T U M

I N S T I T U T F Ü R I N F O R M A T I K

Modeling Work Flows For Building Context-Aware Applications

Christian Leuxner, Wassiou Sitou, Bernd Spanfelner,
Veronika Thurner, Armin Schneider

TUM-I0913
Juni 09

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-06-I0913-0/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2009

Druck: Institut für Informatik der
Technischen Universität München

Modeling Work Flows For Building Context-Aware Applications*

Christian Leuxner, Wassiou Sitou, Bernd Spanfelner,
Veronika Thurner† and Armin Schneider‡

Technische Universität München, Departement of Informatics
Boltzmannstr. 3, D-85748 Garching, Germany
{leuxner|sitou|spanfeln}@in.tum.de

† University of Applied Sciences – Munich
Lothstr. 34, D-80335 Munich, Germany
thurner@hm.edu

‡ Klinikum rechts der Isar, MITI Research Group
Trogerstr. 26, D-81675 Munich, Germany
armin.schneider@mitigroup.de

June 19, 2009

*This research has been supported by the CAWARFlow project (grant BR 887/21-1) sponsored by the German Research Foundation

Abstract

The consistent specification of reactive, context-aware systems is still a challenging and error-prone task. One reason for this observation is, that those systems are typically involved in complex user interactions and support multi-variant work flows of humans and other technical systems. On the one hand, it is necessary to capture all system functions required for each supported activity of the work flow. On the other hand, interrelations between system functions such as their order of execution or mutual influences have to be considered as well.

The idea of modeling data and control flows occurring within a software-intensive system and its environment by means of a common and precise graphical notation has been present for a couple of years now. Most modeling techniques in practice like, *e.g.*, UML's Activity Diagrams merely offer an 'appealing' graphical syntax without including a precise mathematical interpretation for the behavioral modules – denoted *processes* in the following – exhibiting these data and control flows. This causes ambiguous model interpretations, which cannot be readily resolved. On the other hand, formally founded description techniques like, *e.g.*, Petri Nets can not express aspects like communication, which are relevant for the faithful description of the processes of the sorts arising in computer science. Methodically relevant concepts such as hierarchy and refinement are often not supported by such description techniques. The presented modeling approach incorporates the advantages of both worlds: a graphical representation supporting a modular, hierarchical description in terms of processes, and a formal semantics accurately reflecting the execution of these processes – thus laying the foundation for automatic verification and tool support.

Furthermore, we sketch an elementary development approach to the model-based design of system behavior, which uses different views to structure its design and analysis. It starts with a formal, structured description of a user's work flows to be supported by the system. This structure is exploited for the construction of two complementary views: one that characterizes the relevant usage conditions of the user's work flow in terms of context, and another view describing the required system behavior in that context.

Contents

1 Introduction

Due to their inherent complexity, the consistent specification of many reactive systems is a challenging and error-prone task. Building concurrent, multi-functional and context-aware systems complicates the specification and reasoning about those systems even more. One way to manage this complexity is the usage of formal, structured models for their specification and analysis. Such models describe the considered system characteristics with mathematical rigor and enable a hierarchical decomposition of the system descriptions into smaller parts. Orthogonally to this hierarchy, *views* structure the specification according to a few selected aspects of the system. Both hierarchies and views can be exploited to structure the specification and analysis. A crucial view we emphasize in this paper focuses on the *work flows* to be supported by the system.

The work flows which humans accomplish in their daily business are typical starting points for the development of reactive systems. Consider a complex work flow like, *e.g.*, a surgery. Such a work flow can be described by a structured specification whose building blocks we denote *processes*. Each process models an exemplary, finite part of the overall work flow, in which surgeons and nurses cooperate in order to achieve a certain objective, *e.g.*, the removal of a patient's gallbladder. Such work flows essentially influence how a system supporting these work flows needs to interact with its environment. A useful description technique for building systems on basis of work flows must fulfill several requirements. Among other things, it should be able to accurately reflect *both* kinds of processes in a *uniform* fashion: *(i)* the work flows accomplished by humans in the real world, and *(ii)* the data and control flows of the system supporting the real-world work flows.

Even more important, a useful modeling approach smoothly integrates the resulting system specifications, such as, *e.g.*, the partially defined processes presented here and the totally defined components of a component-based system design, in order to contribute to a integrated, consistent specification of the developed system.

