo + newLiterature})) \wedge \odot r\,.\,\texttt{litrInfo}_{write}(v_{11})$

   where

   $\mathcal{T}_{r,X}(v_{11} = \texttt{litrInfo + newLiterature}) \overset{6.15,4.(c)}{\equiv}$
   $\exists v_{12}, v_{13}(\mathcal{T}_{r,X}(v_{12} = \texttt{litrInfo})$
   $\qquad \wedge \mathcal{T}_{r,X}(v_{13} = \texttt{newLiterature})$
   $\qquad \wedge (v_{11} = v_{12} + v_{13})) \overset{6.15,2.}{\equiv}$
   $\exists v_{12}, v_{13}(\triangleright r\,.\,\texttt{litrInfo}_{read}(v_{12}) \wedge v_{13} = \texttt{newLiterature} \wedge (v_{11} = v_{12} + v_{13}))$,

2. $\mathcal{A}_{r,X}(\texttt{searchForKeywords(keywords, newLiterature)}) \overset{6.21,2.}{\equiv}$
   $\exists v_{21}, v_{22}\, \mathsf{Y}(\mathcal{T}_{r,X}(v_{21} = \texttt{keywords}))$
   $\qquad \wedge \mathsf{Y}(\mathcal{T}_{r,X}(v_{22} = \texttt{newLiterature}))$
   $\qquad \wedge \odot\, r\,.\,\texttt{searchForKeywords}(v_{21},\ v_{22})$

   where $\mathcal{T}_{r,X}(v_{21} = \texttt{keywords}) \overset{6.15,2.(b)}{\equiv} \triangleright r\,.\,\texttt{keywords}_{read}(v_{21})$
   and $\mathcal{T}_{r,X}(v_{22} = \texttt{newLiterature}) \overset{6.15,2.(a)}{\equiv} v_{22} = \texttt{newLiterature}$,

3. $\mathcal{A}_{r,X}(\texttt{currentLoc:= nxtLocBuf}) \overset{6.21,1.(b)}{\equiv}$
$\exists v_{31}\, \mathsf{Y}(\mathcal{T}_{r,X}(v_{31} = \texttt{nxtLocBuf})) \wedge \odot r\,.\,\texttt{currentLoc}_{write}(v_{31}) \overset{6.15,2.(a)}{\equiv}$
$\exists v_{31}\, \mathsf{Y}(v_{31} = \texttt{nxtLocBuf}) \wedge \odot r\,.\,\texttt{currentLoc}_{write}(v_{31})$,

4. $\mathcal{A}_{r,X}(\texttt{nextTarget:= nxxtTargetBuf}) \overset{6.21,1.(b)}{\equiv}$
$\exists v_{41}\, \mathsf{Y}(\mathcal{T}_{r,X}(v_{41} = \texttt{nxxtTargetBuf})) \wedge \odot r\,.\,\texttt{nextTarget}_{write}(v_{41}) \overset{6.15,2.(a)}{\equiv}$
$\exists v_{41}\, \mathsf{Y}(v_{41} = \texttt{nxxtTargetBuf}) \wedge \odot r\,.\,\texttt{nextTarget}_{write}(v_{41})$,

5. $\mathcal{A}_{r,X}(\texttt{determineNextTarget(currentLoc, nxtLocBuf)}) \overset{6.21,2.}{\equiv}$
$\exists v_{51}, v_{52}\, \mathsf{Y}(\mathcal{T}_{r,X}(v_{51} = \texttt{currentLoc}))$
$\wedge \mathsf{Y}(\mathcal{T}_{r,X}(v_{52} = \texttt{nxtLocBuf}))$
$\wedge \odot r\,.\,\texttt{determineNextTarget}(v_{51}, v_{52}) \overset{6.15,2.}{\equiv}$
$\exists v_{51}, v_{52}\, \mathsf{Y}(\triangleright r\,.\,\texttt{currentLoc}_{read}(v_{51}))$
$\wedge \mathsf{Y}(v_{52} = \texttt{nxtLocBuf})$
$\wedge \odot r\,.\,\texttt{determineNextTarget}(v_{51}, v_{52})$,

6. $\mathcal{A}_{r,X}(\texttt{determineNextTarget(nextTarget, nxxtTargetBuf)}) \overset{6.21,2.}{\equiv}$
$\exists v_{61}, v_{62}\, \mathsf{Y}(\mathcal{T}_{r,X}(v_{61} = \texttt{nextTarget}))$
$\wedge \mathsf{Y}(\mathcal{T}_{r,X}(v_{62} = \texttt{nxxtTargetBuf}))$
$\wedge \odot r\,.\,\texttt{determineNextTarget}(v_{61}, v_{62}) \overset{6.15,2.}{\equiv}$
$\exists v_{61}, v_{62}\, \mathsf{Y}(\triangleright r\,.\,\texttt{nextTarget}_{read}(v_{61}))$
$\wedge \mathsf{Y}(v_{62} = \texttt{nxxtTargetBuf})$
$\wedge \odot r\,.\,\texttt{determineNextTarget}(v_{61}, v_{62})$, and

7. $\mathcal{A}_{r,X}(\texttt{if (currentLoc \# homeLoc) moveTo nextTarget}) \overset{6.52,3.}{\equiv}$
$(\mathsf{Y}(\mathcal{P}_{r,X}(\texttt{currentLoc \# homeLoc})) \Rightarrow$
$\exists v_{71}\, (\mathsf{Y}(\triangleright r.\texttt{nextTarget}_{read}(v_{71})) \wedge \mathsf{X}(\odot loc.\lambda_{write}(v_{71}))))$
where

$\mathcal{P}_{r,X}(\texttt{currentLoc \# homeLoc}) \overset{6.13,1.}{\equiv}$
$\exists v_{72}\, \mathcal{T}_{r,X}(v_{72} = (\texttt{currentLoc \# homeLoc})) \wedge v_{72} = \mathsf{true} \overset{6.15,4.(c)}{\equiv}$
$\exists v_{72}\, \exists v_{73}, v_{74}\, (\mathcal{T}_{r,X}(v_{73} = \texttt{currentLoc}) \wedge \mathcal{T}_{r,X}(v_{74} = \texttt{homeLoc})$
$\wedge v_{72} = (v_{73} \neq v_{74}))$

with $\mathcal{T}_{r,X}(v_{73} = \texttt{currentLoc}) \overset{6.15,2.(b)}{\equiv} \triangleright r.\texttt{currentLoc}_{read}(v_{73})$
and $\mathcal{T}_{r,X}(v_{74} = \texttt{homeLoc}) \overset{6.15,2.(b)}{\equiv} \triangleright r.\texttt{homeLoc}_{read}(v_{74})$.

All in all, we thus obtain

$r.(\odot r.\texttt{continueRetrieval()} \Rightarrow$
$\exists\, \texttt{newLiterature, nxtLocBuf, nxxtTargetBuf}$
$\exists v_{11}\, \mathsf{Y}(\exists v_{12}, v_{13}(\triangleright r\,.\,\texttt{litrInfo}_{read}(v_{12})$
$\wedge v_{13} = \texttt{newLiterature} \wedge (v_{11} = v_{12} + v_{13})))$

$$
\begin{array}{l}
\qquad \wedge \odot r.\texttt{litrInfo}_{write}(v_{11}) \\
\wedge \exists v_{21}, v_{22}\, \mathsf{Y}(\triangleright r.\texttt{keywords}_{read}(v_{21})) \wedge \mathsf{Y}(v_{22} = \texttt{newLiterature}) \\
\qquad \wedge \odot r.\texttt{searchForKeywords}(v_{21},\ v_{22}) \\
\wedge \exists v_{31}\, \mathsf{Y}(v_{31} = \texttt{nxtLocBuf}) \wedge \odot r.\texttt{currentLoc}_{write}(v_{31}) \\
\wedge \exists v_{41}\, \mathsf{Y}(v_{41} = \texttt{nxxtTargetBuf}) \wedge \odot r.\texttt{nextTarget}_{write}(v_{41}) \\
\wedge \exists v_{51}, v_{52}\, \mathsf{Y}(\triangleright r.\texttt{currentLoc}_{read}(v_{51})) \wedge \mathsf{Y}(v_{52} = \texttt{nxtLocBuf}) \\
\qquad \wedge \odot r.\texttt{determineNextTarget}(v_{51},\ v_{52}) \\
\wedge \exists v_{61}, v_{62}\, \mathsf{Y}(\triangleright r.\texttt{nextTarget}_{read}(v_{61})) \wedge \mathsf{Y}(v_{62} = \texttt{nxxtTargetBuf}) \\
\qquad \wedge \odot r.\texttt{determineNextTarget}(v_{61},\ v_{62}) \\
\wedge (\mathsf{Y}(\exists v_{72}, v_{73}, v_{74} \\
\qquad (\triangleright r.\texttt{currentLoc}_{read}(v_{73}) \wedge \triangleright r.\texttt{homeLoc}_{read}(v_{74}) \wedge v_{72} = (v_{73} \neq v_{74}))) \Rightarrow \\
\quad \exists v_{71}(\mathsf{Y}(\triangleright r.\texttt{nextTarget}_{read}(v_{71})) \wedge \mathsf{X}(\odot loc.\lambda_{write}(v_{71}))))
\end{array}
$$

),

which can be simplified to

$$
\begin{array}{l}
r.(\odot r.\texttt{continueRetrieval()} \Rightarrow \\
\quad \exists\, \texttt{newLiterature, nxtLocBuf, nxxtTargetBuf} \\
\quad \exists v_{11}, v_{12}\, \mathsf{Y}(\triangleright r.\texttt{litrInfo}_{read}(v_{12}) \wedge v_{11} = v_{12} + \texttt{newLiterature}) \\
\qquad \wedge \odot r.\texttt{litrInfo}_{write}(v_{11}) \\
\quad \wedge \exists v_{21}\, \mathsf{Y}(\triangleright r.\texttt{keywords}_{read}(v_{21})) \wedge \odot r.\texttt{searchForKeywords}(v_{21},\ \texttt{newLiterature}) \\
\quad \wedge \odot r.\texttt{currentLoc}_{write}(\texttt{nxtLocBuf}) \\
\quad \wedge \exists v_{51}\, \mathsf{Y}(\triangleright r.\texttt{currentLoc}_{read}(v_{51})) \wedge \odot r.\texttt{determineNextTarget}(v_{51},\ \texttt{nxtLocBuf}) \\
\quad \wedge \odot r.\texttt{nextTarget}_{write}(\texttt{nxxtTargetBuf}) \\
\quad \wedge \exists v_{61}\, \mathsf{Y}(\triangleright r.\texttt{nextTarget}_{read}(v_{61})) \\
\qquad \wedge \odot r.\texttt{determineNextTarget}(v_{61},\ \texttt{nxxtTargetBuf}) \\
\quad \wedge (\mathsf{Y}(\exists v_{72}, v_{73}, v_{74} \\
\qquad (\triangleright r.\texttt{currentLoc}_{read}(v_{73}) \wedge \triangleright r.\texttt{homeLoc}_{read}(v_{74}) \wedge v_{72} = (v_{73} \neq v_{74}))) \Rightarrow \\
\quad \exists v_{71}(\mathsf{Y}(\triangleright r.\texttt{nextTarget}_{read}(v_{71})) \wedge \mathsf{X}(\odot loc.\lambda_{write}(v_{71}))))
\end{array}
$$

)

Similarly,

```
launchRetrieval(kw, tL, resultSet)
 onlyIf cnt(tL) > 1
do
 keywords:=kw,
 homeLoc:=tL(1),
 nextTarget:=tL(2), ...,
 if (true) moveTo tL(1),
od;
```

corresponds to

$$
\begin{array}{l}
r.(\odot r.\texttt{launchRetrieval(kw, tL, resultSet)} \Rightarrow \\
\quad \exists v_{11} \mathsf{Y}(v_{11} = \texttt{kw}) \wedge \odot r.\texttt{keywords}_{write}(v_{11}) \\
\quad \wedge \exists v_{21} \mathsf{Y}(v_{21} = \texttt{tL(1)}) \wedge \odot r.\texttt{homeLoc}_{write}(v_{21}) \\
\quad \wedge \exists v_{31} \mathsf{Y}(v_{31} = \texttt{tL(2)}) \wedge \odot r.\texttt{nextTarget}_{write}(v_{31}) \wedge \ldots \wedge \\
\quad \wedge (\mathsf{Y}(\exists v_{42}\ v_{42} = true) \Rightarrow \exists v_{41}(v_{41} = \texttt{tL(1)} \wedge \mathsf{X}(\odot loc.\lambda_{write}(v_{41}))))),
\end{array}
$$

plus

$r.(\triangleright r$`.launchRetrieval(kw, tL, resultSet)` $\Rightarrow$ (`cnt(tL)` $> 1))$

for the precondition, cf. appendix A.4 for details. □

As discussed in section 2.3, one point to keep in mind when developing compound systems containing both mobile and stationary components is the (relative) resource poverty of the former with respect to the latter. In our TROLL extension, this is reflected by the two productions <*mobiSpec*> and <*statSpec*>. Another issue to consider for this type of systems is that special attention should be paid to connectivity. Here, the <*linkDef*> and <*unlinkDef*> clauses in the **onEntry/onExit** and **onEntering/onExiting** sections can be used to identify actions of interface objects of the stationary or mobile subsystem, respectively. A <*linkDef*> clause specifies an action for which a link can be established if a matching action with an equal number of corresponding types of parameters is offered in the same way by the entering unit or the unit being entered. Similarly, an <*unlinkDef*> clause is used to determine which established connection will be released if a unit departs from another.

Syntax

<*linkDef*> ::= **link** <*ident*> '.' <*actionSignature*>
<*unlinkDef*> ::= **unlink** <*ident*> '.' <*actionSignature*>

The optional <*actionTerm*> in the **onEntry/onExit/onEntering/onExiting** sections can be used to name an action which should be triggered when the corresponding event occurs.

Definition 6.54 (attribute and action declarations refined) Let the specification of a mobile or stationary subsystem ψ be given.

1. $Attr(c)$ denotes the set of all attribute declarations for a given object class c or an interface object c.
2. $Act(c)$ denotes the set of all action declarations for a given specification of an object class c or an interface object c. In addition to those sets introduced in definition 6.3, the following sets are distinguished for interface objects c:
   (a) $Act_{enter}(c) \subseteq Act(c)$ denotes the set of actions corresponding to those listed in the <*linkDef*> clauses of the **onEntry** or **onEntering** section of a stationary or mobile subsystem ψ, respectively.
   (b) $Act_{exit}(c) \subseteq Act(c)$ denotes the set of actions corresponding to those listed in the <*unlinkDef*> clauses of the **onExit** or **onExiting** section of a stationary or mobile subsystem ψ, respectively.

□

Example 6.55 (action alphabet for interface objects) From the specification of the mobile subsystem `RetrievalUnit` the following sets of actions can be derived according to definition 6.54 (2.):

$$\begin{aligned} Act_{enter}(\texttt{searchEngine}) &= \{(\texttt{searchKeywords}_{\mathsf{set(string),set(litInfo)}}, \\ &\quad para_{in}(\texttt{searchKeywords}_{\mathsf{set(string),set(litInfo)}}), \\ &\quad para_{out}(\texttt{searchKeywords}_{\mathsf{set(string),set(litInfo)}}))\} \\ Act_{exit}(\texttt{searchEngine}) &= \{(\texttt{searchKeywords}_{\mathsf{set(string),set(litInfo)}}, \\ &\quad para_{in}(\texttt{searchKeywords}_{\mathsf{set(string),set(litInfo)}}), \\ &\quad para_{out}(\texttt{searchKeywords}_{\mathsf{set(string),set(litInfo)}}))\} \end{aligned}$$

with

$$\begin{aligned} para_{in}(\texttt{searchKeywords}_{\mathsf{set(string),set(litInfo)}}) &= \{1\}, \\ para_{out}(\texttt{searchKeywords}_{\mathsf{set(string),set(litInfo)}}) &= \{2\} \end{aligned}$$

□

Definition 6.56 (link set-up and release) Let the specification of a complex (sub-) system ξ be given. Let α denote ξ's local module sort, and $l \in T_{\Sigma_{M,\alpha}}$ be the local module term of the module signature derived from ξ. Let ψ_1 be a mobile or stationary subsystem of ξ, i.e. $\psi_1 \in subSpec(\xi)$, and let ψ_2 be another mobile subsystem within the entire system. Let $id^1_{su_1}$ and $id^2_{su_2}$ denote declarations of corresponding instances of these subsystems; in particular, we have $id^1_{su_1} \in subDecl(\xi)$. Finally, let $ids^1_{\epsilon,ids^1} \in sobj(\psi_1), ids^2_{\epsilon,ids^2} \in sobj(\psi_2), a^1_{s_{1,1},\ldots,s_{1,n}} \in Act_{enter}(ids^1)$ and $a^2_{s_{2,1},\ldots,s_{2,n}} \in Act_{enter}(ids^2)$ for $ids_1 \in ic(\psi_1)$ and $ids_2 \in ic(\psi_2)$.

1. If $s_{1,1} = s_{2,1}, \ldots, s_{1,n} = s_{2,n}$ and $a^1_{s_{1,1},\ldots,s_{1,n}} \in Act_{exit}(ids^1)$ or $a^2_{s_{2,1},\ldots,s_{2,n}} \in Act_{exit}(ids^2)$, then Φ_ξ contains the following axiom:

   $$l.((\exists\, x_1, x_2 \triangleright id^1.loc.\lambda_{read}(x_1) \wedge \triangleright id^2.loc.\lambda_{read}(x_2) \wedge x_1 = x_2) \Rightarrow (\odot id^1.ids^1.a^1(v_1, \ldots, v_n) \Leftrightarrow \odot id^2.ids^2.a^2(v_1, \ldots, v_n)))$$

2. If $s_{1,1} = s_{2,1}, \ldots, s_{1,n} = s_{2,n}$ and neither $a^1_{s_{1,1},\ldots,s_{1,n}} \in Act_{exit}(ids^1)$ nor $a^2_{s_{2,1},\ldots,s_{2,n}} \in Act_{exit}(ids^2)$, then Φ_ξ contains the following axiom:

   $$l.((\exists\, x_1, x_2\ P \triangleright id^1.loc.\lambda_{read}(x_1) \wedge \triangleright id^2.loc.\lambda_{read}(x_2) \wedge x_1 = x_2) \Rightarrow (\odot id^1.ids^1.a^1(v_1, \ldots, v_n) \Leftrightarrow \odot id^2.ids^2.a^2(v_1, \ldots, v_n)))$$

where $v_i \in X_{s_{1,i}} (= X_{s_{2,i}}), i \in \{1, \ldots, n\}$ with $v_i \neq v_j$ for $i \neq j$. □

Formulae according to definition 6.56 (1.) express that two subsystems may only communicate as long as they are at the same location. The second clause reflects the case that two subsystems established a connection while they simultaneously were at the same place sometime in the past, but maintained this link after departing.

Example 6.57 (link set-up and release) Considering the specification ξ of the stationary subsystem denoted by `LiteratureSystem`, we have α = `LiteratureSystem` and l = `litSys`. From example 6.47 we know ψ_1 = `LiteratureHost1` $\in$ $subSpec$(`LiteratureSystem`), with $id^1_{su_1}$ = $\texttt{litHost1}_{\texttt{LiteratureHost1}}$ being the corresponding instance declaration. ψ_2 = `RetrievalUnit` is another mobile subsystem within the entire system, and $id^2_{su_2}$ = $\texttt{retrievalUnit}_{\texttt{RetrievalUnit}}$ is its instance declaration. According to example 6.29 ids^2_{ϵ,ids^2} = $\texttt{searchEngine}_{\epsilon,\texttt{searchEngine}}$ $\in$ $sobj$(`RetrievalUnit`), and from example 6.55 we know $a^2_{s_{2,1},\, s_{2,2}}$ = $\texttt{searchKeywords}_{\mathsf{set(string)},\texttt{set(litInfo)}}$ $\in$ Act_{enter}(`searchEngine`). Analogously, we get ids^1_{ϵ,ids^1} = $\texttt{searchF}_{\epsilon,\texttt{searchF}}$ $\in$ $sobj$(`LiteratureHost1`) and $a^1_{s_{1,1},\, s_{1,2}}$ = $\texttt{searchKeyw}_{\mathsf{set(string)},\texttt{set(litInfo)}}$ $\in$ Act_{enter}(`searchF`), and as both $a^1_{s_{1,1},\, s_{1,2}}$ = $\texttt{searchKeyw}_{\mathsf{set(string)},\texttt{set(litInfo)}}$ $\in$ Act_{exit}(`searchF`) as well as $a^2_{s_{2,1},\, s_{2,2}}$ = $\texttt{searchKeywords}_{\mathsf{set(string)},\texttt{set(litInfo)}}$ $\in$ Act_{exit}(`searchEngine`) hold we obtain

$$
\begin{aligned}
&\texttt{litSys.}(\\
&\quad (\exists\, x_1, x_2 \triangleright \texttt{litHost1.}loc.\lambda_{read}(x_1) \wedge \triangleright\texttt{retrievalUnit.}loc.\lambda_{read}(x_2) \wedge\ x_1 = x_2)\\
&\quad \Rightarrow\\
&\quad (\odot\texttt{litHost1.searchF.searchKeyw(keyw, resultS)}\\
&\qquad \Leftrightarrow \odot\texttt{retrievalUnit.searchEngine.searchKeywords(keyw, resultS)}))
\end{aligned}
$$

as an axiom of $\Theta_{\texttt{LiteratureSystem}}$. □

6.3 Summary

The goal of this chapter was to provide language constructs in a formal language which, as a first step, allow to reflect the particular effects mobility has on information systems. We therefore extended the TROLL specification language with constructs to distinguish between the mobile and the stationary units in a compound systems containing both types of subsystems, to express the (conditional) migration of mobile parts, and to describe the set-up and release of communication links between those subsystems, thus allowing intermittent connectivity and mismatching resources between mobile and stationary components to be captured in the design of such complex systems. Starting with the basic language constructs from TROLL$_3$, we continued by building upon the subsystem concept introduced with TROLL$_M$, which allowed us to describe location—as a prerequisite to expressing mobility—in relative terms upon the subsystem hierarchy. Along the way, the formal syntax and semantics with regard to the theoretical framework underlying TROLL was defined, which considers the specification of a (complex) module $ModSpec = (\Theta, \Phi)$ to be a pair consisting of a module signature Θ and a set Φ of axioms constructed upon Θ.

In order to round off our work, in the following chapter we come back from these more theoretical issues to the application level and briefly summarise the design and implementation of a small system for accessing an online dictionary from mobile phones, which has been developed using our mobility-related extensions to TROLL.

Chapter 7

ODiMoD—An Online Dictionary for Mobile Devices

In recent years, more and more mobile devices with the possibility for accessing the Internet—like Personal Digital Assistants (PDAs), mobile phones, smartphones, etc.—have been developed and gained popularity. Due to their restricted resources, several means for adapting web content for the needs of these devices have been proposed and realised, which, among others, include the development of separate platforms [Rot02] (with WAP/WML and iMode being the most popular examples) and the transformation of existing web pages with regard to the capabilities of the target hardware via proxys [JWH97, NSN+97, Ali02]. In this chapter, we give an outline of a small system belonging to the latter (transforming) category which has been realised based on a specification in our extended "mobile Troll."

The emphasis of this case study was to pay attention to the mismatch between the (relative) resource poverty of mobile units and the potentially large amounts of data that may be provided by information systems, and to express this in the design of the system. The task was to specify and implement a system accepting queries to Internet services from mobile and stationary devices, forwarding these requests to the services and receiving the results from them, and transforming them with respect to the capabilities of the client. The implementation should be based on the Java 2 Micro Edition (J2ME), and should exemplarily be realised for mobile phones and the online dictionary of the Link Everything Online (LEO) service [LEO] of the Technical University Munich. Figure 7.2 shows a screenshot of the dictionary currently available at `http://dict.leo.org/`.

Figures 7.1 and 7.2 show screenshots of the dictionary currently available at `http://dict.leo.org/` which, when compared to those showing the developed application running on an emulator in figure 7.4 a) and b), respectively, also quite vividly illustrate the point of restricted resources for mobile devices: if only a small number of translations is available for a certain word, then displaying it on the mobile device is just as easy as with the stationary browser (cf. figures 7.1 and 7.4 a)); for a larger set of results, however, limitations of the display, processing speed, storage capabilities, etc. have to be considered. The restrictions on

Figure 7.1: The online dictionary of the Link Everything Online service [LEO] for a small result set

the output device are obvious when comparing figures 7.2 and 7.4 b). The model captures this issue by distinguishing between the mobile and stationary units using our TROLL extension and furthermore pays particular attention the limited amount of storage available on the mobile device, as we will see in the following.

As already mentioned in the task description, the system should support queries from mobile as well as stationary devices. With regard to Java chosen as the basis for the implementation—which provides an application programming interface for mobile devices as well as one for web server extensions, leading to applications referred to as *MIDlets* [Mah02] and *Servlets* [Hun98], respectively—the system's functionality for device-dependent transformations is part of a stationary subsystem `Servlet_sub` and the mobile device is represented by a mobile subsystem `Midlet_sub`, with corresponding instances `serv` and `mid`:

```
object system ODiMoD
  mobile Midlet_sub  ...  end_mobile;
  stationary HTML-Formular_sub  ...  end_stationary;
  stationary Servlet_sub  ...  end_stationary;
  stationaries HTML-Formular_sub;
  mobiles mid: Midlet_sub;
  stationaries form: HTML-Formular_sub;
  stationaries serv: Servlet_sub;
   ...
  behavior
    mid.data.getData(parameter, bytes, results) do
      serv.data.sendData(parameter, bytes, results)
```

Figure 7.2: The [LEO] online dictionary for a larger result set

```
    od;
    form.HTML-Page.getHTML(parameter, results) do
      serv.HTML-Page.sendHTML(parameter, results)
    od;
  end;

end.
```

When being accessed from a stationary device, the system basically passes through the information received from the Internet service in HTML, and the respective stationary subsystem is therefore referred to as `HTML-Formular_sub`, with the instance `form`. The `behavior` rule indicates that the `HTML-Formular_sub` contains a request-object `HTML-Page`, whose action `getHTML` is synchronised with the action `sendHTML` of an offer-object within the `Servlet_sub` of the same name. As further details regarding the subsystem `HTML-Formular_sub` can be neglected, we refrain from discussing it here and instead turn to the one modelling the mobile device.

Within the mobile subsystem `Midlet_sub`, the class `Midlet` represents the application to be developed. Its attribute `memorySize` models the upper limit for the amount of memory available on the mobile unit. It is initialised in the birth action of the class, the action `startMidlet`:

```
object class Midlet
  attributes
    memorySize: nat;
     ...
  actions
    * startMidlet(m : nat);
    invokeServlet(parameter: string,
                  bytes: nat,
                  ! results: set(string));
     ...
  behavior
    startMidlet(m) do
      memorySize:= m
    od;
     ...
end;
```

The action `invokeServlet` is used to initiate a request for information from the `Servlet_sub`. Its parameter `bytes` indicates the amount of storage currently available on the device, which due to other running applications etc. may be less then `memorySize`, and the string `parameter` encodes other information needed to answer the query. The result should be delivered as a set of strings. This action is then synchronised with the action `getData` of the `data` request-object within the `Midlet_sub`:

```
mobile Midlet_sub
  request_object data
    actions
     getData(parameter: string, bytes: nat, !results: set(string));
  end;

  object class Midlet  ...  end;
  objects mid: Midlet;

  behavior
    mid.invokeServlet(parameter, bytes, results) do
      data.getData(parameter, bytes, results)
    od;
  end;
end_mobile;
```

As indicated in the `behavior` rule of the object system, `getData` is in turn synchronised with the action `sendData` of an offer-object `data` within the instance `serv` of the stationary subsystem `Servlet_sub`, which finally propagates the call to the internal `server` object, an instance of the `Servlet` class:

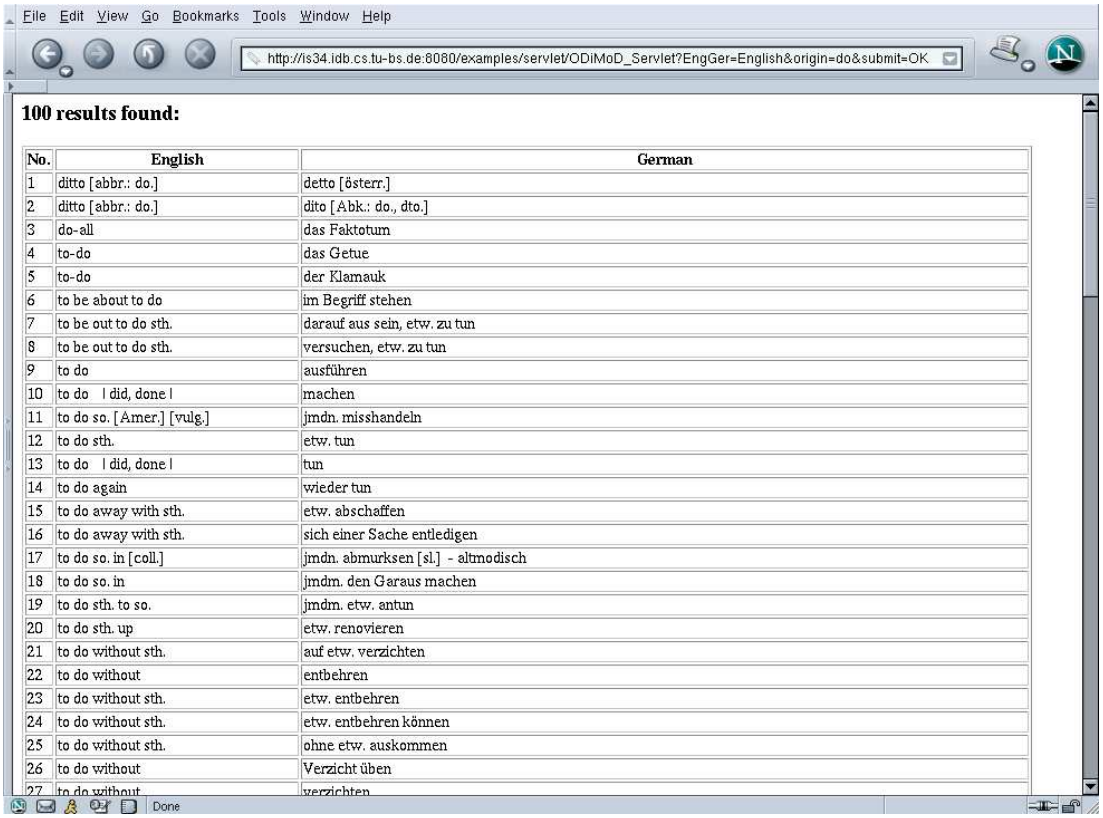
```
stationary Servlet_sub
  offer_object data
    actions
      sendData(parameter: string, bytes: nat, !results: set(string));
  end;
  offer_object HTML-Page
    actions
      sendHTML(parameter: string, !results HTMLPage);
  end;

  object class Servlet
    actions
      getExtractedData(parameter: string, !data: set(string));
      sendData(parameter: string, bytes: nat,
               size: nat, !data: set(string));
      sendHTML(parameter: string, !results: HTMLPage);
       ...
    behavior
      sendData(parameter, bytes, size, data)
        onlyif size <= bytes
      do
        getExtractedData(parameter, data)
      od;
       ...
  end;
```

```
  object class ExtractData
    actions
      extract(parameter: string, !data: set(string));
      determineResultSize(parameter: string, !resultSize: nat);
    ...
  end;

  objects server: Servlet;
  objects ex: ExtractData;

  behavior
    server.getExtractedData(parameter, data) do
      ex.extract(parameter, data)
    od;
    data.sendData(parameter, bytes, results)
      var s: nat;
    do
      ex.determineResultSize(parameter, s);
      server.sendData(parameter, bytes, s, results);
    od;
    HTML-Page.sendHTML(parameter, results) do
      server.sendHTML(parameter, results)
    od;
  end;
end_stationary;
```

The class `ExtractData` represents the point where the system could be adapted to access other Internet services. Its action `extract` should connect to the service and retrieve the desired information according to the `parameter`s given, and return them in a suitable format. The action `determineResultSize` should provide the size of this response, e.g. as the number of bytes.

The `behavior` rules of the stationary subsystem `Servlet_sub` and `Servlet` class describe the different ways in which queries from mobile and stationary devices should be treated: while a call from a stationary unit received via the action `sendHTML` of the `HTML-Page` offer-object is processed in a straightforward way, for a request obtained by means of the `sendData` action of the `data` offer-object the size of the result is determined and passed on to the `sendData` action of the `server` object, where further processing only takes place if this size is at most equal to the capacity available on the device, as indicated by the parameter `bytes`.

As already mentioned, the motivation for this chapter was to show the usability of our language constructs in the development of a real system, where the focus of this case study in turn was to pay attention to the mismatch between the (relative) resource poverty of mobile devices compared to stationary units, in



100 results found:

No.	English	German
1	ditto [abbr.: do.]	detto [österr.]
2	ditto [abbr.: do.]	dito [Abk.: do., dto.]
3	do-all	das Faktotum
4	to-do	das Getue
5	to-do	der Klamauk
6	to be about to do	im Begriff stehen
7	to be out to do sth.	darauf aus sein, etw. zu tun
8	to be out to do sth.	versuchen, etw. zu tun
9	to do	ausführen
10	to do I did, done I	machen
11	to do so. [Amer.] [vulg.]	jmdn. misshandeln
12	to do sth.	etw. tun
13	to do I did, done I	tun
14	to do again	wieder tun
15	to do away with sth.	etw. abschaffen
16	to do away with sth.	sich einer Sache entledigen
17	to do so. in [coll.]	jmdn. abmurksen [sl.] - altmodisch
18	to do so. in	jmdm. den Garaus machen
19	to do sth. to so.	jmdm. etw. antun
20	to do sth. up	etw. renovieren
21	to do without sth.	auf etw. verzichten
22	to do without	entbehren
23	to do without sth.	etw. entbehren
24	to do without sth.	etw. entbehren können
25	to do without sth.	ohne etw. auskommen
26	to do without	Verzicht üben
27	to do without	verzichten

Figure 7.3: Accessing the LEO dictionary via the stationary interface

particular with respect to the large amounts of data which information systems may provide, and to express this in the design of the system. Due to this focus, we refrain from going into the details of the program code here (which is available on demand), and merely illustrate the access to the system with screenshots from a browser on a stationary device (PC) and the MIDlet running on an emulator as well as on a real mobile device (Siemens S55 mobile phone) in figure 7.3 and 7.4, respectively.

a) b) c)

Figure 7.4: Accessing the LEO dictionary via the MIDlet running on a) and b) an emulator and c) a mobile phone

Chapter 8

Concluding Remarks

In this chapter, we summarise the main results of this thesis in the first section, discuss a similar—though less formal—approach in order to show the general applicability and significance of the developed language constructs in section 8.2, and finally indicate some directions for future research.

8.1 Summary

The goals of the thesis were to further investigate the impact of mobile hardware and software on information systems, to survey formal approaches for specifying mobile systems in computer science in general, and to derive from them those which meet the particularities of information systems with mobile units or to develop suitable language constructs in case the former should not be available.

Starting with a short overview on mobile entities in hardware and software systems, in chapter 2 we discussed the main effects of mobility on computer systems in general and on information systems in particular, structured according to the issues of restricted and varying resources, disconnections, location management, location dependent information, and security. We argued that the mismatching resources between the stationary and mobile parts of a compound system and the intermittent and changing connectivity should receive special consideration for information systems containing or being accessed by mobile units.

Following this, chapter 3 then surveyed specification techniques for mobile systems in computer science in general, yielding that none of the formal approaches discussed there provides adequate means to express the just mentioned particularities for information systems with mobile components.

We therefore set out to develop suitable constructs as an extension to the object-oriented specification language TROLL. As a brief introduction, chapter 4 sketched the historical development of the language and presented its established concepts and our new extensions informally and also by means of an example specification for mobile-agent-based information retrieval. We frequently came back to this example in chapter 5 and 6, where a thorough discussion of the TROLL concepts and the mobility-related extensions, its formal syntax, and semantics was given.

With the introduction of a module concept, the formal basis of TROLL moved from extended data signatures and the distributed temporal logic DTL interpreted over event structures to module signatures and the module distributed temporal logic MDTL, the latter also being interpreted over event structures. The hierarchical structuring of specifications offered in this way parallelled the approach to describe locations in a relative way frequently found in specification languages for mobile systems, and with this as the basis we provided the formal background for the concepts to distinguish between the mobile and stationary components of a compound system, the movement of entities, and the establishment and release of communication links.

In order to round off our work and to show the usability of our constructs in application development, chapter 7 summarised the design and implementation of a small system for accessing an online dictionary from mobile phones.

8.2 A Less Formal Approach

As already mentioned in the beginning of chapter 3, probably due to the general popularity of the Unified Modeling Language a number of mobility-related extensions to the UML have been suggested in recent years. In the following, we discuss the one given in [BKKW03] in some more detail, as it seems to be the one closest to our approach, emphasising the general significance of the abstractions presented in this thesis.

In [BKKW03], Baumeister et al. add concepts for describing locations, mobile objects, mobile locations, and movements to the UML by first introducing an abstract stereotype ≪spatial≫ dependent on the *Class* metaclass into the metamodel, and then requiring via an OCL constraint that any class bearing this stereotype or one of its specialisations has to provide an attribute "atLoc," this way also introducing an "atLoc" relation. The stereotypes ≪location≫ and ≪mobile≫ both specialise ≪spatial≫, the former allowing to designate classes whose instances are locations and the latter to label those whose instances can change their location. In order to also permit locations to be mobile, the stereotype ≪mobile location≫ is introduced as a specialisation of both ≪location≫ and ≪mobile≫. Similar to e.g. the Ambient Calculus [CG98], they, too, allow locations to be nested, which is expressed using the composition notation. Compared to our approach, the ≪location≫ stereotype therefore corresponds to the <*statSpec*> production and the ≪mobile≫ and ≪mobile location≫ together closely resemble the <*mobiSpec*> production. Accordingly, using these stereotypes a simplified version of our example specification for the mobile-agent-based information retrieval could be represented by the class diagram shown in figure 8.1.

In addition to the stereotypes applicable to classes, [BKKW03] also introduce a ≪move≫ and a ≪clone≫ stereotype usable for action states in activity diagrams in order to express migrations and clone actions, respectively, where the latter is provided merely as a notational convenience that could otherwise be ex-

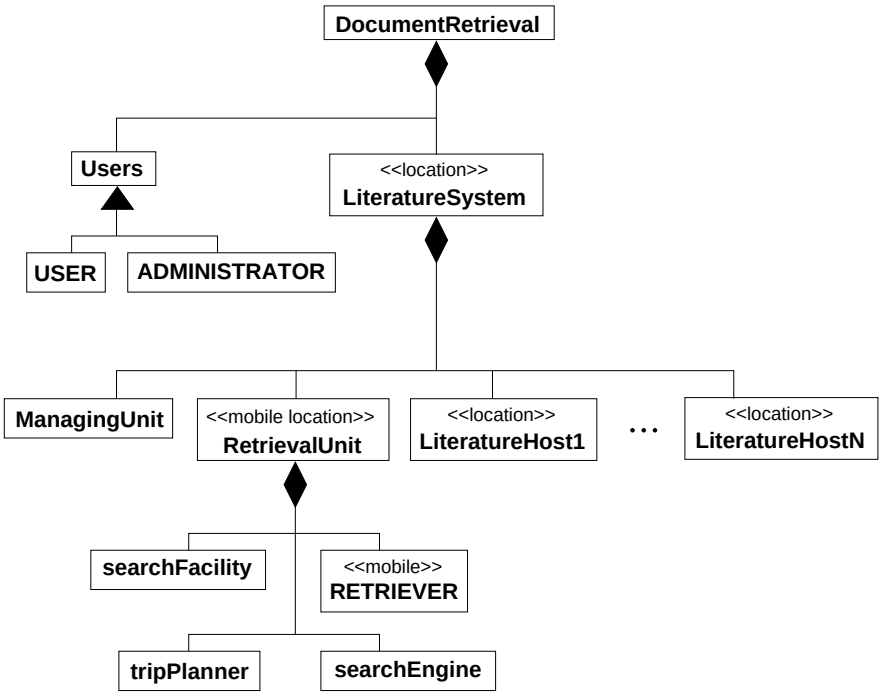


Figure 8.1: A simplified class diagram in the UML extension according to [BKKW03] for the mobile-agent-based information retrieval

pressed as a composition of a copy operation followed by a move operation. These actions have two additional attributes, the first one indicating who is performing the action and the second one denoting the location where the action takes place. Accordingly, two variants of activity diagrams are thus supported, one responsibility centred and one location centred. However, as nested locations cannot be represented using the former variant and also because our interest is more on mobility than in responsibilities, in the following we restrict our discussion to the latter approach.

The notation for the location centred variant uses containment of the boxes for mobile objects/locations in the boxes of other locations in order to show the "atLoc" relation, reusing an alternative notation for the UML's composition relation. Furthermore, [BKKW03] also allows action states to be drawn inside composite objects bearing the ≪location≫ label in order to indicate that the action is performed at the corresponding place. Entities affected by a movement (or clone) action are given using the object-flow notation. Limiting ourselves to the movement actions, figure 8.2 shows a location centred activity diagram corresponding to our example specification and the class diagram given in figure 8.1: The *retrievalUnit* as an instance of the compound class *RetrievalUnit* stereotyped as a ≪mobile location≫ is initially directly contained within the *litSys* instance of the *LiteratureSystem* at the same level as the *LiteratureHost*s, but upon the first—unconditional—migration its location changes to the one of *litHost1,* and provided that its current location then is not already the home location, it moves on to *litHost2* etc., all of these hosts being directly comprised by *litSys.*

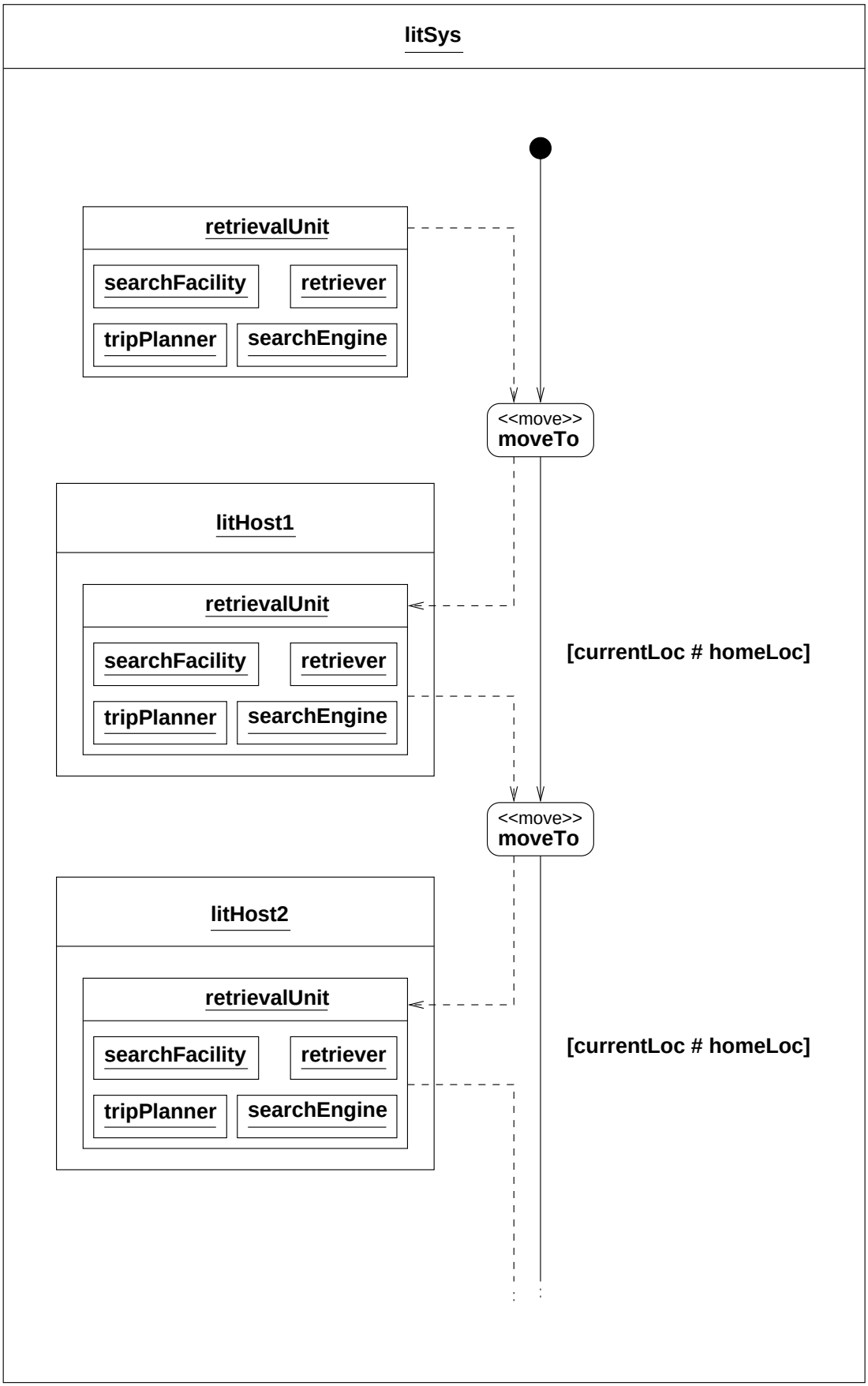


Figure 8.2: A simplified activity diagram in the UML extension according to [BKKW03] for the mobile-agent-based information retrieval

A part of our TROLL extension can therefore also already be expressed in this UML profile, and it should also be possible to come up with a suitable notation for describing the establishment and release of communication links, for example by means of appropriate OCL rules. In fact, a combination of a thus completed version of the UML profile and our TROLL extension could be used to provide the benefits like more intuitive comprehensibility and clarity of a graphical specification language with those like preciseness, unambiguity, and rigour of a formal one. This, however, will be a task for additional works, which finally brings us to the topic of future research.