1.1 Contribution

We introduce processes as a formal, structured model for the modular description of concurrent, reactive systems and their environment. The model is formal in that it defines a process as a mathematical object, which can be analyzed. The model is structured in that it permits the hierarchical definition of a process, and that hierarchy can be exploited for structuring the analysis. We formally define the syntax of process descriptions and provide a formal interpretation for process behavior. Processes form the basic building blocks for describing the data and control flows within systems and their environment. They are capsules of behavior that can be composed sequentially, alternatively, and in parallel – arbitrarily nested.

A mathematical semantics for process descriptions not only enables their unambiguous in-

terpretation. It also provides the basis for tool support, thus enabling an efficient development process. Tools can verify requirements on specifications that are too complex to be verified manually by a person. Moreover, they can support the synthesis of design models from process specifications, thus avoiding error-prone, manual translations.

Moreover, we propose an elementary, model-based approach to the specification of (partial) system behavior, which exploits the work flows of users in order to elicit and structure its functional requirements in terms of a *labeled transition system* (*LTS*). Our approach structures the specification of system behavior into three complementary views: The *work flow view* describes the work flows to be supported by the system in terms of a formal, structured model. This model reflects the execution order and the communication between the work flow parts by means of control and data flows, resp. The work flow view structures the description and analysis of two complementary system views, namely the *contextual view* – which describes the characteristic, observable usage conditions of each work flow part in terms of *context*– and the *functional view* – that represents the system functionalities supporting each part of the work flow. By this, we introduce a '*process-oriented*' view for the development of reactive systems, which complements *service-oriented* approaches by concentrating on execution order and communication rather than on functional dependencies.

1.2 Scope

Our approach addresses the late analysis phase within the development process, in which functional requirements are formalized in terms of a LTS. We assume an informal description of the exemplary usage scenarios / user work flows to be given, *e.g.*, in terms of Use Cases [1]. We formalize those descriptions by the model described in Sec. 3. These partial work flow descriptions, denoted *processes*, are used to derive the intended system behavior with the aid of two complementary views. The result is a formal, structured specification of system functions, which is build up in accordance with the views describing a user's work flows and its corresponding usage conditions in terms of context.

1.3 Related Approaches

A variety of description techniques and formalism for describing work flows already exists, which differ in many aspects such as communication, formal foundations, separation between control and data flow, process composition, refinement concepts, hierarchical structuring, and so on. We mention just a few formal approaches which influenced the definition of our description technique the most.

The concept of activating and deactivating processes by means of control points goes back to the control tokens introduced in *Petri Nets* [2] and variants thereof [3]. *Activity Diagrams* as used in the Unified Modeling Language [1] constitute an informal description technique, which also supports the specification of control flow in terms of choice, iteration, and concurrency. Approaches like [4] emerged since their introduction, which formalize the Activity Diagram semantics in terms of existing formalisms such as Petri Nets or by introducing new formalisms.

In contrast to the above approaches, we support process communication via interface variables, since we aim at the construction of reactive systems, for which data flow is fundamental.

The idea of defining process behavior mathematically and structuring processes hierarchically is inspired by the *business process nets* introduced in [5]. However, the specification of behavior in business process nets is restricted to 'conventional' mathematical functions. This appears to be very limiting when describing complex system behavior, since this approach imposes some kind of '*one step per process*' semantics. In addition, we wanted a specification technique that explicitly captures the notion of control flow for enabling a proper composition of processes – a concept not supported in business process nets.

An explicit separation of data and control flow enables such a composition by partitioning the overall system behavior according to *control points*. More precisely, control points structure a complex labeled transition system into modular behavioral parts. Indeed, our approach is essentially inspired by Henzinger's *components* [6] and Schätz's *functions* [7, 8]. Both approaches focus on the construction of reactive systems, thereby providing designers with *disjunctive* and *conjunctive composition* of behavioral modules, which cover sequential executions by handing over activation and parallel executions by exchanging messages. However, in order to cope with the specification of work flows, some design decisions within the above approaches had to be reconsidered. As opposed to the strict black-box view of functions, processes support the internal communication via shared variables which emphasizes their concentration on (system) executions. In order to reflect their exemplary nature, the processes in our approach are *not input enabled* as opposed to the components in [6] and the functions in [8]. Accordingly, the underlying system model, the definition of behavioral composition, and the corresponding appearance of processes differ.

1.4 Outline

This paper is organized as follows. In Sec. 2, we introduce the notion of context and present a medical case study of a minimal-invasive surgery, denoted laparoscopic cholecystectomy. In Sec. 3, we informally introduce our description technique for modeling data and control flows by means of the case study mentioned before. In Sec. 4, we formally introduce the syntax and semantics of processes and process composition, as well as the notion of process hierarchy. In Sec. 5, by means of a running example, we outline an elementary, process-oriented approach for designing context-aware systems on basis of user work flows and contextual information. In Sec. 6, we summarize our work and outline further promising research directions.