8.3 Future Research

As already mentioned in the last section, one possible direction for future research with regard to this thesis and the development of mobile systems is to investigate an integration of a more intuitive graphical specification language like the UML with a formal one like our TROLL extension, providing mappings between the language constructs, determining restrictions from one on the other, etc. Approaches made combining an earlier version of TROLL with the Object Modelling Technique OMT as a precursor to UML in [DH97], and Troll$_M$ with the UML with a focus on modularisation and reuse in [EAN01] show the general feasibility.

In the other direction, in order to come closer to the implementation level of systems development, the extension of existing TROLL syntax and semantic checkers [Boh01, Gra01] for our language constructs and code generation from them for MIDlets and Servlets or agent platforms distinguishing between mobile and stationary agents like Grasshopper, Jade, Mole, etc. would provide useful tools. Similarly, adjusting MDTL to support model checking and revising TROLL and our extensions accordingly would also increase the usability for application development. So far, MDTL contains past and future-directed temporal operators as well as a concurrency operator, giving rise to the so-called *backtracking problem* that makes model checking undecidable [KF00]. Approaching this task, however, would imply a major revision of TROLL and its formal foundations.

Finally, with regard to the impact of mobility in particular on information systems and their design, language constructs allowing to specify the processing of transactions could be beneficially added to our extension. Here, picking up the concepts discussed in [Den96] with regard to refinements in object-oriented specifications could provide a good starting point.

Appendix A

Example Specifications

This chapter contains the example specification for the mobile-agent-based information retrieval as one continuous part. It also provides some of the signatures used in the main part of the thesis in full detail, together with explanations of how they are derived.

A.1 Mobile-Agent-Based Information Retrieval

```
object system DocumentRetrieval

  stationary Users
    request_object searchFacility
      actions
        doKeywordSearch(uname: string,
                        kw: set(string),
                        targets: list(string),
                        ! answer: set(litInfo));
    end;
    object class ADMINISTRATOR
       ...
      actions
        * becomeAdmin();
        registerUser(userName: string);
        deregisterUser(userName: string);
       ...
      // behaviour for
      // - registerUser: check credentials,
      //                 if ok report to managingUnit
      // - deregisterUser: remove from managingUnit
    end; // ADMINISTRATOR
```

```
object class USER
  attributes
    name: string;
    keywords: set(string);
    targetList: list(string);
   ...
  actions
    * becomeAUser(n: string);
    setKeywords(keyw: set(string));
    setTargets(targets: list(string));
    subscribe();
    unsubscribe();
    searchByKeywords(! resultSet: set(litInfo));
    searchByKeywords(k: set(string),
                     t: list(string),
                     ! r: set(litInfo)) hidden;
   ...
  behavior
    becomeAUser(n) do name:= n od;
    setKeywords(keyw) do keywords:= keyw od;
    setTargets(targets) do targetList:= targets od;
    searchByKeywords(r) do
      searchByKeywords(keywords, targetList, r)
    od;
   ...
end; // USER

objects admin: ADMINISTRATOR;
objects user(name: string): USER;

behavior
  user(name).subscribe() do
    admin.registerUser(name);
  od;
  user(name).unsubscribe() do
    admin.deregisterUser();
  od;
  // communication with interface objects:
  user(name).searchByKeywords(keywords, targetList, resultSet)
  do
    searchFacility.doKeywordSearc(name,
                                  keywords,
                                  targetList,
                                  resultSet);
  od;
```

```
    // communication regarding admin omitted; would also
    // require additional interface objects in managingUnit
  end;
end_stationary; // Users


stationary LiteratureSystem
  offer_object searchEng
    actions
      searchKeywords(userName: string,
                     keywords: set(string),
                     homeAndFirstTarget: list(string),
                     ! answer: set(litInfo))
  end;

  stationary ManagingUnit
    offer_object searchFacility
      actions
        searchKeywords(un: string,
                       kw: set(string),
                       tL: list(string),
                       ! answer: set(litInfo));
    end;
    offer_object tripPlanner
      actions
        determineNextTarget(someLocation: string,
                            ! nextLocation: string);
    end;
    request_object searchEngine
      actions
        searchForKeywords(kw: set(string),
                          tL: list(string),
                          ! answer: set(litInfo));
    end;


    object class PLANNER
      attributes
        targets: list(string);
        registeredUsers: list(string);
      actions
        checkRegistration(userName: string, ! isRegistered: bool);
        checkAvailability(! isAvailable);
        getNextTarget(someLoc: string, ! nextLoc: string);
        setTargetList(tList: list(string));
```

```
    behavior
      checkRegistration(userName, isRegistered) do
      // isRegistered = userName in registeredUsers
      od;
      checkAvailability(isAvailable) do
      // check if retrievalUnit is available, e.g.
      // if retrievalUnit.homeLoc = retrievalUnit.currentLoc;
      // would require additional interface objects and
      // communications - here ommited
      od;
      getNextTarget(someLoc, nextLoc) do
      // set nextLoc to the value following someLoc in targets,
      // treating targets as a circular list;
      // set nextLoc to targets(2) if someLoc = ‘‘none’’
      // (first migration from stationary to stationary)
      od;
      setTargetList(tList) do targets:= tList od;
    end;
  end; // PLANNER

  objects planner: PLANNER;
  behavior
    tripPlanner.determineNextTarget(someLocation, nextLocation) do
      planner.getNextTarget(someLocation, nextLocation);
    od;
    searchFacility.searchKeywords(un, kw, tL, answer)
      var isRegistered: bool,  isAvailable: bool;
    do
      checkRegistration(un, isRegistered),
      checkAvailability(isAvailable),
      if (isRegistered and isAvailable) then
        planner.setTargetList(tL),
        searchEngine.searchForKeywords(kw, tL, answer)
      fi;
    od;
  end;
end_stationary; // ManagingUnit

mobile RetrievalUnit
  offer_object searchFacility
    actions
      searchKeywords(keywords: set(string),
                     homeAndFirstTarget: list(string),
                     ! resultSet: set(litInfo));
  end;
```

```
request_object tripPlanner
  actions
    getNextTarget(someLoc: string, ! nextLoc: string);
end;

request_object searchEngine
  actions
    searchKeywords(keywords: set(string),
                   ! resultSet: set(litInfo));
end;

object class RETRIEVER
  attributes
    homeLoc: string;
    currentLoc: string initialized ‘‘none’’;
    nextTarget: string;
    keywords: set(string);
    litrInfo: set(litInfo);
  actions
    * becomeRetriever();
    launchRetrieval(kw: set(string),
                    tL: list(string),
                    ! resultSet: set(litInfo));
    searchForKeywords(keywords: set(string),
                      ! resultSet: set(litInfo));
    determineNextTarget(someLoc: string,
                        ! nextLoc: string);
    continueRetrieval();
    + cease();
  behavior
    launchRetrieval(kw, tL, resultSet)
    // tL = targetList containing two entries:
    //   tL(1) = start stationary,
    //   tL(2) = first target from start stationary
      onlyIf cnt(tL) > 1
    do
      keywords:=kw,
      homeLoc:=tL(1),
      nextTarget:=tL(2),
      if (true) moveTo tL(1),
       ...
    od;
    searchForKeywords(keywords, resultSet) do
    // no functionality, required only for module-global
    // coupling with offer-object (see behavior spec below)
```

```
    determineNextTarget(someLoc, nextLoc) do
    // no functionality, required only for module-global
    // coupling with request-object
    od;
    continueRetrieval()
      var newLiterature: set(litInfo),
          nxtLocBuf: string,
          // two steps ahead = next target _after_ migration:
          nxxtTargetBuf: string
    do
      litrInfo:= litrInfo + newLiterature,
      searchForKeywords(keywords, newLiterature),
      // due to the interaction with/from an interface object,
      // litrInfo.new will be litrInfo.old + litrInfo obtained
      // locally --- but only from the next step on
      currentLoc:=nxtLocBuf,
      nextTarget:=nxxtTargetBuf,
      determineNextTarget(currentLoc, nxtLocBuf),
      determineNextTarget(nextTarget, nxxtTargetBuf),
      if (currentLoc # homeLoc) moveTo nextTarget
    od;
end; // RETRIEVER

objects retriever: RETRIEVER;

behavior
  retriever.searchForKeywords(keywords, resultSet) do
    searchEngine.searchKeywords(keywords, resultSet);
  od;
  retriever.determineNextTarget(someLoc, nextLoc) do
    tripPlanner.getNextTarget(someLoc, nextLoc)
  od;
  searchFacility.searchKeywords(keywords,
                                homeAndFirstTarget,
                                resultSet)        do
    retriever.launchRetrieval(keywords,
                              homeAndFirstTarget,
                              resultSet);
  od;
end;
onEntering
  link searchEngine.searchKeywords(set(string),
                                   ! set(litInfo));
  retriever.continueRetrieval();
end
```

```
  onExiting
    unlink searchEngine.searchKeywords(set(string),
                                       ! set(litInfo));
  end
end_mobile; // RetrievalUnit


stationary LiteratureHost1
  // and likewise further n LiteratureHost_n_ .
  offer_object searchF
    actions
      searchKeyw(keyw: set(string),
                 ! resultS: set(litInfo));
  end;

  stationary Reception
    offer_object searchFacility
      actions
        searchByKeywords(keywords: set(string),
                         ! resultSet: set(litInfo));
    end;

    object class FETCHINGUNIT
      attributes
        // something that can be searched through using keywords
      actions
        doKeywordSearch(keywords: set(string),
                        ! resultSet: set(litInfo));
      behavior
        doKeywordSearch(keywords, resultSet)
          ...
        do
        // search through local information using keywords,
        // return results in resultSet
        od;
    end; // FETCHINGUNIT
    objects fetchingUnit: FETCHINGUNIT;
    behavior
      searchFacility.searchByKeywords(keywords, resultSet) do
        fetchingUnit.doKeywordSearch(keywords, resultSet);
      od;
    end;
  end_stationary; // Reception

  stationaries reception: Reception;
```

```
    behavior
      searchF.searchKeyw(keyw, resultS) do
        reception.searchFacility.searchByKeywords(keyw, resultS);
      od;
    end;
    onEntry
      link searchF.searchKeyw(set(string), ! set(litInfo));
    end
    onExit
      unlink searchF.searchKeyw(set(string), ! set(litInfo));
    end
  end_stationary; // LiteratureHost1

  stationaries litHost1: LiteratureHost1;

  stationaries managingUnit: ManagingUnit;

  mobiles retrievalUnit: RetrievalUnit;

  behavior
    searchEng.searchKeywords(userName,
                             keywords,
                             homeAndFirstTarget,
                             answer)
    do
      managingUnit.searchFacility.searchKeywords(userName,
                                                 keywords,
                                                 homeAndFirstTarget,
                                                 answer)
    od;
    retrievalUnit.tripPlanner.getNextTarget(someLocation,
                                            nextLocation)
    do
      managingUnit.tripPlanner.determineNextTarget(someLocation,
                                                   nextLocation)
    od;
    managingUnit.searchEngine.searchForKeywords(kw, tL, answer) do
      retrievalUnit.searchFacility.searchKeywords(kw, tL, answer)
    od;
  end;
end_stationary; // LiteratureSystem

stationaries usrs: Users;

stationaries litSys: LiteratureSystem;
```

```
  behavior
    usrs.searchFacility.doKeywordSearch(userName,
                                        keywords,
                                        homeAndFirstTarget,
                                        resultSet)
    do
      litSys.searchEng.searchKeywords(userName,
                                      keywords,
                                      homeAndFirstTarget,
                                      resultSet)

    od;
    // communication regarding managingUnit - admin ommited
  end;

  data type litInfo = record(title: string, authors: list(string),
                             year: nat, type: string);
end. // DocumentRetrieval
```

A.2 LiteratureHost1

1. For the specification ω of the stationary subsystem denoted by `LiteratureHost1` ($= su$) the local module sort according to definition 6.39 (1.) is $\alpha = su =$ `LiteratureHost1`. With

   - $ic($`LiteratureHost1`$) = \{$`searchF`$, loc\}$
     ($ic(\omega) = \{ids_1, \ldots, ids_n\}$, set of all identifiers of ω's interface objects with local module sorts $\alpha_1, \ldots, \alpha_n$, where (again by definition 6.39 (2.)) $\alpha_i = ids_i$)
   - $subSpec($`LiteratureHost1`$) = \{$`Reception`$\}$
     ($subSpec(\omega) = \{\omega_1, \ldots, \omega_m\}$)
   - $mSubSpec($`LiteratureHost1`$) = \{\}$
   - $sSubSpec($`LiteratureHost1`$) = \{$`Reception`$\}$
     ($subSpec \supseteq sSubSpec = \{\omega_{l_1}, \ldots, \omega_{l_h}\}$)
   - $ic($`Reception`$) = \{$`searchFacility`$, loc\}$
     ($ic(\omega_i) = \{ids_{i,1}, \ldots, ids_{i,k_i}\} \overset{6.39\ (2.)}{=} \{\gamma_{i,1}, \ldots, \gamma_{i,k_i}\}$)

   we therefore obtain

   (a) $S_M^e = \{$`LiteratureHost1, searchF`$, loc\}$

   (b) $S_M^i = \{$`Reception.searchFacility, Reception.`loc`, Reception`$\}$

   (c) $S_M^\times = \{$`LiteratureHost1`$, \beta\}$

   according to definition 6.45. Continuing at the formal level, we have

   $\Theta_{\texttt{LiteratureHost1}}$

$$
\begin{aligned}
&\overset{5.48}{=} && (\Sigma_K, \Sigma_{K_{bod}}, Imp, Exp)\\
&\overset{5.45}{=} && ((\Sigma, \Omega), \Sigma_{K_{bod}}, Imp, Exp)\\
&\overset{5.38}{=} && ((S_D \cup S_O^i \cup S_O^{at} \cup S_O^{ac} \cup S_M, \Omega_D \cup \Omega_O^i \cup \Omega_O^{at} \cup \Omega_O^{ac} \cup \Omega_M),\\
& && \Sigma_{K_{bod}}, Imp, Exp)\\
& && \qquad\qquad\qquad\qquad\qquad\qquad \overbrace{\qquad\qquad\qquad\qquad}^{S_M^+}\\
& && \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \overbrace{\qquad\qquad}^{S_M^\times}\\
&\overset{5.45}{=} && ((S_D \cup S_O^i \cup S_O^{at} \cup S_O^{ac} \cup S_M^e \cup S_M^i \cup \overbrace{S_M^b \cup S_M^o}^{},\\
& && \Omega_D \cup \Omega_O^i \cup \Omega_O^{at} \cup \Omega_O^{ac} \cup \Omega_M),\\
& && \Sigma_{K_{bod}}, Imp, Exp)\\
&\overset{\substack{6.39+\\ 6.45(1.)}}{=} && ((S_D \cup S_O^i \cup S_O^{at} \cup S_O^{ac} \cup\\
& && \{\texttt{LiteratureHost1, searchF}, loc\} \cup S_M^i \cup S_M^b \cup S_M^o,\\
& && \Omega_D \cup \Omega_O^i \cup \Omega_O^{at} \cup \Omega_O^{ac} \cup \Omega_M),\\
& && \Sigma_{K_{bod}}, Imp, Exp)
\end{aligned}
$$

$$\overset{6.45(2.)}{=} ((S_D \cup$$
$$S^i_O \cup$$
$$S^{at}_O \cup$$
$$S^{ac}_O \cup$$
$$\{\texttt{LiteratureHost1, searchF}, loc\} \cup$$
$$\{\texttt{Reception.searchFacility, Reception.}loc, \texttt{Reception}\} \cup$$
$$S^b_M \cup$$
$$S^o_M,$$
$$\Omega_D \cup \Omega^i_O \cup \Omega^{at}_O \cup \Omega^{ac}_O \cup \Omega_M),$$
$$\Sigma_{K_{bod}}, Imp, Exp)$$

$$\overset{6.45(3.)}{=} ((S_D \cup$$
$$S^i_O \cup$$
$$S^{at}_O \cup$$
$$S^{ac}_O \cup$$
$$\{\texttt{LiteratureHost1, searchF}, loc\} \cup$$
$$\{\texttt{Reception.searchFacility, Reception.}loc, \texttt{Reception}\} \cup$$
$$\{\texttt{LiteratureHost1}, \beta\}$$
(with $\beta = bod$ due to def. 5.48 ("local modul sort *bod*)")
$$\cup\{\texttt{LiteratureHost1}\},$$
$$\Omega_D \cup \Omega^i_O \cup \Omega^{at}_O \cup \Omega^{ac}_O \cup \Omega_M),$$
$$\Sigma_{K_{bod}}, Imp, Exp)$$

2. According to definition 5.38, kernel signatures consist of extended data signatures augmented by special module sorts and module instance symbols. Definition 6.45 determines these module sorts and has already been used above for this purpose. Definition 6.42 explains how to obtain the corresponding instance symbols, which in this example leads to :

$$\begin{aligned}\Omega_M \supset \{&\texttt{litHost1} : \rightarrow \texttt{LiteratureHost1},\\ &\texttt{litHost1.searchF} : \rightarrow \texttt{searchF},\\ &\texttt{litHost1.}loc : \rightarrow loc\}.\end{aligned}$$

Regarding the sorts of the imported module sorts, it seems reasonable to simply take over the corresponding instance symbols given by the respective export signatures, here:

$$\begin{aligned}\Omega_M \supset \{&\texttt{reception.searchFacility} : \rightarrow \texttt{searchFacility},\\ &\texttt{reception.}loc : \rightarrow loc, \texttt{reception} : \rightarrow \texttt{Reception}\}\end{aligned}$$

and overall:

$$\begin{aligned}\Omega_M \;=\; \{&\texttt{litHost1} : \rightarrow \texttt{LiteratureHost1},\\ &\texttt{litHost1.searchF} : \rightarrow \texttt{searchF},\\ &\texttt{litHost1.}\mathit{loc} : \rightarrow \mathit{loc},\\ &\texttt{reception.searchFacility} : \rightarrow \texttt{searchFacility},\\ &\texttt{reception.}\mathit{loc} : \rightarrow \mathit{loc},\\ &\texttt{reception} : \rightarrow \texttt{Reception}\\ &\texttt{bod} : \rightarrow \mathit{bod}\}.\end{aligned}$$

3. In order to be able to formalise the signature part of a specification of a complex subsystem completely, we finally need to characterise the structure of the extended data signature itself. For the extended data signature of a complex subsystem as this, first the class signatures corresponding to the interface objects have to be provided, and in a second step the matching extended data signatures can then be constructed from them (cf. e.g. example 6.41):

   (a) class signatures: The stationary subsystem `LiteratureHost1` contains only the offer-object `searchF`, with class signature

$$\begin{aligned}\Sigma_C(\texttt{searchF}) \;&=\; (S_O, I, AT, AC)\\ &=\; (\{\texttt{searchF}\},\\ &\quad\{\texttt{searchF} : \rightarrow \texttt{searchF}\},\\ &\quad\{\},\\ &\quad\{\texttt{searchByKeywords} :\\ &\quad\mathsf{set}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchF}\}),\end{aligned}$$

   (cf. example 6.44 and [Eck01, p. 131]), used to establish a connection to `searchFacility` in `Reception`, and the implied location-reflecting interface object *loc*, whose class signature according to definition 6.35 is given by

$$\begin{aligned}\Sigma_C(\mathit{loc}) \;&=\; (S_O, I, AT, AC)\\ &=\; (\{\mathit{loc}\},\\ &\quad\{\mathit{loc} : \rightarrow \mathit{loc}\},\\ &\quad\{\lambda_{\mathsf{string}} : \mathit{loc} \rightarrow \mathsf{string}\},\\ &\quad\{\lambda_{\mathit{read}} : \mathsf{string} \rightarrow \mathit{loc}\}).\end{aligned}$$

   (b) extended data signatures: Building upon the former we obtain

$$\begin{aligned}S^{i}_{O_{io}} \;&=\; \{\texttt{searchF}^{i}, \mathit{loc}^{i}\}\\ S^{at}_{O_{io}} \;&=\; \{\texttt{searchF}^{at}, \mathit{loc}^{at}\}\\ S^{ac}_{O_{io}} \;&=\; \{\texttt{searchF}^{ac}, \mathit{loc}^{ac}\}\end{aligned}$$

$$
\begin{array}{lcl}
\Omega^{i}_{O_{io}} & = & \{\texttt{searchF} : \rightarrow \texttt{searchF}^{i}, loc : \rightarrow loc^{i}\} \\
\Omega^{at}_{O_{io}} & = & \{\lambda_{\mathsf{string}} : loc^{i} \times loc^{at} \rightarrow \mathsf{string}\} \\
\Omega^{ac}_{O_{io}} & = & \{\texttt{searchKeyw} : \\
 & & \texttt{searchF}^{i} \times \mathsf{set}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchF}^{ac}, \\
 & & \lambda_{read} : loc^{i} \times \mathsf{string} \rightarrow loc^{ac}\}
\end{array}
$$

in analogy to example 6.44 and [Eck01, p. 131] and [Eck01, Bsp. 3.2].