2 Preliminaries

In this section, we introduce the notion of context and present a case study which serves as a running example throughout this paper.

2.1 Context

Informally, the *context* of a system constitutes the sufficiently exact characterization of all information in the system environment. An early work considering the role of context in software-intensive systems is provided by Schilit et. al in [9]. More recent work directs to the systematic elicitation of context, and the question which context is relevant for a system under construction. This consideration manifests in an explicit model of a system's context as proposed in [10], in which context is basically categorized into three dimensions characterizing a usage situation: *(i)* the *user*, *(ii)* her *activities*, and *(iii)* the *operational environment*.

Eventually, the notion of context is exploited to unburden the user from some of her direct interactions with the system. More precisely, the system adapts its observable behavior automatically on basis of context (*context adaptation*). A formally founded discussion of *adaptive system behavior* – behavior that not only depends on explicit user inputs, but also on the context of use and the work flows of users as one part of that context – can be found in [11]. Essentially, Broy et al. propose to use the notion of adaptive system behavior always with respect to a user interacting with the system. Then, depending on her perspective, three kinds of adaptive system behavior can be distinguished, denoted *nontransparent adaptive*, *transparent adaptive*, and *diverted adaptive* system behavior.

2.2 Case Study: Laparoscopic Cholecystectomy

We already mentioned in the previous section that we illustrate the application of our modeling approach using a running example from the medical domain. We describe in parts the work flow of a highly standardized, minimal invasive surgery, denoted as *laparoscopic cholecystectomy (lapCHE)* [12], within which a patient's gallbladder is removed in case of inflammations. The medical case study demonstrates the approach's capabilities to model work flows within a system's environment.

We shortly sketch the execution of the considered surgery. The lapCHE can roughly be partitioned into eight individual phases. For illustrating the modeling capabilities of our approach, we concentrate on the fifth phase of the surgery in which the gallbladder is dissected.

Cholecystectomy is performed under general anesthesia. Initially, a small needle is inserted into the peritoneal cavity for inflating the abdomen with carbon dioxide. This provides room for easier viewing and for the surgical manipulations to be performed.

At the same point, a small incision is made and a thin tube, called *trocar*, is inserted (T1) afterwards. Via this first port, the telescope is introduced to visualize the interior of the abdomen. After a test insertion with a hypodermic needle, three other trocars are inserted under view of the laparoscopic camera (T2, T3 and T4).

To get access to the gallbladder, first a retraction device is inserted in T3. The right liver lobe is elevated. The laparoscopic camera is changed from trocar T1 to T2, to provide sufficient view of the surgical field. Finally, a grasping forceps is inserted into T4 and the dissection device in T1. The primary step of the surgical procedure is to dissect the area which includes the bile duct and the cystic artery (Calot's triangle). This is done by blunt dissection with a forceps and cutting current. In case of bleedings, coagulation current is used. If both structures are clearly visible, each of them is clipped with three clips, followed by cutting both structures between the clips with laparoscopic scissors. The following step is dissection of the gallbladder. In minimally invasive surgery, this is done by touching the areas between gallbladder and liver and applying cutting current.

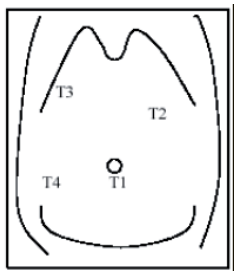

Fig. 2.1: Trocar Points

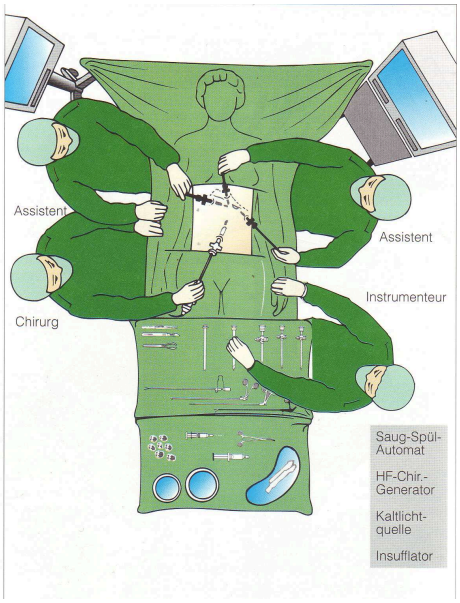

Fig. 2.2: Operating Team Lineup

To remove the dissected gall bladder a salvage bag is inserted into the abdomen, the gallbladder packed up into the bag and the bag extracted together with trocar T1. In case of big stones, the bag cannot be extracted through the trocar incision. In that case, the calculi are extracted extracorporeally out of the salvage bag. Thus, the content of the bag is adequately reduced to pull it out. Finally, the surgical area is explored again to detect and take care of bleedings. A drainage is inserted through a trocar hole and all instruments are removed. The trocars are extracted under visual control and the incisions are closed by sutures. During the procedure, in case of bleedings in the operation field, a device which allows flushing and suction is used. Also controlling for bleedings after extraction of the gallbladder is done with this device.