4. Together with the extended data signatures for the body module signature and the import module signatures derived from the interface objects of the internal subsystems this forms the extended data signature of the complex subsystem. The class signature for the interface objects are obtained as explained in definition 6.28, 6.31, and 6.33, but this time the sorts and operation symbols are prefixed with the local module sort of the subsystem comprising the corresponding interface object (using definition 6.39 (1.)) or its instance symbol, respectively. This is due to the problem that identifiers for interface objects have to be unique only within one subsystem, but not throughout the entire system; the same identifier can be used in several subsystems.

   (a) extended data signatures for the body module signature: Because of definition 5.48 "$\Sigma_{K_{bod}}$ is a body module signature with local module sort *bod*" and 5.43 "A basic module signature $\Theta = (\Sigma_K, Exp)$ is a body module signature if $Exp = \{(\Sigma, id)\}$ and S_M is a singleton" we get

$$
\begin{array}{lcl}
\Sigma_{K_{bod}} & = & (\Sigma_{bod}, \{\Sigma, id\}) \\
 & \overset{5.43}{=} & ((S, \Omega), \{\Sigma, id\}) \\
 & \overset{5.38}{=} & ((S_D \cup S^{i}_{O} \cup S^{at}_{O} \cup S^{ac}_{O} \cup S^{e}_{M} \cup S^{+}_{M}, \Omega), \{\Sigma, id\}) \\
 & & \mid 5.48\text{: local module sort } bod \\
 & & \mid +\,5.38\text{: } S^{e}_{M} \cap S^{+}_{M} = \{\alpha\} \text{ local module sort,} \\
 & & \mid S_M = S^{e}_{M} \cup S^{+}_{M} \\
 & & \mid +\,5.43\text{: } S_M \text{ is a singleton} \\
 & = & ((S_D \cup S^{i}_{O} \cup S^{at}_{O} \cup S^{ac}_{O} \cup \{bod\}, \\
 & & \Omega_D \cup \Omega^{i}_{O} \cup \Omega^{at}_{O} \cup \Omega^{ac}_{O} \cup \Omega_M), \\
 & & \{\Sigma, id\}) \\
 & & \mid \text{explanation following def. 6.45: "a body module} \\
 & & \mid \text{signature may only provide information on data} \\
 & & \mid \text{type specifications. It therefore follows that} \\
 & & \mid \text{the corresponding signature has to be a simple} \\
 & & \mid \text{data signature, of the form } \Sigma_\beta = (S_D, \Omega_D)\text{."} \\
 & = & ((S_D \cup \{\} \cup \{\} \cup \{\} \cup \{bod\},
\end{array}
$$

$$\begin{array}{rcl} & & \Omega_D \cup \{\} \cup \{\} \cup \{\} \cup \{bod : \rightarrow bod\}), \\ & & \{\Sigma, id\}) \\ & = & ((S_D \cup \{bod\}, \Omega_D \cup \{bod : \rightarrow bod\}), \{\Sigma, id\}). \end{array}$$

(b) import module signatures derived from the interface objects of the internal subsystems [...]: `LiteratureHost1` has only one internal subsystem, `Reception`, whose set of interface objects is

$$ic(\texttt{Reception}) = \{\texttt{searchFacility}, loc\}$$

with class signatures
$\Sigma_C(\texttt{searchFacility})$

$$\begin{array}{rcl} & = & (S_O, I, AT, AC) \\ & = & (\{\texttt{searchFacility}\}, \\ & & \{\texttt{searchFacility} : \rightarrow \texttt{searchFacility}\}, \\ & & \{\}, \\ & & \{\texttt{searchByKeywords} : \\ & & \mathsf{set}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchFacility}\}), \\ \Sigma_C(loc) & = & (S_O, I, AT, AC) \\ & = & (\{loc\}, \\ & & \{loc : \rightarrow loc\}, \\ & & \{\lambda_{\mathsf{string}} : loc \rightarrow \mathsf{string}\}, \\ & & \{\lambda_{read} : \mathsf{string} \rightarrow loc\}), \end{array}$$

and corresponding extended data signatures, where this time the sorts and operation symbols are prefixed with the local module sort of the subsystem comprising the corresponding interface object (using definition 6.39 (1.)) or its instance symbol, respectively:

$$\begin{array}{rcl} S^i_{O_{Rec_1}} & = & \{\texttt{Reception.searchFacility}^i\}, \\ S^{at}_{O_{Rec_1}} & = & \{\texttt{Reception.searchFacility}^{at}\}, \\ S^{ac}_{O_{Rec_1}} & = & \{\texttt{Reception.searchFacility}^{ac}\}, \\ \Omega^i_{O_{Rec_1}} & = & \{\texttt{reception.searchFacility} : \\ & & \rightarrow \texttt{Reception.searchFacility}^i\} \\ \Omega^{at}_{O_{Rec_1}} & = & \{\} \\ \Omega^{ac}_{O_{Rec_1}} & = & \{\texttt{reception.searchByKeywords} : \\ & & \texttt{Reception.searchFacility}^i \times \mathsf{set}(\mathsf{string}) \\ & & \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{Reception.searchFacility}^{ac}\}, \\ S^i_{O_{Rec_2}} & = & \{\texttt{Reception.}loc^i\}, \\ S^{at}_{O_{Rec_2}} & = & \{\texttt{Reception.}loc^{at}\}, \\ S^{ac}_{O_{Rec_2}} & = & \{\texttt{Reception.}loc^{ac}\}, \end{array}$$

$$
\begin{aligned}
\Omega^{i}_{O_{Rec_2}} &= \{\texttt{reception}.loc : \rightarrow \texttt{Reception}.loc^{i}\} \\
\Omega^{at}_{O_{Rec_2}} &= \{\texttt{reception}.\lambda_{\mathsf{string}} : \\
&\quad \texttt{Reception}.loc^{i} \times \texttt{Reception}.loc^{at} \rightarrow \mathsf{string}\} \\
\Omega^{ac}_{O_{Rec_2}} &= \{\texttt{reception}.\lambda_{read} : \\
&\quad \texttt{Reception}.loc^{i} \times \mathsf{string} \rightarrow \texttt{Reception}.loc^{ac}\}
\end{aligned}
$$

from $\Sigma_{K_{Rec}}$.

All in all, this leads to

$$
\begin{aligned}
&\Theta_{\texttt{LiteratureHost1}} \\
= \; &((S_D \cup \\
&\{\texttt{searchF}^{i}, loc^{i}\} \cup \; : S^{i}_{O_{io}} \\
&\{\texttt{Reception.searchFacility}^{i}\} \cup \; : S^{i}_{O_{Rec_1}} \\
&\{\texttt{Reception}.loc^{i}\} \cup \; : S^{i}_{O_{Rec_2}} \\
&\{\texttt{searchF}^{at}, loc^{at}\} \cup \; : S^{at}_{O_{io}} \\
&\{\texttt{Reception.searchFacility}^{at}\} \cup \; : S^{at}_{O_{Rec_1}} \\
&\{\texttt{Reception}.loc^{at}\} \cup \; : S^{at}_{O_{Rec_2}} \\
&\{\texttt{searchF}^{ac}, loc^{ac}\} \cup \; : S^{ac}_{O_{io}} \\
&\{\texttt{Reception.searchFacility}^{ac}\} \cup \; : S^{ac}_{O_{Rec_1}} \\
&\{\texttt{Reception}.loc^{ac}\} \cup \; : S^{ac}_{O_{Rec_2}} \\
&\left.\begin{array}{l}
\left.\begin{array}{l}
\{\texttt{LiteratureHost1, searchF}, loc\} \cup \quad : S^{e}_{M} \\
\left.\begin{array}{l}
\{\texttt{Reception.searchFacility, Reception}.loc, \\
\texttt{Reception}\} \cup
\end{array}\right\} : S^{i}_{M} \\
\left.\begin{array}{l}
\{\texttt{LiteratureHost1}, bod\} \cup \quad : S^{b}_{M} \\
\{\texttt{LiteratureHost1}\}, \quad : S^{o}_{M}
\end{array}\right\} : S^{\times}_{M}
\end{array}\right\} : S^{+}_{M}
\end{array}\right\} : S_M \\
&\Omega_D \cup \\
&\{\texttt{searchF} :\rightarrow \texttt{searchF}^{i}, loc :\rightarrow loc^{i}\} \cup \; : \Omega^{i}_{O_{io}} \\
&\{\texttt{reception.searchFacility} : \rightarrow \texttt{Reception.searchFacility}^{i}\} \cup \; : \Omega^{i}_{O_{Rec_1}} \\
&\{\texttt{reception}.loc : \rightarrow \texttt{Reception}.loc^{i}\} \cup \; : \Omega^{i}_{O_{Rec_2}} \\
&\{\lambda_{\mathsf{string}} : loc^{i} \times loc^{at} \rightarrow \mathsf{string}\} \cup \; : \Omega^{at}_{O_{io}} \\
&\{\} \cup \; : \Omega^{at}_{O_{Rec_1}} \\
&\{\texttt{reception}.\lambda_{\mathsf{string}} : \texttt{Reception}.loc^{i} \times \texttt{Reception}.loc^{at} \rightarrow \mathsf{string}\} \cup \; : \Omega^{at}_{O_{Rec_2}} \\
&\left.\begin{array}{l}
\{\texttt{searchKeyw} : \texttt{searchF}^{i} \times \mathsf{set}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchF}^{ac}, \\
\lambda_{read} : loc^{i} \times \mathsf{string} \rightarrow loc^{ac}\} \cup
\end{array}\right\} : \Omega^{ac}_{O_{io}} \\
&\left.\begin{array}{l}
\{\texttt{reception.searchByKeywords} : \\
\texttt{Reception.searchFacility}^{i} \times \mathsf{set}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \\
\rightarrow \texttt{Reception.searchFacility}^{ac}\} \cup
\end{array}\right\} : \Omega^{ac}_{O_{Rec_1}}
\end{aligned}
$$

$\{\texttt{reception}.\lambda_{read} : \texttt{Reception}.loc^{i} \times \mathsf{string} \rightarrow \texttt{Reception}.loc^{ac}\} \cup \quad : \Omega^{ac}_{O_{Rec_2}}$

$$\left.\begin{array}{l}\{\texttt{litHost1} : \rightarrow \texttt{LiteratureHost1},\\ \texttt{litHost1.searchF} : \rightarrow \texttt{searchF},\\ \texttt{litHost1}.loc : \rightarrow loc,\\ \texttt{reception.searchFacility} : \rightarrow \texttt{searchFacility},\\ \texttt{reception}.loc : \rightarrow loc,\\ \texttt{reception} : \rightarrow \texttt{Reception}\\ \texttt{bod} : \rightarrow bod\}),\end{array}\right\} : \Omega_M$$

$((S_D \cup \{bod\}, \Omega_D \cup \{bod : \rightarrow bod\}), \{\Sigma, id\}), \quad : \Sigma_{K_{bod}}$

$\{\Sigma_{K_{Rec}}\}, \quad : Imp$

$Exp)$

A.3 LiteratureSystem

In the TROLL example specification for the mobile-agent-based information retrieval, we refrained from providing interface objects for the communication between the `admin` object from `Users` and the `planner` object from `ManagingUnit` for reasons of simplicity. Likewise, only one `LiteratureHost` was specified. The following signature of the `LiteratureSystem` complies with the specification as given, and therefore neither contains items related to the `admin/planner` relations nor to multiple `LiteratureHosts`.

1. For the specification ω of the stationary subsystem denoted by `LiteratureSystem` ($= su$) the local module sort according to definition 6.39 (1.) is $\alpha = su =$ `LiteratureSystem`. With

   (a) $ic($`LiteratureSystem`$) = \{$`searchEng`$, loc\}$
   $(ic(\omega) = \{ids_1, \ldots, ids_n\}$, set of all identifiers of ω's interface objects with local module sorts $\alpha_1, \ldots, \alpha_n$, where (again by definition 6.39 (2.)) $\alpha_i = ids_i)$

   (b) $subSpec($`LiteratureSystem`$) =$
   $\{$`ManagingUnit, RetrievalUnit, LiteratureHost1`$\}$
   $(subSpec(\omega) = \{\omega_1, \ldots, \omega_m\})$

   (c) $mSubSpec($`LiteratureSystem`$) = \{$`RetrievalUnit`$\}$

   (d) $sSubSpec($`LiteratureSystem`$) = \{$`ManagingUnit, LiteratureHost1`$\}$
   $(subSpec \supseteq sSubSpec = \{\omega_{l_1}, \ldots, \omega_{l_h}\})$

   (e) $ic($`ManagingUnit`$) = \{$`searchFacility, tripPlanner, searchEngine`$, loc\}$
   $(ic(\omega_i) = \{ids_{i,1}, \ldots, ids_{i,k_i}\} \overset{6.39\ (2.)}{=} \{\gamma_{i,1}, \ldots, \gamma_{i,k_i}\})$

   (f) $ic($`RetrievalUnit`$) = \{$`searchFacility, tripPlanner, searchEngine`$, loc\}$
   $(ic(\omega_i) = \{ids_{i,1}, \ldots, ids_{i,k_i}\} \overset{6.39\ (2.)}{=} \{\gamma_{i,1}, \ldots, \gamma_{i,k_i}\})$

   (g) $ic($`LiteratureHost1`$) = \{$`searchF`$, loc\}$
   $(ic(\omega_i) = \{ids_{i,1}, \ldots, ids_{i,k_i}\} \overset{6.39\ (2.)}{=} \{\gamma_{i,1}, \ldots, \gamma_{i,k_i}\})$

   we therefore obtain

   (a) $S_M^e = \{$`LiteratureSystem, searchEng`$, loc\}$

   (b) $S_M^i =$

   $\{$`ManagingUnit.searchFacility`$, \ldots,$ `ManagingUnit.`loc,
   `RetrievalUnit.searchFacility`$, \ldots,$ `RetrievalUnit.`loc,
   `LiteratureHost1.searchF`, `LiteratureHost1.`loc,
   `ManagingUnit`, `RetrievalUnit`, `LiteratureHost1`$\}$

   (c) $S_M^\times = S_M^b \cup S_M^o = \{$`LiteratureSystem`$, \beta\} \cup \{$`LiteratureSystem`$\}$

2. $\Omega_M =$

$\{\texttt{litSys} : \rightarrow \texttt{LiteratureSystem},$
$\texttt{litSys.searchEng} : \rightarrow \texttt{searchEng}, \texttt{litSys.}loc : \rightarrow loc,$
$\texttt{managingUnit.searchFacility} : \rightarrow \texttt{searchFacility}, \ldots,$
$\texttt{managingUnit.}loc : \rightarrow loc, \texttt{managingUnit} : \rightarrow \texttt{ManagingUnit},$
$\texttt{retrievalUnit.searchFacility} : \rightarrow \texttt{searchFacility}, \ldots,$
$\texttt{retrievalUnit.}loc : \rightarrow loc, \texttt{retrievalUnit} : \rightarrow \texttt{RetrievalUnit},$
$\texttt{litHost1.searchF} : \rightarrow \texttt{searchF}, \texttt{litHost1.}loc : \rightarrow loc,$
$\texttt{litHost1} : \rightarrow \texttt{LiteratureHost1},$
$\texttt{bod} : \rightarrow bod\}$

3. (a) The stationary subsystem `LiteratureSystem` contains only the offer-object `searchEng`, with class signature

$$\begin{aligned}\Sigma_C(\texttt{searchEng}) &= (S_O, I, AT, AC)\\ &\quad (\{\texttt{searchEng}\},\\ &\quad \{\texttt{searchEng} : \rightarrow \texttt{searchEng}\},\\ &\quad \{\},\\ &\quad \{\texttt{searchKeywords} : \mathsf{string} \times \mathsf{set}(\mathsf{string}) \times\\ &\quad \mathsf{list}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo})\}),\end{aligned}$$

used to establish a connection to `searchFacility` in `ManagingUnit`, and the implied location-reflecting interface object loc, whose class signature according to definition 6.35 is given by

$$\begin{aligned}\Sigma_C(loc) &= (S_O, I, AT, AC)\\ &= (\{loc\},\\ &\quad \{loc : \rightarrow loc\},\\ &\quad \{\lambda_{\mathsf{string}} : loc \rightarrow \mathsf{string}\},\\ &\quad \{\lambda_{read} : \mathsf{string} \rightarrow loc\}).\end{aligned}$$

(b) extended data signatures: Building upon the former we obtain

$$\begin{aligned}S^i_{O_{io}} &= \{\texttt{searchEng}^i, loc^i\}\\ S^{at}_{O_{io}} &= \{\texttt{searchEng}^{at}, loc^{at}\}\\ S^{ac}_{O_{io}} &= \{\texttt{searchEng}^{ac}, loc^{ac}\}\\ \Omega^i_{O_{io}} &= \{\texttt{searchEng} : \rightarrow \texttt{searchEng}^i, loc : \rightarrow loc^i\}\\ \Omega^{at}_{O_{io}} &= \{\lambda_{\mathsf{string}} : loc^i \times loc^{at} \rightarrow \mathsf{string}\}\\ \Omega^{ac}_{O_{io}} &= \{\texttt{searchKeywords} : \texttt{searchEng}^i \times \mathsf{string} \times \mathsf{set}(\mathsf{string}) \times\\ &\quad \mathsf{list}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchEng}^{ac},\\ &\quad \lambda_{read} : loc^i \times \mathsf{string} \rightarrow loc^{ac}\}\end{aligned}$$

in analogy to example 6.44 and [Eck01, p. 131] and [Eck01, Bsp. 3.2].

4. (a) extended data signatures for the body module signature:
$\Sigma_{K_{bod}} = ((S_D \cup \{bod\}, \Omega_D \cup \{bod : \rightarrow bod\}), \{\Sigma, id\})$

   (b) import module signatures derived from the interface objects of the internal subsystems [...]: `LiteratureSystem` has three internal subsystem: `ManagingUnit, RetrievalUnit` and `LiteratureHost1`. (We refrain from considering `LiteratureHost2`, ..., `LiteratureHost`N, which would be identical to `LiteratureHost1`). Their sets of interface objects have already been given under 1.(e)–(g). The corresponding class signatures are:

   - for the *stationary* `ManagingUnit`:
   $\Sigma_C(\texttt{searchFacility})$
   $= (S_O, I, AT, AC)$
   $= (\{\texttt{searchFacility}\},$
   $\{\texttt{searchFacility} : \rightarrow \texttt{searchFacility}\},$
   $\{\}$
   $\{\texttt{searchKeywords} : \mathsf{string} \times \mathsf{set}(\mathsf{string})$
   $\times \mathsf{list}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchFacility}\}),$
   ...,
   $\Sigma_C(loc)$
   $= (S_O, I, AT, AC)$
   $= (\{loc\},$
   $\{loc : \rightarrow loc\},$
   $\{\lambda_{\mathsf{string}} : loc \rightarrow \mathsf{string}$
   $\{\lambda_{read} : \mathsf{string} \rightarrow loc\}),$
   - for the *mobile* `RetrievalUnit`:
   $\Sigma_C(\texttt{searchFacility})$
   $= (S_O, I, AT, AC)$
   $= (\{\texttt{searchFacility}\},$
   $\{\texttt{searchFacility} : \rightarrow \texttt{searchFacility}\},$
   $\{\},$
   $\{\texttt{searchKeywords} : \mathsf{set}(\mathsf{string}) \times \mathsf{list}(\mathsf{string})$
   $\times \mathsf{set}(\texttt{litInfo}) \rightarrow \texttt{searchFacility}\}),$
   ...,
   $\Sigma_C(loc)$
   $= (S_O, I, AT, AC)$
   $= (\{loc\},$
   $\{loc : \rightarrow loc\},$
   $\{\lambda_{\mathsf{string}} : loc \rightarrow \mathsf{string}\},$

$$\{\lambda_{read} : \mathsf{string} \to loc, \lambda_{write} : \mathsf{string} \to loc\})$$

and

- for the *stationary* `LiteratureHost1`:
  $\Sigma_C(\texttt{searchF})$
  $$\begin{aligned} &= (S_O, I, AT, AC) \\ &= (\{\texttt{searchF}\}, \\ &\quad \{\texttt{searchF} : \to \texttt{searchF}\}, \\ &\quad \{\}, \\ &\quad \{\texttt{searchKeyw} : \mathsf{set}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \\ &\quad \to \texttt{searchFacility}\}), \end{aligned}$$
  $\ldots,$
  $\Sigma_C(loc)$
  $$\begin{aligned} &= (S_O, I, AT, AC) \\ &= (\{loc\}, \\ &\quad \{loc : \to loc\}, \\ &\quad \{\lambda_{\mathsf{string}} : loc \to \mathsf{string}\}, \\ &\quad \{\lambda_{read} : \mathsf{string} \to loc\}). \end{aligned}$$

The extended data signatures are then:

- for the *stationary* `ManagingUnit`:
  $$\begin{aligned} S^i_{O_{MU_1}} &= \{\texttt{ManagingUnit.searchFacility}^i\}, \\ S^{at}_{O_{MU_1}} &= \{\texttt{ManagingUnit.searchFacility}^{at}\}, \\ S^{ac}_{O_{MU_1}} &= \{\texttt{ManagingUnit.searchFacility}^{ac}\}, \\ \Omega^i_{O_{MU_1}} &= \{\texttt{managingUnit.searchFacility} : \\ &\quad \to \texttt{ManagingUnit.searchFacility}^i\}, \\ \Omega^{at}_{O_{MU_1}} &= \{\}, \\ \Omega^{ac}_{O_{MU_1}} &= \{\texttt{managingUnit.searchKeywords} : \\ &\quad \texttt{ManagingUnit.searchFacility}^i \times \mathsf{string} \\ &\quad \times \mathsf{set}(\mathsf{string}) \times \mathsf{list}(\mathsf{string}) \times \mathsf{set}(\texttt{litInfo}) \\ &\quad \to \texttt{ManagingUnit.searchFacility}^{ac}\}, \\ &\vdots \\ S^i_{O_{MU_4}} &= \{\texttt{ManagingUnit.}loc^i\}, \\ S^{at}_{O_{MU_4}} &= \{\texttt{ManagingUnit.}loc^{at}\}, \\ S^{ac}_{O_{MU_4}} &= \{\texttt{ManagingUnit.}loc^{ac}\}, \\ \Omega^i_{O_{MU_4}} &= \{\texttt{managingUnit.}loc : \to \texttt{ManagingUnit.}loc^i\}, \\ \Omega^{at}_{O_{MU_4}} &= \{\texttt{managingUnit.}\lambda_{\mathsf{string}} : \texttt{ManagingUnit.}loc^i \\ &\quad \times \texttt{ManagingUnit.}loc^{at} \to \mathsf{string}\}, \end{aligned}$$

$$\Omega^{ac}_{O_{MU_4}} = \{\texttt{managingUnit}.\lambda_{read} : \texttt{ManagingUnit}.loc^i \times \mathsf{string} \rightarrow \texttt{ManagingUnit}.loc^{ac}\}$$

from $\Sigma_{K_{MU}}$,

- for the *mobile* `RetrievalUnit`:

$$\begin{aligned}
S^i_{O_{RU_1}} &= \{\texttt{RetrievalUnit.searchFacility}^i\},\\
S^{at}_{O_{RU_1}} &= \{\texttt{RetrievalUnit.searchFacility}^{at}\},\\
S^{ac}_{O_{RU_1}} &= \{\texttt{RetrievalUnit.searchFacility}^{ac}\},\\
\Omega^i_{O_{RU_1}} &= \{\texttt{retrievalUnit.searchFacility} : \rightarrow \texttt{RetrievalUnit.searchFacility}^i\}\\
\Omega^{at}_{O_{RU_1}} &= \{\}\\
\Omega^{ac}_{O_{RU_1}} &= \{\texttt{retrievalUnit.searchKeywords} : \texttt{RetrievalUnit.searchFacility}^i \times \mathsf{set(string)} \times \mathsf{list(string)} \times \mathsf{set(litInfo)} \rightarrow \texttt{RetrievalUnit.searchFacility}^{ac}\},\\
&\vdots\\
S^i_{O_{RU_4}} &= \{\texttt{RetrievalUnit}.loc^i\},\\
S^{at}_{O_{RU_4}} &= \{\texttt{RetrievalUnit}.loc^{at}\},\\
S^{ac}_{O_{RU_4}} &= \{\texttt{RetrievalUnit}.loc^{ac}\},\\
\Omega^i_{O_{RU_4}} &= \{\texttt{retrievalUnit}.loc : \rightarrow \texttt{RetrievalUnit}.loc^i\}\\
\Omega^{at}_{O_{RU_4}} &= \{\texttt{retrievalUnit}.\lambda_{\mathsf{string}} : \texttt{RetrievalUnit}.loc^i \times \texttt{RetrievalUnit}.loc^{at} \rightarrow \mathsf{string}\}\\
\Omega^{ac}_{O_{RU_4}} &= \{\texttt{retrievalUnit}.\lambda_{read} : \texttt{RetrievalUnit}.loc^i \times \mathsf{string} \rightarrow \texttt{RetrievalUnit}.loc^{ac},\\
&\quad \texttt{retrievalUnit}.\lambda_{write} : \texttt{RetrievalUnit}.loc^i \times \mathsf{string} \rightarrow \texttt{RetrievalUnit}.loc^{ac}\}
\end{aligned}$$

from $\Sigma_{K_{RU}}$, and

- for the *stationary* `LiteratureHost1`:

$$\begin{aligned}
S^i_{O_{LH1_1}} &= \{\texttt{LiteratureHost1.searchF}^i\},\\
S^{at}_{O_{LH1_1}} &= \{\texttt{LiteratureHost1.searchF}^{at}\},\\
S^{ac}_{O_{LH1_1}} &= \{\texttt{LiteratureHost1.searchF}^{ac}\},\\
\Omega^i_{O_{LH1_1}} &= \{\texttt{litHost1.searchF} : \rightarrow \texttt{LiteratureHost1.searchF}^i\}\\
\Omega^{at}_{O_{LH1_1}} &= \{\}\\
\Omega^{ac}_{O_{LH1_1}} &= \{\texttt{litHost1.searchKeyw} : \texttt{LiteratureHost1.searchF}^i \times \mathsf{set(string)}
\end{aligned}$$

$$
\begin{aligned}
& \times \mathsf{set}(\texttt{litInfo}) \to \texttt{LiteratureHost1.searchF}^{ac}\},\\
S^{i}_{O_{LH1_2}} &= \{\texttt{LiteratureHost1.}loc^{i}\},\\
S^{at}_{O_{LH1_2}} &= \{\texttt{LiteratureHost1.}loc^{at}\},\\
S^{ac}_{O_{LH1_2}} &= \{\texttt{LiteratureHost1.}loc^{ac}\},\\
\Omega^{i}_{O_{LH1_2}} &= \{\texttt{litHost1.}loc : \to \texttt{LiteratureHost1.}loc^{i}\}\\
\Omega^{at}_{O_{LH1_2}} &= \{\texttt{litHost1.}\lambda_{\mathsf{string}} : \texttt{LiteratureHost1.}loc^{i}\\
& \quad \times \texttt{LiteratureHost1.}loc^{at} \to \mathsf{string}\}\\
\Omega^{ac}_{O_{LH1_2}} &= \{\texttt{litHost1.}\lambda_{read} : \texttt{LiteratureHost1.}loc^{i}\\
& \quad \times \mathsf{string} \to \texttt{LiteratureHost1.}loc^{ac}\}
\end{aligned}
$$

from $\Sigma_{K_{LH1}}$.

All in all, this leads to

$$
\begin{aligned}
& \Theta_{\texttt{LiteratureSystem}}\\
= \; & ((S_D \cup\\
& \{\texttt{searchEng}^{i}, loc^{i}\} \cup \quad : S^{i}_{O_{io}}\\
& \{\texttt{ManagingUnit.searchFacility}^{i}\} \cup \ldots \cup \{\texttt{ManagingUnit.}loc^{i}\} \cup \quad : S^{i}_{O_{MU_{1-4}}}\\
& \{\texttt{RetrievalUnit.searchFacility}^{i}\} \cup \ldots \cup \{\mathrm{RetrievalUnit.}loc^{i}\} \cup \quad : S^{i}_{O_{RU_{1-4}}}\\
& \{\texttt{LiteratureHost1.searchF}^{i}\} \cup \{\texttt{LiteratureHost1.}loc^{i}\} \cup \quad : S^{i}_{O_{LH1_1}} \cup S^{i}_{O_{LH1_2}}\\
& \{\texttt{searchEng}^{at}, loc^{at}\} \cup \quad : S^{at}_{O_{io}}\\
& \{\texttt{ManagingUnit.searchFacility}^{at}\} \cup \ldots \cup \{\texttt{ManagingUnit.}loc^{at}\} \cup \quad : S^{at}_{O_{MU_{1-4}}}\\
& \{\texttt{RetrievalUnit.searchFacility}^{at}\} \cup \ldots \cup \{\texttt{RetrievalUnit.}loc^{at}\} \cup \quad : S^{at}_{O_{RU_{1-4}}}\\
& \{\texttt{LiteratureHost1.searchF}^{at}\} \cup \{\texttt{LiteratureHost1.}loc^{at}\} \cup \quad : S^{at}_{O_{LH1_1}} \cup S^{at}_{O_{LH1_2}}\\
& \{\texttt{searchEng}^{ac}, loc^{ac}\} \cup \quad : S^{ac}_{O_{io}}\\
& \{\texttt{ManagingUnit.searchFacility}^{ac}\} \cup \ldots \cup \{\texttt{ManagingUnit.}loc^{ac}\} \cup \quad : S^{ac}_{O_{MU_{1-4}}}\\
& \{\texttt{RetrievalUnit.searchFacility}^{ac}\} \cup \ldots \cup \{\texttt{RetrievalUnit.}loc^{ac}\} \cup \quad : S^{ac}_{O_{RU_{1-4}}}\\
& \{\texttt{LiteratureHost1.searchF}^{ac}\} \cup \{\texttt{LiteratureHost1.}loc^{ac}\} \cup \quad : S^{ac}_{O_{LH1_1}} \cup S^{ac}_{O_{LH1_2}}\\
& \left.\begin{array}{l}
\{\texttt{LiteratureSystem, searchEng, }loc\} \cup \quad : S^{e}_{M}\\
\left.\begin{array}{l}
\{\texttt{ManagingUnit.searchFacility}, \ldots, \texttt{ManagingUnit.}loc,\\
\texttt{RetrievalUnit.searchFacility}, \ldots, \texttt{RetrievalUnit.}loc,\\
\texttt{LiteratureHost1.searchF, LiteratureHost1.}loc,\\
\texttt{ManagingUnit, RetrievalUnit, LiteratureHost1}\} \cup
\end{array}\right\} : S^{i}_{M}\\
\left.\begin{array}{l}
\{\texttt{LiteratureSystem}, bod\} \cup \quad : S^{b}_{M}\\
\{\texttt{LiteratureSystem}\}, \quad : S^{o}_{M}
\end{array}\right\} : S^{\times}_{M}
\end{array}\right\} : S^{+}_{M}\Bigg\} : S_M\\
& \Omega_D \cup\\
& \{\texttt{searchEng} : \to \texttt{searchEng}^{i}, loc :\to loc^{i}\} \cup \quad : \Omega^{i}_{O_{io}}
\end{aligned}
$$

$$\left.\begin{array}{l}\{\texttt{managingUnit.searchFacility}: \\ \rightarrow \texttt{ManagingUnit.searchFacility}^i\} \cup \ldots \cup \\ \{\texttt{managingUnit}.loc : \rightarrow \texttt{ManagingUnit}.loc^i\} \cup\end{array}\right\} : \Omega^i_{O_{MU_{1-4}}}$$

$$\left.\begin{array}{l}\{\texttt{retrievalUnit.searchFacility}: \\ \rightarrow \texttt{RetrievalUnit.searchFacility}^i\} \cup \ldots \cup \\ \{\texttt{retrievalUnit}.loc : \rightarrow \texttt{RetrievalUnit}.loc^i\} \cup\end{array}\right\} : \Omega^i_{O_{RU_{1-4}}}$$

$$\left.\begin{array}{l}\{\texttt{litHost1.searchF} : \rightarrow \texttt{LiteratureHost1.searchF}^i\} \cup \\ \{\texttt{litHost1}.loc : \rightarrow \texttt{LiteratureHost1}.loc^i\} \cup\end{array}\right\} : \Omega^i_{O_{LH_{1+2}}}$$

$$\{\lambda_{\text{string}} : loc^i \times loc^{at} \rightarrow \text{string}\} \cup \quad : \Omega^{at}_{O_{io}}$$

$$\left.\begin{array}{l}\{\} \cup \ldots \cup \\ \{\texttt{managingUnit}.\lambda_{\text{string}} : \texttt{ManagingUnit}.loc^i \\ \times \texttt{ManagingUnit}.loc^{at} \rightarrow \text{string}\} \cup\end{array}\right\} : \Omega^{at}_{O_{MU_{1-4}}}$$

$$\left.\begin{array}{l}\{\} \cup \ldots \cup \\ \{\texttt{retrievalUnit}.\lambda_{\text{string}} : \texttt{RetrievalUnit}.loc^i \\ \times \texttt{RetrievalUnit}.loc^{at} \rightarrow \text{string}\} \cup\end{array}\right\} : \Omega^{at}_{O_{RU_{1-4}}}$$

$$\left.\begin{array}{l}\{\} \cup \ldots \cup \\ \{\texttt{litHost1}.\lambda_{\text{string}} : \texttt{LiteratureHost1}.loc^i \\ \times \texttt{LiteratureHost1}.loc^{at} \rightarrow \text{string}\} \cup\end{array}\right\} : \Omega^{at}_{O_{LH1_{1+2}}}$$

$$\left.\begin{array}{l}\{\texttt{searchKeywords} : \texttt{searchEng}^i \times \text{string} \times \text{set(string)} \\ \times \text{list(string)} \times \text{set}(\texttt{litInfo}) \rightarrow \texttt{searchEng}^{ac}, \\ \lambda_{read} : loc^i \times \text{string} \rightarrow loc^{ac}\} \cup\end{array}\right\} : \Omega^{ac}_{O_{io}}$$

$$\left.\begin{array}{l}\{\texttt{managinUnit.searchKeywords}: \\ \texttt{ManagingUnit.searchFacility}^i \times \text{string} \times \text{set(string)} \\ \times \text{list(string)} \times \text{set}(\texttt{litInfo}) \\ \rightarrow \texttt{ManagingUnit.searchFacility}^{ac}\} \cup \ldots \cup \\ \{\texttt{managingUnit}.\lambda_{read} : \\ \texttt{ManagingUnit}.loc^i \times \text{string} \rightarrow \texttt{ManagingUnit}.loc^{ac}\} \cup\end{array}\right\} : \Omega^{ac}_{O_{MU_{1-4}}}$$

$$\left.\begin{array}{l}\{\texttt{retrievalUnit.searchKeywords}: \\ \texttt{RetrievalUnit.searchFacility}^i \times \text{set(string)} \\ \times \text{list(string)} \times \text{set}(\texttt{litInfo}) \\ \rightarrow \texttt{RetrievalUnit.searchFacility}^{ac}\} \cup \ldots \cup \\ \{\texttt{retrievalUnit}.\lambda_{read} : \\ \texttt{RetrievalUnit}.loc^i \times \text{string} \rightarrow \texttt{RetrievalUnit}.loc^{ac}, \\ \texttt{retrievalUnit}.\lambda_{write} : \\ \texttt{RetrievalUnit}.loc^i \times \text{string} \rightarrow \texttt{RetrievalUnit}.loc^{ac}\} \cup\end{array}\right\} : \Omega^{ac}_{O_{RU_{1-4}}}$$

$$\left.\begin{array}{l}\{\texttt{litHost1.searchKeyw:} \\ \texttt{LiteratureHost1.searchF}^i \times \text{set(string)} \times \text{set(litInfo)} \\ \rightarrow \texttt{LiteratureHost1.searchF}^{ac}\} \cup \\ \{\texttt{litHost1}.\lambda_{read} : \texttt{LiteratureHost1}.loc^i \times \text{string} \\ \rightarrow \texttt{LiteratureHost1}.loc^{ac}\} \cup\end{array}\right\} : \Omega^{ac}_{O_{LH1_{1+2}}}$$

$$\left.\begin{array}{l}\{\texttt{litSys} : \rightarrow \texttt{LiteratureSystem},\\ \texttt{litSys.searchEng} : \rightarrow \texttt{searchEng}, \texttt{litSys}.loc : \rightarrow loc,\\ \texttt{managingUnit.searchFacility} : \rightarrow \texttt{searchFacility}, \ldots,\\ \texttt{managingUnit}.loc : \rightarrow loc, \texttt{managingUnit} : \rightarrow \texttt{ManagingUnit},\\ \texttt{retrievalUnit.searchFacility} : \rightarrow \texttt{searchFacility}, \ldots,\\ \texttt{retrievalUnit}.loc : \rightarrow loc, \texttt{retrievalUnit} : \rightarrow \texttt{RetrievalUnit},\\ \texttt{litHost1.searchF} : \rightarrow \texttt{searchF}, \ldots, \texttt{litHost1}.loc : \rightarrow loc,\\ \texttt{litHost1} : \rightarrow \texttt{LiteratureHost1},\\ \texttt{bod} : \rightarrow bod\}),\end{array}\right\} : \Omega_M$$

$$((S_D \cup \{bod\}, \Omega_D \cup \{bod : \rightarrow bod\}), \{\Sigma, id\}), \quad : \Sigma_{K_{bod}}$$

$$\{\Sigma_{K_{MU}}, \Sigma_{K_{RU}}, \Sigma_{K_{LH1}}\}, \quad : Imp$$

$$Exp)$$

A.4 launchRetrieval(kw, tL, resultSet) and Migration Pattern

In analogy to example 6.53, treating the mobile subsystem `RetrievalUnit` as a system specification its operation definition

```
launchRetrieval(kw, tL, resultSet)
 onlyIf cnt(tL) > 1
do
 keywords:=kw,
 homeLoc:=tL(1),
 nextTarget:=tL(2), ...,
 if (true) moveTo tL(1)
od;
```

gives rise to the following axioms for the `retriever` object (abbreviated by r):

$$r.(\odot r.\texttt{launchRetrieval(kw, tL, resultSet)} \Rightarrow \mathcal{A}_{r,X}(\texttt{keywords:=kw}, \texttt{homeLoc:=tL(1)}, \texttt{nextTarget:=tL(2)}, \texttt{if (true) moveTo tL(1)}))$$

for $X = var_{in} \cup var_{out} \cup var_{local} = \{\texttt{kw, tL}\} \cup \{\texttt{resultSet}\} \cup \{\}$ according to definition 6.21, and

$$r.(\triangleright r.\texttt{launchRetrieval(kw, tL, resultSet)} \Rightarrow \mathcal{P}_{r,\{\texttt{kw, tL}\}}(\texttt{cnt(tL)} > 1)$$

using definition 6.18. Again annotating equivalences by definitions applied,

$$\mathcal{P}_{r,\{\texttt{kw, tL}\}}(\texttt{cnt(tL)} > 1) \overset{6.13,1.}{\equiv} \exists\, v\, \mathcal{T}_{r,\{\texttt{kw, tL}\}}(v = \texttt{cnt(tL)} > 1) \wedge v = true \overset{6.15,4.(c)}{\equiv}$$
$$\exists\, v\, \exists\, v_1, v_2\, (\mathcal{T}_{r,\{\texttt{kw, tL}\}}(v_1 = \texttt{cnt(tL)}) \wedge \mathcal{T}_{r,\{\texttt{kw, tL}\}}(v_2 = 1) \wedge (v = v_1 > v_2) \wedge v = true),$$

where

$\mathcal{T}_{r,\{\texttt{kw, tL}\}}(v_1 = \texttt{cnt(tL)}) \overset{6.15,4.(c)}{\equiv} \exists\, v_1^1\, (\mathcal{T}_{r,\{\texttt{kw, tL}\}}(v_1^1 = \texttt{tL}) \wedge (v_1 = \texttt{cnt}(v_1^1)))$,
$\mathcal{T}_{r,\{\texttt{kw, tL}\}}(v_1^1 = \texttt{tL}) \overset{6.15,2.(a)}{\equiv} v_1^1 = \texttt{tL}$,

and $\mathcal{T}_{r,\{\texttt{kw, tL}\}}(v_2 = 1) \overset{6.15,1.}{\equiv} v_2 = 1$, leading to

$r.(\triangleright r\texttt{.launchRetrieval(kw, tL, resultSet)} \Rightarrow \mathcal{P}_{r,\{\texttt{kw, tL}\}}(\texttt{cnt(tL)} > 1) \equiv$
$\exists\, v\, \exists\, v_1, v_2\, (\exists\, v_1^1 (v_1^1 = \texttt{tL} \wedge v_1 = \texttt{cnt}(v_1^1)) \wedge v_2 = 1 \wedge (v = v_1 > v_2))$,

which can be simplified to

$r.(\triangleright r\texttt{.launchRetrieval(kw, tL, resultSet)} \Rightarrow (\texttt{cnt(tL)} > 1)$,

can be derived for the precondition, and the for the other axiom we obtain

$r.(\odot r.\texttt{launchRetrieval(kw, tL, resultSet)} \Rightarrow$
$\quad \mathcal{A}_{r,X}(\texttt{keywords:=kw}, \texttt{homeLoc:=tL(1)}, \texttt{nextTarget:=tL(2)},$
$\qquad \texttt{if (true) moveTo tL(1)})) \overset{6.21,5.}{\equiv}$
$r.(\odot r.\texttt{launchRetrieval(kw, tL, resultSet)} \Rightarrow$
$\quad \mathcal{A}_{r,X}(\texttt{keywords:=kw}),$
$\quad \mathcal{A}_{r,X}(\texttt{homeLoc:=tL(1)}),$
$\quad \mathcal{A}_{r,X}(\texttt{nextTarget:=tL(2)}),$
$\quad \mathcal{A}_{r,X}(\texttt{if (true) moveTo tL(1)}))$.

Using

1. $\mathcal{A}_{r,X}(\texttt{keywords:=kw}) \overset{6.21,1.(b)}{\equiv}$
   $\exists\, v_{11}\, \mathsf{Y}(\mathcal{T}_{r,X}(v_{11} = \texttt{kw})) \wedge \odot r\texttt{.keywords}_{write}(v_{11})$, where
   $\mathcal{T}_{r,X}(v_{11} = \texttt{kw}) \overset{6.15,2.(a)}{\equiv} v_{11} = \texttt{kw}$,

2. $\mathcal{A}_{r,X}(\texttt{homeLoc:=tL(1)}) \overset{6.21,1.(b)}{\equiv}$
   $\exists\, v_{21}\, \mathsf{Y}(\mathcal{T}_{r,X}(v_{21} = \texttt{tL(1)})) \wedge \odot r\texttt{.homeLoc}_{write}(v_{21})$, where
   $\mathcal{T}_{r,X}(v_{21} = \texttt{tL(1)}) \overset{6.15,2.(b)}{\equiv} v_{21} = \texttt{tL(1)}$,

3. $\mathcal{A}_{r,X}(\texttt{nextTarget:=tL(2)}) \overset{6.21,1.(b)}{\equiv}$
   $\exists\, v_{31}\, \mathsf{Y}(\mathcal{T}_{r,X}(v_{31} = \texttt{tL(2)})) \wedge \odot r\texttt{.nextTarget}_{write}(v_{31})$, where
   $\mathcal{T}_{r,X}(v_{31} = \texttt{tL(2)}) \overset{6.15,2.(b)}{\equiv} v_{31} = \texttt{tL(2)}$, and

4. $\mathcal{A}_{r,X}(\texttt{if (true) moveTo tL(1)})) \overset{6.52,2.}{\equiv}$
   $(\mathsf{Y}(\mathcal{P}_{r,X}(true)) \Rightarrow \exists\, v_{41}\, (v_{41} = \texttt{tL(1)} \wedge \mathsf{X}(\odot loc.\lambda_{write}(v_{41}))))$, where
   $\mathcal{P}_{r,X}(true) \overset{6.13,1.}{\equiv} \exists\, v_{42}\, \mathcal{T}_{r,X}(v_{42} = true) \wedge v_{42} = true$,
   $\mathcal{T}_{r,X}(v_{42} = true) \overset{6.15,1.}{\equiv} v_{42} = true$,

this gives

$r.(\odot r.\texttt{launchRetrieval(kw, tL, resultSet)} \Rightarrow$
$\quad \exists\, v_{11}\, \mathsf{Y}(v_{11} = \texttt{kw}) \wedge \odot r.\texttt{keywords}_{write}(v_{11})$
$\quad \wedge \exists\, v_{21}\, \mathsf{Y}(v_{21} = \texttt{tL(1)}) \wedge \odot r.\texttt{homeLoc}_{write}(v_{21})$
$\quad \wedge \exists\, v_{31}\, \mathsf{Y}(v_{31} = \texttt{tL(2)}) \wedge \odot r.\texttt{nextTarget}_{write}(v_{31}) \wedge \ldots$
$\quad \wedge(\mathsf{Y}(\exists\, v_{42}\, v_{42} = true \wedge v_{42} = true) \Rightarrow \exists\, v_{41}\, (v_{41} = \texttt{tL(1)} \wedge \mathsf{X}(\odot loc.\lambda_{write}(v_{41})))))$,

which can be simplified to

$r.(\odot r.\texttt{launchRetrieval(kw, tL, resultSet)} \Rightarrow$
$\quad \exists\, v_{11}\, \mathsf{Y}(v_{11} = \texttt{kw}) \wedge \odot r.\texttt{keywords}_{write}(v_{11})$
$\quad \wedge \exists\, v_{21}\, \mathsf{Y}(v_{21} = \texttt{tL(1)}) \wedge \odot r.\texttt{homeLoc}_{write}(v_{21})$
$\quad \wedge \exists\, v_{31}\, \mathsf{Y}(v_{31} = \texttt{tL(2)}) \wedge \odot r.\texttt{nextTarget}_{write}(v_{31}) \wedge \ldots$
$\quad \wedge(\mathsf{Y}(\exists\, v_{42}\, v_{42} = true) \Rightarrow \exists\, v_{41}\, (v_{41} = \texttt{tL(1)} \wedge \mathsf{X}(\odot loc.\lambda_{write}(v_{41})))))$

or even further to

$r.(\odot r.\texttt{launchRetrieval(kw, tL, resultSet)} \Rightarrow$
$\quad \odot r.\texttt{keywords}_{write}(\texttt{kw})$
$\quad \wedge \odot\, r.\texttt{homeLoc}_{write}(\texttt{tL(1)})$
$\quad \wedge \odot\, r.\texttt{nextTarget}_{write}(\texttt{tL(2)}) \wedge \ldots$
$\quad \wedge \mathsf{X}(\odot loc.\lambda_{write}(\texttt{tL(1)})))$.

Provided that

1. *within one state*

   - a *new* value can be assigned to an *attribute* (= "set", $\odot object.attribute_{write}$) but the *old* one is accessed—which corresponds to reading the value in the state immediately preceding the one under consideration (= "use", $\triangleright object.attribute_{read}$), but that
   - a local *variable* has only one (set and used) value,

2. in *state* $n+1$ via $\mathsf{Y}(\triangleright object.attribute_{read}(\ldots))$ the value is read which has been set in *state* n by $\odot object.attribute_{write}(\ldots)$,

3. there are four `LiteratureHosts` `litHost1, litHost2, litHost3, litHost4` nested at the same level within the `litSys` instance of the `LiteratureSystem`,

4. `launchRetrieval(...)` sets `tL(1) = /DocumentRetrieval/litSys/litHost1`, `tL(2) = /DocumentRetrieval/litSys/litHost2` and

5. `determineNextTarget(someLoc, nextLoc)` sets `nextLoc` to the value given in its column in the following table for the value of `someLoc` on the same line:

someLoc	nextLoc
none	/DocumentRetrieval/litSys/litHost2
/DocumentRetrieval/litSys/litHost1	/DocumentRetrieval/litSys/litHost2
/DocumentRetrieval/litSys/litHost2	/DocumentRetrieval/litSys/litHost3
/DocumentRetrieval/litSys/litHost3	/DocumentRetrieval/litSys/litHost4
/DocumentRetrieval/litSys/litHost4	/DocumentRetrieval/litSys/litHost1

and using the abbreviations

A: /DocumentRetrieval/litSys/litHost1

B: /DocumentRetrieval/litSys/litHost2

C: /DocumentRetrieval/litSys/litHost3

D: /DocumentRetrieval/litSys/litHost4

I: /DocumentRetrieval/litSys/retrievalUnit

α: $\odot r$.currentLoc$_{write}$(nxtLocBuf),

β: $\exists v_{51}\mathsf{Y}(\triangleright r$.currentLoc$_{read}(v_{51}))\wedge\odot r$.determineNextTarget($v_{51}$, nxtLocBuf),

γ: $\odot r$.nextTarget$_{write}$(nxxtTargetBuf),

δ: $\exists v_{61}\mathsf{Y}(\triangleright r$.nextTarget$_{read}(v_{61}))\wedge\odot r$.determineNextTarget($v_{61}$, nxxtTargetBuf),

ϵ: $\mathsf{Y}(\exists\, v_{72}, v_{73}, v_{74}$
$(\triangleright r$.currentLoc$_{read}(v_{73}) \wedge \triangleright r$.homeLoc$_{read}(v_{74}) \wedge v_{72} = (v_{73} \neq v_{74}))) \Rightarrow$
$\exists v_{71}(\mathsf{Y}(\triangleright r$.nextTarget$_{read}(v_{71})) \wedge \mathsf{X}(\odot loc.\lambda_{write}(v_{71}))),$

we obtain the "migration pattern" shown in table A.1.

λ use/set	Loc. of mob. agent	State	nextTarget use/set	homeLoc use/set	nxtTargetBuf	currentLoc use/set	nxtLocBuf
I	?	init	? / B	? / A	?	? / none	?
[def. 6.50]			- / [tL(2)]	- / [tL(1)]			
↓ [Migration because of $\mathsf{Y}(\exists\, v_{42}\; v_{42} = true) \Rightarrow \exists\, v_{41}\; (v_{41} = \texttt{tL(1)} \wedge \mathsf{X}(\odot loc.\lambda_{write}(v_{41})))$ from launchRetrieval(...)]							
I / A	A	1	B / C	A / -	C	none / B	B
			[set(nextTarget)@init] / [γ]	[set(homeLoc)@init] / -	[δ]	[set(currentLoc)@init] / [α]	[β]
		↓ [ϵ]					
A / B	B	2	C / D	A / -	D	B / C	C
			[set(nextTarget)@1] / [γ]		[δ]	[set(currentLoc)@1] / [α]	[β]
		↓ [ϵ]					
B / C	C	3	D / A	A / -	A	C / D	D
			[set(nextTarget)@2] / [γ]		[δ]	[set(currentLoc)@2] / [α]	[β]
		↓ [ϵ]					
C / D	D	4	A / B	A / -	B	D / A	A
			[set(nextTarget)@3] / [γ]		[δ]	[set(currentLoc)@3] / [α]	[β]
		↓ [ϵ]					
D / A	A	5	B / C	A / -	C	A / B	B
			[set(nextTarget)@3] / [γ]		[δ]	[set(currentLoc)@3] / [α]	[β]
use(currentLoc)@5 = use(homeLoc)@5 !							

Table A.1: "Migration pattern" of the mobile agent from the sample specification, assuming four LiteratureHosts litHost1, litHost2, litHost3, litHost4 nested at the same level within the litSys instance of the LiteratureSystem. (Rationales for attribute and variable values are given in square brackets.)

Appendix B

Complete Grammar for MOBILE TROLL

The syntax is given using *Extended Backus Naur Form* (EBNF), i.e.

- alternatives (which are always exclusive) are separated by a vertical bar (|),
- optional details are included in square brackets ([,]),
- elements enclosed in between curly braces ({, }) have arbitrary cardinality, i.e. may occur none, one, or several times,
- terminal symbols are printed in **bold** font or are enclosed in single quotes (‘,’), and
- non-terminal symbols are written between angular brackets (<, >).

Comments in a TROLL specification may be given any time, either as line comments extending from two leading slashes ‘//’ until the end of line, or possibly spanning several lines between an opening ‘/∗’ and a closing ‘∗/’.

Keywords

actions	**all**	**and**	**any**	**aspect of**
attributes	**bag**	**behavior**	**bool**	**char**
cnt	**components**	**constant**	**constraints**	**data type**
date	**def**	**derived**	**div**	**do**
dom	**elem**	**else**	**empty**	**end**
end_mobile	**end_stationary**	**enum**	**false**	**fi**
forEach	**from**	**head**	**hidden**	**if**
implies	**in**	**initialized**	**initially**	**int**
isA	**link**	**list**	**map**	**mk-**
mod	**mobile**	**mobiles**	**money**	**moveTo**
nat	**not**	**num**	**object class**	**objects**
object system	**od**	**offer_object**	**once**	**onEntering**
onEntry	**onExit**	**onExiting**	**onlyIf**	**optional**
or	**real**	**record**	**request_object**	**rng**
select	**set**	**stationaries**	**stationary**	**string**
sublist	**tail**	**then**	**toSet**	**true**
unlink	**var**	**where**	**xor**	

Identifiers and Constants

<*letter*> ::= **A** | ... | **Z** | **a** | ... | **z**

<*number*> ::= **0** | ... | **9**

<*ident*> ::= <*letter*> { <*letter*> | <*number*> | '_' }

<*charConst*> ::= ' ' *asciichar* ' '

<*stringConst*> ::= '" ' { *asciichar* } '" '

<*natConst*> ::= <*number*> { <*number*> }

<*intConst*> ::= ['-'] <*natConst*>

<*moneyConst*> ::= <*natConst*> '.' <*natConst*> ['E' ['+' | '-'] <*natConst*>]

<*realConst*> ::= ['-'] <*moneyConst*>

<*boolConst*> ::= **true** | **false**

<*dateConst*> ::= '[' <*natConst*> ',' <*natConst*> ',' <*natConst*> ']'

Data Types

<*type*> ::= <*ident*> | |<*ident*>| | **enum**(<*ident*> { ',' <*ident*> }) |
set(<*type*>) | **list**(<*type*>) | **bag**(<*type*>) |
record(<*field*> { ',' <*field*> }) |
record(<*field*> { ';' <*field*> }) | **map**(<*field*>, <*type*>) |
bool | **char** | **date** | **int** | **nat** | **money** | **real** | **string**

<*dataTypeSpec*> ::= **data type** <*ident*> '=' <*type*>

<*field*> ::= [<*ident*> ':'] <*type*>

Variables

<*variableDecl*> ::= **var** <*variable*> { ',' <*variable*> }

<*variable*> ::= <*ident*> **:** <*type*>

Data Terms

<*constTerm*> ::= <*ident*> | <*natConst*> | <*intConst*> | <*realConst*> |
<*stringConst*> | <*charConst*> | <*boolConst*> |
<*dateConst*> | <*moneyConst*>

<*mapLet*> ::= (<*dataTerm*>,<*dataTerm*>)

<*constructor*> ::= **mk-list**(<*dataTerm*> { ‘,’<*dataTerm*> }) |
mk-set(<*dataTerm*> { ‘,’<*dataTerm*> }) |
mk-bag(<*dataTerm*> { ‘,’<*dataTerm*> }) |
mk-record(<*dataTerm*> { ‘,’<*dataTerm*> }) |
mk-<*ident*>(<*dataTerm*> { ‘,’<*dataTerm*> }) |
mk-map(<*mapLet*> { ‘,’<*mapLet*> }) |
mk-<*ident*>(<*mapLet*> { ‘,’<*mapLet*> })

<*relation*> ::= ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘=’ | ‘#’ | **in**

<*boolOp*> ::= **or** | **and** | **implies** | **xor**

<*infixOp*> ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | **div** | **mod** | **isA** |
<*boolOp*> | <*relation*>

<*prefixOp1*> ::= ‘-’ | **head** | **tail** | **cnt** | **toSet** | **rng** | **dom**

<*prefixOp2*> ::= **def** | **num** | **elem**

<*condTerm*> ::= ‘[’ <*pFormula*> ‘?’ <*dataTerm*> ‘:’ <*dataTerm*> ‘]’

<*selectTerm*> ::= **select** <*dataTerm*> **from** <*rangeDecl*> [**where** <*pFormula*>]

<*dataTerm*> ::= <*ident*> | <*constTerm*> | (<*dataTerm*>) |
<*qualidentTerm*> ‘.’ <*ident*> [(<*dataTerm*>)] |
<*qualidentTerm*> ‘.’ ‘@’ <*natConst*> |
sublist(<*dataTerm*> ‘,’ <*dataTerm*> ‘..’ <*dataTerm*>) |
<*prefixOp1*>(<*dataTerm*>) |
<*prefixOp2*>(<*dataTerm*> ‘,’ <*dataTerm*>) |
<*dataTerm*> <*infixOp*> <*dataTerm*>|
<*constructor*> | <*condTerm*> | <*selectTerm*>

Qualified Identifier Terms

<*qualidentTerm*>::= [<*qualidentTerm*> ‘.’] <*qualidentItem*> |
elem(<*dataTerm*> ‘,’ <*dataTerm*>)

<*qualidentItem*>::= <*ident*> [(<*dataTerm*>)] | ‘@’ <*natConst*>

Range Declarations

<*rangeDecl*> ::= <*ident*> **in** <*dataTerm*> { ',' <*ident*> **in** <*dataTerm*> }

Propositions

<*pFormula*> ::= **not** <*pFormula*> | <*pFormula*> <*boolOp*> <*pFormula*> |
<*dataTerm*> | **all**<*rangeDecl*> (<*pFormula*>) |
(<*pFormula*>) | **any**<*rangeDecl*>(<*pFormula*>)

Object Classes

<*objectClassSpec*> ::= **object class** <*ident*> [<*specialization*>]
[**components** <*componentDecl*> {';' <*componentDecl*> } ';']
[<*signatureDecl*>]
[<*behaviorDef*>]
end

<*specialization*> ::= **aspect of** <*ident*> **if** <*specCondition*> { ',' <*specCondition*> }

<*specCondition*> ::= <*specAction*> [**and** <*pFormula*>]

<*specAction*> ::= <*ident*> [(<*ident*> { ',' <*ident*> })]

Signature Declaration

<*signatureDecl*> ::= [**attributes**<*attributeDecl*> { ';' <*attributeDecl*> }';']
[**actions**<*actionDecl*> { ';' <*actionDecl*> }';']

<*attributeDecl*> ::= <*variable*> [<*attributeDesc*> { ',' <*attributeDesc*> }]

<*attributeDesc*> ::= **hidden** | **constant** | **optional** |
derived <*dataTerm*> | **initialized** <*constTerm*>

<*componentDecl*> ::= <*ident*> [(<*field*>)] **:** <*ident*> [**once**] [**hidden**]

<*actionDecl*> ::= [*|+] <*actionSignature*> [**hidden**]

<*actionSignature*> ::= <*ident*> [(<*parameter*> { ',' <*parameter*> })]

<*parameter*> ::= [**!**]<*field*>

Behaviour Definition

<*behaviorDef*> ::= [**behavior**<*operationDef*> { ';' <*operationDef*> }';']
[<*constraintDef*>**;**]

<*operationDef*> ::= <*actionTerm*>
[**onlyIf**<*pFormula*>]
[<*variableDecl*>]
[**do** <*actionRule*> **od**]

<*actionTerm*> ::= { <*qualident*> '.' } <*ident*> [(<*ident*> { ',' <*ident*> })]

<*qualident*> ::= <*ident*> [(<*qualident*>)]

<*actionRule*> ::= <*valuation*> | <*movement*> | <*callTerm*> |
<*repetitiveRule*> | <*conditionalRule*> |
<*actionRule*> { ',' <*actionRule*> }

<*valuation*> ::= <*assignTerm*> ':=' <*dataTerm*>

<*assignTerm*> ::= [<*qualidentTerm*> '.'] <*ident*> |
<*qualidentTerm*> '.' '@' <*natConst*> |
elem(<*dataTerm*> ',' <*dataTerm*> **)**

<*movement*> ::= **if** <*pFormula*> **moveTo** <*target*>

<*target*> ::= <*ident*> | { '/' <*ident*> } '/' <*ident*>

<*callTerm*> ::= [<*qualidentTerm*> '.'] <*ident*>
[(<*dataTerm*> { ',' <*dataTerm*> })]

<*conditionalRule*> ::= **if**<*pFormula*> **then**<*actionRule*> [**else**<*actionRule*>] **fi**

<*repetitiveRule*> ::= **forEach**<*rangeDecl*>
do <*actionRule*> **od**

<*constraintDef*> ::= **constraints** <*constraintRule*> { ',' <*constraintRule*> } ';'

<*constraintRule*> ::= <*pFormula*> | **initially** <*pFormula*>

System Specification

<*systemSpec*> ::= **object system** <*ident*>
(<*partSpec*> { ';' <*partSpec*> } ';' |
<*specItem*> { ';' <*specItem*> } ';')
end.

<*specItem*> ::= <*dataTypeSpec*> | <*objectClassSpec*> |
<*instanceDecl*> | <*behaviorSpec*>

<*instanceDecl*> ::= **objects** <*ident*>[(<*field*>)] **:**<*ident*> [**once**]

<*behaviorSpec*> ::= **behavior** <*operationDef*> { ';' <*operationDef*> } ';'
end

<*partSpec*> ::= <*statSpec*> | <*statInstanceDecl*> |
<*mobiSpec*> | <*mobiInstanceDecl*> |
<*dataTypeSpec*> | <*behaviorSpec*>

<*statSpec*> ::= **stationary** <*ident*>
(<*statSpec*> | <*mobiSpec*> | <*subSpec*>)
{ ';' <*statSpec*> | <*mobiSpec*> | <*subSpec*> |
<*statInstanceDecl*> | <*mobiInstanceDecl*> }
[**onEntry** { <*linkDef*> ';' } [<*actionTerm*>] **end**]
[**onExit** { <*unlinkDef*> ';' } [<*actionTerm*>] **end**]
end_stationary

<*mobiSpec*> ::= **mobile** <*ident*>
(<*mobiSpec*> | <*subSpec*>)
{ ';' <*mobiSpec*> | <*subSpec*> | <*mobiInstanceDecl*> }
[**onEntering** { <*linkDef*> ';' } [<*actionTerm*>] **end**]
[**onExiting** { <*unlinkDef*> ';' } [<*actionTerm*>] **end**]
end_mobile

<*subSpec*> ::= [<*offerSpec*> { ';' <*offerSpec*> }]
[<*requestSpec*> { ';' <*requestSpec*> }]
[<*specItem*> { ';' <*specItem*> }]

<*offerSpec*> ::= **offer_object** <*ident*>
actions <*actionDecl*> { ';' <*actionDecl*> }';'
[<*behaviorSpec*>]
end

<*requestSpec*> ::= **request_object** <*ident*>
actions <*actionDecl*> { ';' <*actionDecl*> }';'
[<*behaviorSpec*>]
end

<*statInstanceDecl*> ::= **stationaries** <*ident*> **:** <*ident*>

<*mobiInstanceDecl*> ::= **mobiles** <*ident*> **:** <*ident*>

<*linkDef*> ::= **link** <*ident*> '.' <*actionSignature*>

<*unlinkDef*> ::= **unlink** <*ident*> '.' <*actionSignature*>

Bibliography

[AA92] D. Agrawal and A. E. Abbadi. Resilient logical structures for efficient management of replicated data. In L.-Y. Yuan, editor, *Proc. 18th Int. Conf. on Very Large Data Bases*, pages 151–162, 1992. Online available at http://www.vldb.org/conf/1992/P151.PDF (May 7, 2002).

[AB96] A. Asperti and N. Busi. Mobile Petri nets. Technical Report UBLCS-96-10, Department of Computer Science, University of Bologna, 1996. Online available at ftp://cs.unibo.it/pub/techreports/96-10.ps.gz (Mar. 4, 2003).

[AF89] Y. Artsy and R. Finkel. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, 22(9):47–56, 1989. Online available at http://dlib2.computer.org/co/books/co1989/pdf/r9047.pdf (Dec. 17, 2002).

[AGE] Internet. AGENTBUILDER. Agent Construction Tools. Online available via http://www.agentbuilder.com/AgentTools/index.html (June 2, 2003).

[AHK92] R. Alonso, E. M. Haber, and H. F. Korth. A Database Interface for Mobile Computers. In *Proc. IEEE GLOBECOM 92 Workshop on Networking of Personal Communications Applications*, LNCS 1222, 1992. Online available at http://citeseer.nj.nec.com/alonso92database.html (Apr. 22, 2003).

[AK93] R. Alonso and H. F. Korth. Database System Issues in Nomadic Computing. In *Proc. 1993 ACM SIGMOD Int. Conf. on Management of Data*, pages 388–392, 1993. Online available at http://delivery.acm.org/10.1145/180000/170092/p388-alonso.pdf (Feb. 7, 2003).

[Ali02] F. Alimadhi. Mobile internet: Wireless access to web-based interfaces of legacy simulations. Master's thesis, University of Amsterdam, Faculty of Science, Informatic Institute, Section Computational Science, 2002. Online avail-

able at http://www.science.uva.nl/research/pscs/papers/archive/-Alimadhi2002a.pdf (Mar. 6, 2003).

[ALSS03] S. Abeck, P. C. Lockemann, J. Schiller, and J. Seitz. *Verteilte Informationssysteme: Integration von Datenübertragungstechnik und Datenbanktechnik*. dpunkt.verlag, 2003. In German.

[Ama97] R. M. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In *Coordination Languages and Methods. Proc. 2nd Int. Conf., COORDINATION '97*, LNCS 1282, pages 374–391, 1997. Online available at http://citeseer.nj.nec.com/303352.html (July 1, 2001).

[AO98] Y. Aridor and M. Oshima. Infrastructure for Mobile Agents: Requirements and Design. In *Mobile Agents. Proc. 2nd Int. Workshop, MA '98*, LNCS 1477, pages 38–49, 1998. Online available at http://link.springer.de/link/service/series/0558/papers/1477/-14770038.pdf (May 27, 2003).

[AR02] P. Ahlbrecht and J. Röver. Specification and Implementation of Mobile-Agent-Based Data Integration. In *Databases and Information Systems II*, pages 269–283. Kluwer Academic Publisher, 2002.

[BAI93] B. R. Badrinath, A. Acharya, and T. Imielinski. Impact of Mobility on Distributed Computations. *Operating Systems Review*, 27(2):15–20, 1993. Online available at http://citeseer.nj.nec.com/badrinath93impact.html (Mar. 4, 2002).

[BD96] T. Beuter and P. Dadam. Prinzipien der Replikationskontrolle in verteilten Datenbanken. *GI Informatik Forschung und Entwicklung*, 11(4):203–212, 1996. Online available at http://www.informatik.uni-ulm.de/dbis/papers/1996/BeDa96a.ps (May 7, 2001). In German.

[BDM+00] S. Bowers, L. Delcambre, D. Maier, C. Cowan, P. Wagle, D. McNamee, A.-F. le Meur, and H. Hinton. Applying Adaptation Spaces to Support Quality of Service and Survivability. In D. C. Young, editor, *Proceedings of the DARPA Information Survivability Conference & Exposition*, pages 271–286. IEEE Computer Society, January 2000. Online available at http://www.cse.ogi.edu/∼shawn/papers/aasf.pdf (Oct. 30, 2001).

[BGM+99] B. Brewington, R. Gray, K. Moizumi, D. Kotz, G. Cybenko, and D. Rus. Mobile agents in distributed information retrieval. In M. Klusch, editor, *Intelligent Information Agents*, chapter 15, pages 355–395. Springer, 1999. Online available at http://www.cs.dartmouth.edu/∼dfk/papers/brewington:ir.ps.gz (June 12, 2002).

[BHRS98] J. Baumann, F. Hohl, K. Rothermel, and M. Strasser. Mole - Concepts of a Mobile Agent System. *World Wide Web Journal*, 1(3):123–137, 1998.

[BI92] B. R. Badrinath and T. Imielinski. Replication and Mobility. In *Proc. 2nd IEEE Workshop on the Management of Replicated Data*, pages 9–12, 1992. Online available at http://citeseer.nj.nec.com/101577.html (June 21, 2002).

[BKKW03] H. Baumeister, N. Koch, P. Kosiuczenko, and M. Wirsing. Extending Activity Diagrams to Model Mobile Systems. In *NetObjectDays. Proc. Objects, Components, Architectures, Services, and Applications for a Networked World, Int. Conf. NetObjectDays, NODe 2002*, LNCS 2591, pages 278–293, 2003. Online available at http://link.springer.de/link/service/series/0558/papers/2591/-25910278.pdf (May 22, 2003).

[BL98] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems*, 13(4–5):361–372, 1998. Online available at http://www.mosix.org/ftps/mosixhpcc.ps.gz (Dec 9, 2002).

[BMGM94] D. Barbará-Millá and H. Garcia-Molina. Replicated Data Management in Mobile Environments: Anything New Under the Sun? In *Applications in Parallel and Distributed Computing, Proc. IFIP WG10.3 Working Conf. on Applications in Parallel and Distributed Computing*, volume A-44 of *IFIP Transactions*, pages 237–246, 1994. Online available at http://citeseer.nj.nec.com/8165.html (Feb. 7, 2003).

[BNT02] Budiarto, S. Nishio, and M. Tsukamoto. Data Management issues in mobile and peer-to-peer environments. *Data & Knowledge Engineering*, 41(2–3):183–204, 2002.

[Boh01] O. Bohnenberger. Ein java-gestützter erweiterbarer Parser für die Spezifikationssprache TROLL-Kern (TCore). Master's thesis, Technical University Braunschweig, 2001. In German.

[BP98] M. Baldi and G. P. Picco. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In R. Kemmerer, editor, *Proc. 20th Int. Conf. on Software Engineering*, pages 146–155, 1998. Online available at http://citeseer.nj.nec.com/baldi98evaluating.html (Apr. 7, 2003).

[BPW98] A. Bieszczad, B. Pagurek, and T. White. Mobile Agents for Network Management. *IEEE Communications Surveys and Tutorials*, 1(1), 1998. Online available at

http://www.comsoc.org/livepubs/surveys/public/4q98issue/bies.html (Aug. 8, 2001).

[BR98] J. Baumann and K. Rothermel. The Shadow Approach: An Orphan Detection Protocol for Mobile Agents. In *Mobile Agents, Proc. 2nd Int. Workshop, MA '98*, LNCS 1477, pages 2–13, 1998. Online available at http://link.springer.de/link/service/series/0558/papers/1477/-14770002.pdf (Apr. 3, 2003).

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[BW89] A. Barak and R. Wheeler. MOSIX: An Integrated Multiprocessor UNIX. In *Proc. USENIX Winter 1989 Technical Conference.*, pages 101–112, 1989.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[Car99] L. Cardelli. Abstractions for Mobile Computing. In *Secure Internet Programming*, LNCS 1603, pages 51–94. Springer, 1999. Online available at http://www.luca.demon.co.uk/Bibliography.html (Nov. 2, 2001).

[CBS96] T. Connolly, C. Begg, and A. Strachan, editors. *Database Systems*. Addison-Wesley, second edition, 1996.

[CDE+01] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martín-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 2001. To appear. Online available at http://www.maude.csl.sri.com/papers/postscript/-CDELMMQspecprog-2001.ps.gz (Jul. 24, 2002).

[CDK01] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems. Concepts and Design*. Addison-Wesley, third edition, 2001.

[CET] Internet. Cetus Links - Object-Orientation. Distributed Objects & Components: Mobile Agents. Online available via http://www.cetus-links.org/oo_mobile_agents.html (June 2, 2003).

[CG98] L. Cardelli and A. D. Gordon. Mobile Ambients. In *Foundations of Software Science and Computation Structures. Proc. 1st Int. Conf., FoSSaCS '98*, LNCS 1378, pages 140–155, 1998. Online available at http://link.springer.de/link/service/series/0558/papers/1378/-13780140.pdf (June 11, 2002).

[CG00] L. Cardelli and A. D. Gordon. Anytime, Anywhere. Modal Logics for Mobile Ambients. In *Proc. 27th ACM Symp. on Principles of Programming Languages*, 2000. Online available at http://research.microsoft.com/Users/luca/Papers/Anytime-Anywhere.A4.ps (May 7, 2002).

[CGH92] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, TU Braunschweig, 1992.

[CGH+97] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant Agents for Mobile Computing. In M. N. Huhns and M. P. Singh, editors, *Readings in Agents*, pages 267–282. Morgan Kaufmann, 1997. Online available at http://citeseer.nj.nec.com/chess95itinerant.html (May, 26 2003).

[CHK97] D. Chess, C. Harrison, and A. Kershenbaum. Mobile Agents: Are They a Good Idea? In *Mobile Object Systems. Towards the Programmable Internet. Proc. 2nd Int. Workshop, MOS '96. Selected Presentations and Invited Papers.*, LNCS 1222, pages 25–47, 1997. Online available at http://www.research.ibm.com/massive/mobag.ps (Sep. 6, 2001).

[Chr93] P. K. Chrysanthis. Transaction Processing in Mobile Computing Environments. In *Proc. IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 77–83, Princeton, New Jersey, 1993. Online available at http://citeseer.nj.nec.com/chrysanthis93transaction.html (June 21, 2002).

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[CPV97] A. Carzaniga, G. P. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proc. 19th Int. Conf. on Software Engineering.* ACM Press New York, NY, USA, 1997. Online available at http://citeseer.nj.nec.com/carzaniga97designing.html (May 17, 2002).

[CR94] P. K. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994. Online available at http://delivery.acm.org/10.1145/190000/185843/p450-chrysanthis.pdf (June 6, 2002).

[CSS89] J.-F. Costa, A. Sernadas, and C. Sernadas. OBL-89 User Manual (Version 2.3). Internal report, INESC, Lisbon, 1989.

[CWY03] M.-S. Chen, K.-L. Wu, and P. S. Yu. Optimizing Index Allocation for Sequential Data Broadcasting in Wireless Mobile Computing. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):161–173, 2003.

[Dad96] P. Dadam. *Verteilte Datenbanken und Client/Server-Systeme*. Springer, 1996. In German.

[DB96] K. David and T. Benkner. *Digitale Mobilfunksysteme*. Teubner, 1996. In German.

[DELM00] F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In *Agent Systems, Mobile Agents, and Applications. Proc. 2nd Int. Symp. on Agent Systems and Applications and 4th Int. Symp. on Mobile Agents, ASA/MA 2000*, LNCS 1882, pages 73–85, 2000. Similarly online available at http://maude.csl.sri.com/mobile-maude/principlesmm.ps (May 31, 2002).

[Den96] G. Denker. *Verfeinerung in objektorientierten Spezifikationen: Von Aktionen zu Transaktionen*. DISDBIS. infix-Verlag, 1996. In German.

[DH95] M. H. Dunham and A. Helal. Mobile Computing and Databases: Anything New? *SIGMOD Record*, 24(4):5–9, 1995. Online available at http://delivery.acm.org/10.1145/220000/219727/P005.pdf (Mar. 4, 2003).

[DH97] G. Denker and P. Hartel. TROLL – An Object Oriented Formal Method for Distributed Information System Design: Syntax and Pragmatics. Informatik-Bericht 97-03, Technische Universität Braunschweig, 1997. Online available via http://www.cs.tu-bs.de/idb/publications/pub_97.html (Mar. 18, 2004).

[DHB97] M. H. Dunham, A. Helal, and S. Balakrishnan. A Mobile Transaction Model That Captures Both the Data and Movement Behavior. *Mobile Networks and Applications*, 2(2):149–162, 1997. Online available at http://citeseer.nj.nec.com/dunham97mobile.html (May 19, 2003).

[DK99] M. H. Dunham and V. Kumar. Impact of mobility on transaction management. In *Proc. ACM Int. Workshop on Data Engineering for Wireless and Mobile Access*, pages 14–21. ACM, 1999. Online available at http://www.seas.smu.edu/*sim*mhd/pubs/99/mobide1.ps (July 8, 2002).

[DO91] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience*, 21(8):757–785, 1991.

[EAN01] S. Eckstein, P. Ahlbrecht, and K. Neumann. Increasing Reusability in Information Systems Development by Applying Generic Methods. In *Proc. 13th Int. Conf. on Advanced Information Systems Engineering (CAiSE'01)*, LNCS 2068, pages 251–266, 2001. Online available at http://www.springerlink.com/media/n2v6e3309j6vrj9cvkwp/-Contributions/D/3/B/P/D3BPTVD4EQ3XQJJ7.pdf (Mar. 19, 2004).

[EC00] H.-D. Ehrich and C. Caleiro. Specifying Communication in Distributed Information Systems. *Acta Informatica*, 36(8):591–616, 2000. Online available at http://www.springerlink.com/media/eafy7be53j6qupfeqwtm/-Contributions/3/B/9/4/3B94J75EHVCVM0L5.pdf.

[Eck01] S. Eckstein. *Module für verteilte Objektsysteme — Konzepte zur Strukturierung und Wiederverwendung objektorientierter Spezifikationen*. DISDBIS. infix-Verlag, 2001. In German.

[ECSD98] H.-D. Ehrich, C. Caleiro, A Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 167–198. Kluwer Academic Publishers, 1998. Online available at http://www.cs.tu-bs.de/idb/publications/conc.ps.gz.

[EDS93] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing Systems as Object Communities. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. Theory and Practice of Software Development (TAPSOFT'93)*, LNCS 668, pages 453–467, 1993.

[EGL89] H.-D. Ehrich, M. Gogolla, and U. Lipeck. *Algebraische Spezifikation abstrakter Datentypen*. Teubner, 1989. In German.

[EH96] H.-D. Ehrich and P. Hartel. Temporal Specification of Information Systems. In A. Pnueli and H. Lin, editors, *Logic and Software Engineering, Proc. Int. Workshop in Honor of C.S. Tang*, pages 43–71, 1996. Online available at http://www.cs.tu-bs.de/idb/publications/beij95.ps.gz (Mar. 18, 2004).

[Ehr99] H.-D. Ehrich. Object Specification. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, pages 435–465. Springer, 1999.

[ELZ88] D. L. Eager, E. D. Lazowska, and J. Zahorjan. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. In *Proc. 1988 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 63–72, 1988. Online available at http://delivery.acm.org/10.1145/60000/55604/p63-eager.pdf (Oct. 17, 2002).

[EN00] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley, third edition, 2000.

[ES95] H.-D. Ehrich and A. Sernadas. Local Specification of Distributed Families of Sequential Objects. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Types Specification, Proc. 10th Workshop on Specification of Abstract Data Types joint with the 5th COMPASS Workshop, S.Margherita, Italy, May/June 1994, Selected papers*, LNCS 906, pages 219–235, 1995. Online available at http://www.cs.tu-bs.de/idb/publications/es95.ps.gz.

[ESDI93] Lisbon Espírito Santo Data Informática. Oblog case v1.0 - the user's guide. Technical report, ESDI, Av. Alvares Cabral 41-5, 1200 Lisbon, 1993.

[ESS88] H.-D. Ehrich, A. Sernadas, and C. Sernadas. Abstract Object Types for Databases. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, pages 144–149, 1988.

[ESS89] H.-D. Ehrich, A. Sernadas, and C. Sernadas. Objects, Object Types, and Object Identification. In H. Ehrig, H. Herrlich, H.-J. Kreowski, and G. Preuß, editors, *Categorical Methods in Computer Science*, LNCS 393, Springer, Berlin, pages 142–156, 1989.

[ESS90] H.-D. Ehrich, A. Sernadas, and C. Sernadas. From Data Types to Object Types. *Journal on Information Processing and Cybernetics EIK*, 26(1-2):33–48, 1990.

[FG97] S. Franklin and A. Grasser. Is It an Agent, or Just a Program?: A Taxonomy for Autonomous Agents. *LNCS*, 1193:21–35, 1997.

[FGL+96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A Calculus of Mobile Agents. In *CONCUR '96: Concurrency Theory. Proc. 7th Int. Conf.*, LNCS 1119, pages 406–421, Pisa, Italy, 1996.

[FIP] Internet. FIPA. The Foundation for Intelligent Physical Agents. Online available via http://www.fipa.org (Apr. 30, 2003).

[Fla97] D. Flanagan. *Java in a Nutshell.* O'Reilly, 1997.

[FPV98] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering. Special Issue - Mobility and Network-Aware Computing*, 24(5):342–361, 1998. Online available at http://ieeexplore.ieee.org/iel4/32/15047/00685258.pdf (July 17, 2003).

[FRHF03] B. Fong, P. B. Rapajic, G. Y. Hong, and A. C. M. Fong. Factors Causing Uncertainties in Outdoor Wireless Wearable Communication. *IEEE Pervasive Computing*, 2(2):16–19, 2003.

[FZ94] G. H. Forman and J. Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, 27(4):38–47, 1994. Online available at http://dlib2.computer.org/co/books/co1994/pdf/r4038.pdf (Dec. 13, 2002).

[FZ95] M. Faiz and A. Zaslavsky. Database Replica Management Strategies in Multidatabase Systems with Mobile Hosts. In *Proc. 6th Int. Hong Kong Computer Society Database Workshop*, pages 305–318, March 1995. Online available at http://citeseer.nj.nec.com/faiz95database.html (May. 29, 2002).

[GBE+00] R. H. Güting, M. H. Böhlen, M. Erwig, N. A. Lorentzos, C. S. Jensen, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems (TODS)*, 25(1):1–42, 2000. Online available at http://delivery.acm.org/10.1145/360000/352963/p1-guting.pdf (May 13, 2003).

[GCH93] M. Gogolla, S. Conrad, and R. Herzig. Sketching Concepts and Computational Model of TROLL *light*. In A. Miola, editor, *Proc. 3rd Int. Conf. Design and Implementation of Symbolic Computation Systems (DISCO'93)*, LNCS 722, pages 17–32, 1993.

[Gei01] M. Geier. *Fragmentierte Objekte für die Implementierung mobiler Agenten*, volume 34 of *Arbeitsberichte des Instituts für Informatik*. Universität Erlangen-Nürnberg, Institut für Informatik, 2001. In German.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GHOS96] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In H. V. Jagadish and I. S. Mumick, editors, *Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data*, pages 173–182. ACM Press, 1996. Online available at http://citeseer.nj.nec.com/gray96danger.html (Mar. 14, 2002).

[GKK+98] A. Grau, J. Küster Filipe, J. Kowsari, S. Eckstein, R. Pinger, and H.-D. Ehrich. The TROLL Approach to Conceptual Modelling: Syntax, Semantics and Tools. In T. W. Ling, S. Ram, and M. L. Lee, editors, *Proc. 17th Int. Conf. on Conceptual Modeling (ER'98)*, LNCS 1507, pages 277–290, 1998.

[GKP+01] R. S. Gray, D. Kotz, R. A. Peterson, J. Barton, D. Chacón, P. Gerken, M. Hofmann, J. Bradshaw, M. Breedy, R. Jeffers, and N. Suri. Mobile-Agent versus Client/Server Performance: Scalability in an Information-Retrieval Task. In *Mobile Agents. Proc. 5th Int. Conf., MA 2001*, LNCS 2240, pages 231–243, 2001.

[GM01] V. Grassi and R. Mirandola. UML Modelling and Performance Analysis of Mobile Software Architectures. In *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools. Proc. 4th Int. Conf.*, LNCS 2185, pages 209–224, 2001. Online available at http://link.springer.de/link/service/series/0558/papers/2185/-21850209.pdf (May 22, 2003).

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In U. Dayal and I. L. Traiger, editors, *Proc. ACM Special Interest Group on Management of Data 1987 Annual Conf.*, pages 249–259. ACM Press, 1987.

[Goo97] D. J. Goodman. *Wireless Personal Communication Systems*. Addison-Wesley, 1997.

[Gra01] A. Grau. *Computer-Aided Validation of Formal Conceptual Models*. PhD thesis, Technical University Braunschweig, 2001. Online available at http://www.biblio.tu-bs.de/ediss/data/20010403a/20010403a.html (June 27, 2002).

[Gro02] W. Grosso. *Java RMI*. O'Reilly, 2002.

[GS01] J. Gomoluch and M. Schroeder. Information Agents on the Move: A Survey on Load-balancing with Mobile Agents. *Software Focus*, 2(2):31–36, 2001. Online available at http://www.soi.city.ac.uk/∼msch/abstracts/sch_swfocus.html (Feb. 12, 2002).

[HAA00] J. Holliday, D. Agrawal, and A. E. Abbadi. Planned Disconnections for Mobile Databases. In *Proc. 11th Int. Workshop on Database and Expert Systems Applications (DEXA'00)*, 2000. Online available at http://citeseer.nj.nec.com/holliday97planned.html (May 17, 2002).

[Har96] T. Hartmann. *Entwurf einer Sprache für die verhaltensorientierte konzpetionelle Modellierung von Informationssystemen*. DISDBIS. infix-Verlag, 1996. In German.

[Har97] P. Hartel. *Konzeptionelle Modellierung von Informationssystemen als verteilte Objektsysteme*. DISDBIS. infix-Verlag, 1997. In German.

[HDK+97] P. Hartel, G. Denker, M. Kowsari, M. Krone, and H.-D. Ehrich. "Information systems modelling with TROLL — formal methods at work". *Information Systems*, 22(2–3):79–99, 1997.

[HK99] M. Hitz and G. Kappel. *UML@Work*. dpunkt, 1999. In German.

[HKRS02] M. Hitz, G. Kappel, W. Retschitzegger, and W. Schwinger. Ein UML-basiertes Framework zur Modellierung ubiquitärer Web-Anwendungen. *Wirtschaftsinformatik*, 3(44):225–235,

2002. In German. Similarly online available at ftp://ftp.ifs.uni-linz.ac.at/pub/publications/2002/0302.pdf (July 16, 2003).

[Hoh97] F. Hohl. An Approach to Solve the Problem of Malicious Hosts. Fakultätsbericht (Technical Report) 1997/03, TR-1997-03, Universität Stuttgart, Fakultät Informatik, 1997. Online available at ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-1997-03/TR-1997-03.ps (May 14, 2003).

[HSJ+94] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. Revised Version of the Modelling Language TROLL (Version 2.0). Informatik-Bericht 94-03, Technische Universität Braunschweig, 1994.

[HSW94] Y. Huang, P. Sistla, and O. Wolfson. Data Replication for Mobile Computers. In *Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data*, pages 13–24, 1994. Online available at http://delivery.acm.org/10.1145/200000/191845/p13-huang.pdf (Apr. 8, 2003).

[Hun98] J. Hunter. *Java Servlet Programming*. O'Reilly, 1998.

[IB92] T. Imielinski and B. R. Badrinath. Querying in highly mobile distributed environments. In *Proc. 18th Int. Conf. on Very Large Databases*, pages 41–52, 1992. Online available at http://www.cs.rutgers.edu/dataman/papers/vldb92.pdf (June 21, 2002).

[IB94] T. Imielinski and B. R. Badrinath. Mobile Wireless Computing: Challenges in Data Management. *Communications of the ACM*, 37(10):18–28, 1994. Online available at http://delivery.acm.org/10.1145/200000/194317/p18-imielinski.pdf (Mar. 4, 2003).

[IH99] L. Ismail and D. Hagimont. A Performance Evaluation of the Mobile Agent Paradigm. In *Proc. 1999 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 306–313. ACM Press, 1999. Online available at http://delivery.acm.org/10.1145/330000/320415/p306-hagimont.pdf (Jan. 6, 2003).

[IK96] T. Imielinski and H. F. Korth. *Mobile Computing*. Kluwer Academic Publisher, 1996.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988. Similarly online available at http://citeseer.nj.nec.com/jul88finegrained.html (May 17, 2002)).

[Joh98] D. Johansen. Mobile agent applicability. In K. Rothermel and F. Hohl, editors, *Mobile Agents. Proc. 2nd Int. Workshop, MA'98*, LNCS 1477, pages 80–98. Springer, 1998. Online available at http://link.springer.de/link/service/series/0558/papers/1477/-14770080.pdf (Jan 16, 2003).

[JRS95] D. Johansen, R. van Renesse, and F. B. Schneider. Operating System Support for Mobile Agents. In *Proc. 5th Workshop on Hot Topics in Operating Systems*, pages 42–45, 1995.

[JRS99] D. Johansen, R. van Renesse, and F. B. Schneider. What TACOMA Taught Us. In *Mobility: processes, computers, and agents*, pages 564–566. Addison-Wesley, 1999.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.

[JSS91] R. Jungclaus, G. Saake, and C. Sernadas. Formal Specification of Object Systems. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'91, Brighton*, LNCS 494, pages 60–82, 1991.

[JWH97] A. Joshi, S. Weerawarana, and E. N. Houstis. On Disconnected Browsing of Distributed Information. In *Proc. 7th IEEE Int. Workshop on Research Issues in Data Engineering*, 1997.

[KF00] J. Küster Filipe. *Foundations of a Module Concept for Distributed Object Systems*. PhD thesis, Technical University Braunschweig, 2000. Online available at http://www.biblio.tu-bs.de/ediss/data/20001123a/20001123a.html (June 27, 2002).

[KGN+97] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. Agent TCL: Targeting the Needs of Mobile Computers. *IEEE Internet Computing*, 1(4):58–67, 1997. Online available at http://dlib2.computer.org/ic/books/ic1997/pdf/w4058.pdf (Dec 13, 2002).

[KGT99] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest Neighbor Queries in a Mobile Environment. In *Spatio-Temporal Database Management, Int. Workshop STDBM'99*, LNCS 1678, pages 119–134, 1999. Online available at http://link.springer.de/link/service/series/0558/papers/1678/-16780119.pdf (May 5, 2003).

[KKH+96] M. Krone, M. Kowsari, P. Hartel, G. Denker, and H.-D. Ehrich. Developing an Information System Using TROLL: an Application Field Study. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *Proc. 8th Int. Conf. on Advanced Information Systems Engineering (CAiSE'96)*, LNCS 1080, pages 136–159, 1996.

[KP99] A. Küpper and A. S.-B. Park. Stationary vs. Mobile User Agents in Future Mobile Telecommunication Networks. In *Mobile Agents. Proc. 2nd Int. Workshop, MA '98*, LNCS 1477, pages 112–123, 1999. Online available at http://link.springer.de/link/service/series/0558/papers/1477/-14770112.pdf (May 30, 2003).

[KPDS02] V. Kumar, N. Prabhu, M. H. Dunham, and A. Y. Seydim. TCOT-A Timeout-Based Mobile Transaction Commitment Protocol . *IEEE Transactions on Computers*, 51(10):1212–1218, 2002. Online available at http://citeseer.nj.nec.com/543659.html (Apr. 23, 2003).

[KRSW01] C. Klein, A. Rausch, M. Sihling, and Z. Wen. Extension of the Unified Modeling Language for Mobile Agents. In K. Siau and T. Halpin, editors, *Unified Modeling Language: System Analysis, Design and Development Issues*, chapter 8, pages 116–128. Idea Publishing Group, 2001. Online available at http://www4.in.tum.de/ rausch/publications/2001/MobileUML.pdf (Oct. 21, 2001).

[KSM+99] K. Kato, Y. Someya, K. Matsubara, K. Toumura, and H. Abe. An Approach to Mobile Software Robots for the WWW. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):526–548, 1999. Online available at http://dlib2.computer.org/tk/books/tk1999/pdf/k0526.pdf (Jan 16, 2003).

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994. Online available at http://delivery.acm.org/10.1145/180000/177726/p872-lamport.pdf (July 7, 2003).

[LEO] Internet. LEO—Link Everything Online. An Online Service by Informatik der Technischen Universität München. Online available at http://www.leo.org/ (June 2, 2004).

[LEW96] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. John Wiley & Sons, Ltd. and B. G. Teubner, 1996.

[LO98] D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[LO99] D. B. Lange and M. Oshima. Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3):88–89, 1999. Online available at http://www.moe-lange.com/danny/7reasons.pdf (Sep. 6, 2001).

[LS92] M. Litzkow and M. Solomon. Supporting Checkpointing and Process Migration outside the UNIX Kernel. In *Proc. USENIX Winter Conference*, pages 283–290, 1992.

[LSZ99] S. W. Loke, H. Schmidt, and A. Zaslavsky. Programming the Mobility Behaviour of Agents by Composing Itineraries. In *Advances in Computing Science — ASIAN '99. Proc. 5th Asian Computing Science Conf.*, LNCS 1742, pages 214–226, 1999. Online available at http://citeseer.nj.nec.com/loke99programming.html (Jan. 23, 2002).

[Mah02] Q. H. Mahmoud. *Learning Wireless Java*. O'Reilly, 2002.

[MAL] Internet. The Mobile Agent List. Online available via http://mole.informatik.uni-stuttgart.de/mal/mal.html (June 2, 2003).

[MAS] Internet. Mobile Agent System Interoperability Facilities Specification. Object Management Group, OMG TC Document orbos/97-10-05, online available via anonymous ftp from ftp.omg.org/pub/docs/orbos/97-03-09.pdf, revised version OMG Document orbos/98-03-09: ftp.omg.org/pub/docs/orbos/98-03-09.pdf (Mar. 22, 2002).

[Mas99] C. Mascolo. MobiS: A Specification Language for Mobile Systems. In *Coordination Languages and Model. Proc. 3rd Int. Conf., COORDINATION '99*, LNCS 1594, pages 37–54, 1999.

[MDW99] D. Milojičić, F. Douglis, and R. Wheeler. *Mobility: processes, computers, and agents*. Addison-Wesley, 1999.

[MG01] F. Muscutariu and M.-P. Gervais. On the Modeling of Mobile Agent-Based Systems. In *Mobile Agents for Telecommunication Applications. Proc. 3rd Int. Workshop, MATA 2001*, LNCS 2164, pages 219–233, 2001. Online available at http://link.springer.de/link/service/series/0558/papers/2164/-21640219.pdf (June 15, 2003).

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil01] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, second edition, 2001.

[MOC] Internet. Moca. Calculi for Mobile Processes. Online available via http://lampwww.epfl.ch/mobility/ (July 25, 2003).

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I/II. *Information and Computation*, 100(1):1–77, 1992. Online available at http://lampwww.epfl.ch/mobility/papers/ECS-LFCS-89-85—6.ps (Aug. 7, 2002).

[MR98] P. J. McCann and G.-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, 1998. Online available at http://ieeexplore.ieee.org/iel4/32/14664/00666824.pdf (July 17, 2003).

[MRK96] T. Magedanz, K. Rothermel, and S. Krause. Intelligent Agents: An Emerging Technologie for Next Generation Telecommunications? In *5th Annual Joint Conf. IEEE and Communication Societies*, volume 2, 1996. Online available at http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/Publications.html (Aug. 8, 2001).

[MWC96] A. Massari, S Weissman, and P. K. Chrysanthis. Supporting Mobile Database Access through Query by Icons. *Distributed and Parallel Databases*, 4(3):249–269, 1996. Online available at http://citeseer.nj.nec.com/massari96supporting.html (May 3, 2003).

[MWZ03] S. Merz, M. Wirsing, and J. Zappe. A Spatio-Temporal Logic for the Specification and Refinement of Mobile Systems. In *Fundamental Approaches to Software Engineering. Proc. 6th Int. Conf., FASE 2003*, LNCS 2621, pages 87–101, 2003. Online available at http://link.springer.de/link/service/series/0558/papers/2621/-26210087.pdf (June 15, 2003).

[MZDG93] D. Milojičić, W. Zint, A. Dangel, and P. Giese. Task Migration on the Top of the Mach Microkernel. In *Proc. 3rd USENIX Mach Symp.*, pages 273–290, 1993.

[Nel99] J. Nelson. *Programming Mobile Objects with Java*. Wiley Computer Publishing, 1999.

[NFP98] R. De Nicola, G.-L. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998. Online available at http://ieeexplore.ieee.org/iel4/32/15047/00685256.pdf (July 17, 2003).

[NFP00] R. De Nicola, G.-L. Ferrari, and R. Pugliese. Programming Access Control: The KLAIM Experience. In *CONCUR 2000 - Concurrency Theory, 11th Int. Conf.*, LNCS 1877, pages 48–65, 2000. Online available at http://link.springer.de/link/service/series/0558/papers/1877/-18770048.pdf (July 15, 2003).

[NL00] R. De Nicola and M. Loreti. A Modal Logic for KLAIM. In *Algebraic Methodology and Software Technology. Proc. 8th Int. Conf., AMAST 2000.*, LNCS 1816, pages 339–354, 2000.

[Nob98] B. D. Noble. *Mobile Data Access.* PhD thesis, Carnegie Mellon University, 1998. Online available at http://citeseer.nj.nec.com/noble98mobile.html (Mar. 1, 2002).

[NPS95] B. Noble, M. Price, and M Satyanarayanan. A Programming Interface for Application-Aware Adaptation in Mobile Computing. In *Proc. 2nd USENIX Symp. on Mobile and Location-Independent Computing*, 1995. Online available at http://www.usenix.org/publications/library/proceedings/mob95/-full_papers/noble.ps (May 17, 2003).

[NSN+97] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. 16th ACM Symp. on Operating System Principles*, volume 31 of *Operating Systems Review*, pages 276–287. ACM Press, 1997.

[OMG] Internet. OMG. Unified Modeling Language (UML), version 1.4. Online available via http://www.omg.org (July 21, 2003).

[OV99] M. T. Öszu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, second edition, 1999.

[PB94a] E. Pitoura and B. Bhargava. Revising Transaction Concepts for Mobile Computing. In *Proc. 1st IEEE Workshop on Mobile Computing Systems and Applications (MCSA94)*, pages 164–168, 1994. Online available at http://www.cs.uoi.gr/∼pitoura/distribution/Mobile/mobile94.ps (Jan. 13, 2003).

[PB94b] E. Pitoura and B. K. Bhargava. Building Information Systems for Mobile Environments. In *Proc. 3rd Int. Conf. on Information and Knowledge Management (CIKM'94)*, pages 371–378. ACM, 1994. Online available at http://delivery.acm.org/10.1145/200000/191310/p371-pitoura.pdf (Feb. 5, 2003).

[PB95] E. Pitoura and B. Bhargava. Maintaining Consistency of Data in Mobile Distributed Environments. In *Proc. 15th Int. Conf. on Distributed Computing Systems*, pages 404–413, 1995. Online available at http://www.cs.uoi.gr/∼pitoura/distribution/Mobile/icdcs95.ps (Jan. 13, 2003).

[Per01] C. E. Perkins. *Ad Hoc Networking.* Addison-Wesley, 2001.

[Pet01] M. Petsch. Aktuelle Entwicklungsumgebungen für mobile Agenten und Multiagentensysteme. *Wirtschaftsinformatik*, 2:175–182, 2001. In German.

[PPW02] J. Pichler, R. Plösch, and R. Weinreich. MASIF und FIPA: Standards für Agenten. *Informatik Spektrum*, 25(2):91–100, 2002. In German.

[PS97] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *Proc. 1st Int. Workshop on Mobile Agents, MA '97*, LNCS 1219, pages 50–61, 1997.

[PS01] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*. Kluwer Academic Publisher, second edition, 2001.

[PSP99] S. Papastavrou, G. Samaras, and E. Pitoura. Mobile Agents for WWW Distributed Database Access. In *Proc. 11th IEEE Int. Conf. on Data Engineering (ICDE'95)*, 1999.

[PW95] G. Popek and B. Walker. *The LOCUS Distributed System Architecture*. MIT Press, 1995.

[PW01] R. Plösch and R. Weinreich. Ein agentenbasierter Ansatz für die Ferndiagnose und -überwachung von Automatisierungssystemen. *Wirtschaftsinformatik*, 2:167–173, 2001. In German.

[RC93] P. Rob and C. Coronel, editors. *Database Systems*. Wadsorth, 1993.

[RH97] J. Riely and M. Hennessy. Distributed Processes and Location Failures (Extended Abstract). In *Automata, Languages and Programming. Proc. 24th Int. Colloquium, ICALP '97.*, LNCS 1256, pages 471–481, 1997.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Guide*. Addison-Wesley, 1999.

[RM02] G.-C. Roman and P. J. McCann. A Notation and Logic for Mobile Computing. *Formal Methods in System Design*, 20(1):47–68, 2002. Online available at http://www.cs.wustl.edu/mobilab/Publications/Papers/2002/FMSD-02.pdf (Aug. 19, 2003).

[RMP97] G.-C. Roman, P. J. McCann, and P. Y. Plun. Mobile UNITY: Reasoning and Specification in Mobile Computing. In *ACM Transactions on Software Engineering and Methodology 6*, volume 3, pages 250–282, 1997. Online available at http://swarm.cs.wustl.edu/csg/papers/1997-2.ps.Z (Sep. 5, 2001).

[Rot02] J. Roth. *Mobile Computing: Grundlagen, Technik, Konzepte*. dpunkt.verlag, 2002. In German.

[RPM00] G.-C. Roman, G. P. Picco, and A. Murphy. Software Engineering for Mobility: A Roadmap. In *Proc. Conf. on the Future of Software Engineering*, pages 241–258, 2000. Online available

at http://delivery.acm.org/10.1145/340000/336567/p241-roman.pdf (May 6, 2003).

[RPV95] R. Ramjee, T. F. La Porta, and M. Veeraragahvan. The Use of Network-Based Migrating Agents for Personal Communication Services. *IEEE Personal Communications*, 2(6):62–68, 1995. Online available at http://citeseer.nj.nec.com/ramjee95use.html (May 26, 2003).

[RRH00] A. Ralston, E. D. Reilly, and D. Hemmendinger, editors. *Encyclopedia of Computer Science*. Nature Publishing Group, fourth edition, 2000.

[RS95] J. Ramos and A. Sernadas. A brief introduction to Gnome. Report, Section of Computer Science, Department of Mathematics, Instituto Superior Técnico, Lisbon, 1995.

[Saa93] G. Saake. *Objektorientierte Spezifikation von Informationssystemen*. Teubner, Stuttgart/Leipzig, 1993. Habilitationsschrift. In German.

[Sat96] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Annual ACM Symp. on Principles of Distributed Computing. Proc. 15th Annual ACM Symposium on Principles of Distributed Computing*, LNCS 1222, pages 1–7, 1996. Online available at http://delivery.acm.org/10.1145/250000/248053/p1-satyanarayanan.pdf (Dec. 10, 2002).

[Sat00] I. Satoh. A Formalism for Hierarchical Mobile Agents. In *Proc. Symp. on Software Engineering for Parallel and Distributed Systems (PDSE'2000)*, pages 165–172, 2000. Similarly online available at http://citeseer.nj.nec.com/satoh00formalism.html (Nov. 19, 2001).

[SCH01] F. Siegemund, C. H. Cap, and A. Heuer. Einsatz von mobilen Agenten und XML zur Angebotsrecherche im Business-to-Consumer-Commerce. *Wirtschaftsinformatik*, 2:157–166, 2001. In German.

[SDK01] A. Y. Seydim, M. H. Dunham, and V. Kumar. Location Dependent Query Processing. In *Proc. 2nd ACM Int. Workshop on Data Engineering for Wireless and Mobile Access*, pages 47–53, 2001. Online available at http://delivery.acm.org/10.1145/380000/376895/p47-seydim.pdf (May 5, 2003).

[SE90] A. Sernadas and H.-D. Ehrich. What is an Object, After All? In R. Meersman, W. Kent, and S. Khosla, editors, *Proc. IFIP TC2/WG 2.6 Working Conference on Object-Oriented Databases: Analysis, Design & Construction*, pages 39–69, 1990.

[SH98] P. Smith and N. C. Hutchinson. Heterogeneous process migration: the Tui system. *Software: Practice and Experience*, 28(6):611–639, 1998.

[SHB+99] A. Schill, A. Held, W. Böhmak, T. Springer, and T. Ziegert. An Agent Based Application for Personalized Vehicular Traffic Management. In *Mobile Agents. Proc. 2nd Int. Workshop, MA '98.*, LNCS 1477, pages 99–111, 1999. Online available at http://link.springer.de/link/service/series/0558/papers/1477/-14770099.pdf (May 27, 2003).

[SJ91] G. Saake and R. Jungclaus. Konzeptioneller Entwurf von Objektgesellschaften. In H.-J. Appelrath, editor, *Proc. Datenbanksysteme in Büro, Technik und Wissenschaft BTW'91*, pages 327–343. Informatik-Fachberichte IFB 270, Springer, Berlin, 1991. In German.

[SK97] M. Schönhoff and M. Kowsari. Specifying the Remote controlling of Valves in an Explosion Test Environment. In *Formal Methods Europe, FME'97. Proc. 4th Int. Symp.*, LNCS 1313, pages 201–220, 1997. Online available at http://www.cs.tu-bs.de/idb/publications/schkow97.ps (Mar. 18, 2004).

[SM99] A. Sahai and C. Morin. Enabling a Mobile Network Manager (MNM) Through Mobile Agents. In *Mobile Agents. Proc. 2nd Int. Workshop, MA '98.*, LNCS 1477, pages 249–260, 1999. Online available at http://link.springer.de/link/service/series/0558/papers/1477/-14770249.pdf (May 27, 2003).

[SSE87] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P.M. Stoecker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, pages 107–116, 1987.

[SSG+90] A. Sernadas, C. Sernadas, P. Gouveia, P. Resende, J. Gouveia, and L. Silva. OBLOG — Object-Oriented Logic: An Informal Introduction. Internal report, INESC, Lisbon, 1990.

[ST99] D. Sannella and A. Tarlecki. Algebraic Preliminaries. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, pages 13–30. Springer, 1999.

[Sta03] V. Stanford. Pervasive Computing Goes the Last Hundred Feet with RFID Systems. *IEEE Pervasive Computing*, 2(2):9–14, 2003.

[Sti02] W. Stieler. Handgelenk-PDA mit Palm OS. *c't*, 25:30, 2002.

[Sto02] I. Stojmenović. *Handbook of Wireless Networks and Mobile Computing.* Wiley, 2002.

[SWCD97] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In A. Gray and P. Larson, editors, *Proc. 13th Int. Conf. on Data Engineering*, pages 422–432, 1997. Online available at http://citeseer.nj.nec.com/sistla97modeling.html (Apr. 3, 2003).

[SWP99] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-Independent Communication for Mobile Agents: A Two-Level Architecture. In *Internet Programming Languages. Proc. ICCL'98 Workshop*, LNCS 1686, pages 1–31, 1999.

[SZM94] C. Steketee, W. Zhu, and P. Moseley. Implementation of Process Migration in Amoeba. In *Proc. 14th Int. Conf. on Distributed Computing Systems*, pages 194–201. IEEE Computer Society Press, 1994. Online available at http://citeseer.nj.nec.com/steketee94implementation.html (Dec. 17, 2002).

[Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[Tho79] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979. Online available at http://delivery.acm.org/10.1145/330000/320076/p180-thomas.pdf (Apr. 23, 2003).

[TLC85] M. Theimer, K. Lantz, and D. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proc. 10th ACM Symp. on Operating System Principles*, pages 2–12, 1985.

[TR00] W. Theilmann and K. Rothermel. Optimizing the Dissemination of Mobile Agents for Distributed Information Filtering. *IEEE Concurrency*, 8(2):53–61, 2000. Online available at http://dlib2.computer.org/pd/books/pd2000/pdf/p2053.pdf (Jan. 6, 2003).

[VC98] J. Vitek and G. Castagna. Towards a Calculus of Secure Mobile Computations. In *Proc. IEEE Workshop on Internet Programming Languages*, 1998. Online available at http://citeseer.nj.nec.com/vitek98towards.html (June 15, 2003).

[VC99] J. Vitek and G. Castagna. Seal: A Framework for Secure Mobile Computations. In *Internet Programming Languages. Proc. ICCL'98 Workshop*, LNCS 1686, pages 47–77, 1999.

[Wat94] T. Watson. Application Design for Wireless Computing. In *Proc. 1st IEEE Workshop on Mobile, Computing Systems and Applications*, 1994. Online available at

http://citeseer.nj.nec.com/watson94application.html (June 21, 2002).

[WC97] G. D. Walborn and P. K. Chrysanthis. Pro-motion: management of mobile transactions. In *Proc. 1997 ACM Symp. on Applied computing*, pages 101–108. ACM Press, 1997. Online available at http://doi.acm.org/10.1145/331697.331718 (June 6, 2002).

[Whi96] J. E. White. Telescript Technology: Mobile Agents. In J. M. Bradshaw, editor, *Software Agents*. AAAI/MIT Press, 1996.

[Win82] G. Winskel. Event Structure Semantics for CCS and Related Languages. In *Proc. 9th Int. Colloquium on Automata, Languages and Programming, (ICALP)*, LNCS 140, pages 561–576, 1982.

[WJ95] M. Woolridge and N. Jennings. Agent Theories, Architectures and Languages: A Survey. In *Proc. ECAI-94 Workshop on Agent Theories, Architectures and Languages*, 1995. Online available at http://citeseer.nj.nec.com/woolridge94agent.html (July 25, 2001).

[WMB99] A. Wienberg, F. Matthes, and M. Boger. Modeling Dynamic Software Components in UML. In *UML'99: The Unified Modeling Language - Beyond the Standard. Proc. 2nd Int. Conf.*, LNCS 1723, pages 204–219, 1999. Online available at http://link.springer.de/link/service/series/0558/papers/1723/-17230204.pdf (June 6, 2003).

[WN95] G. Winskel and M. Nielsen. Models for Concurrency. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Semantic Modelling, Handbook of Logic in Computer Science*, volume 4, pages 1–148. Oxford Science Publications, 1995.

[Wol02] O. Wolfson. Moving Objects Information Management: The Database Challenge. In *NGITS. Next Generation Information Technologies and Systems, Proc. 5th Int. Workshop, NGITS 2002*, LNCS 2382, pages 75–89, 2002. Online available at http://link.springer.de/link/service/series/0558/papers/2382/-23820075.pdf (May 5, 2003).

[Yu01] E. Yu. Agent orientation as a modelling paradigm. *Wirtschaftsinformatik*, 2:123–132, 2001.

[YZ94] L. H. Yeo and A. Zaslavsky. Submission of transactions from mobile workstations in a cooperative multidatabase processing environment. In *Proc. 14th Int. Conf. on Distributed Computing Systems*, pages 372–379, 1994. Online available at http://citeseer.nj.nec.com/yeo94submission.html (Jan. 13, 2003).

[Zay87] E. R. Zayas. Attacking the Process Migration Bottleneck. In *Proc. 11th ACM Symp. on Operating System Principles*, Operating Systems Review, pages 13–24. ACM Press, 1987.

[Zei03] A. Zeid. A UML Profile for Agent-Based Development. In *Metainformatics. Proc. Int. Symp. MIS '02*, LNCS 2641, pages 161–170, 2003. Online available at http://link.springer.de/link/service/series/0558/papers/2641/-26410161.pdf (May 22, 2003).

Index