3 Describing Processes

We informally introduce the syntax and semantics of processes in this section, and present our description technique by giving a work flow specification for a surgical intervention, the lapCHE.

3.1 Process Interface and Behavior

Processes are the building blocks of the approach presented here. Basically, processes are capsules of behavior, defined by their syntactic interface in terms of data and control flow as well as their semantic interface. The semantic interface describes a process's behavior in terms of a constructive specification, *i.e.* the behavior is defined in a state-transition manner. The control flow between the process and its environment is defined in terms of two kinds of control points; one for accepting control from the process's environment and one for returning control back to its environment, thus allowing the process to be activated and deactivated, resp.

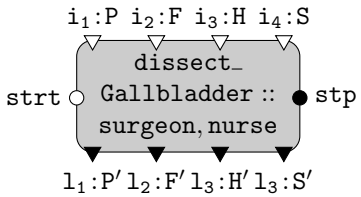

Fig. 3.1: Compound Process '`dissect_Gallbladder`'

Fig. 3.1 shows the graphical representation of a compound process describing a work flow within the surgery mentioned above. Each capsuled behavior is represented by a box, and identified by a process name (`dissect_Gallbladder`). Optionally, a set of role names indicating the process's executing entities may be appended to the process name (`surgeon`, `nurse`). Interface elements such as I/O variables (*e.g.*, $\mathtt{i}_1\mathtt{:P}$) and control points (`strt`, `stp`) are attached to the border of each process; the process's internal structure may be graphically depicted inside the box, or it may be completely abstracted by only exposing a process name as show in Fig. 3.1.

The process `dissect_Gallbladder` observes different signals via the typed input variables $\mathtt{i}_1$ through $\mathtt{i}_4$, and controls different signals via the corresponding output variables $\mathtt{l}_1$ through $\mathtt{l}_4$. More precisely, the process accesses four surgical instruments via the I/O variables, namely the `pe` − `forceps P` via variables $\mathtt{i}_1, \mathtt{l}_1$, the `seizing` − `forceps F` via variables $\mathtt{i}_2, \mathtt{l}_2$, the `surgical aspirator H` via variables $\mathtt{i}_3, \mathtt{l}_3$ and the `surgical rinser S` via variables $\mathtt{i}_4, \mathtt{l}_4$. The input and output signals accessed by a process are indicated by empty and filled triangles at the process's border.

To control the activation and deactivation of a process, it can be entered via the control point `strt` and exited via the control point `stp`. As shown in Fig. 3.1, entry points are indicated by hollow circles, while exit points are indicated by filled circles.

To that end, we sketched how to define a process's *syntactic interface*, *i.e.* we specified how a process can be accessed from the environment. However, we have not yet specified the process's *I/O behavior*, *i.e.* how the process handles signals received on input variables and produces signals sent along output variables. We follow a constructive approach in which the behavior of a process is described in a state-transition manner.

Fig. 3.2 shows an elementary process typically occurring within a usage scenario of an Automated Teller Machine (ATM), namely the process `vrfy_PIN`, verifying the personal identification number (Pin) entered by a bank customer. The *process's control flow* is described via control points `strt`, `scc`, $\texttt{err}_1$, $\texttt{err}_2$ and `lck` as well as labeled transitions between these control points. `strt` constitutes the process's only *entry point*, whereas all other control points may pass the control back to the process's environment. Transitions are influenced by signals observed via input variables and influence signals controlled via output variables. Thus, if *(i)* control is passed to control point `strt`, the Pin value `p` is received via the user input variable `Pin` (`Pin`?`p`), and *(iii)* the same Pin value `p` is received via the card reader input variable `Crd` (`Crd`?`p`), then *(iv)* value `ok` is sent via the output variable `Ack` (`Ack`!`ok`) for acknowledging a correctly entered pin to the user, and *(v)* control is transferred to exit point `scc`, which passes the control back to the environment.

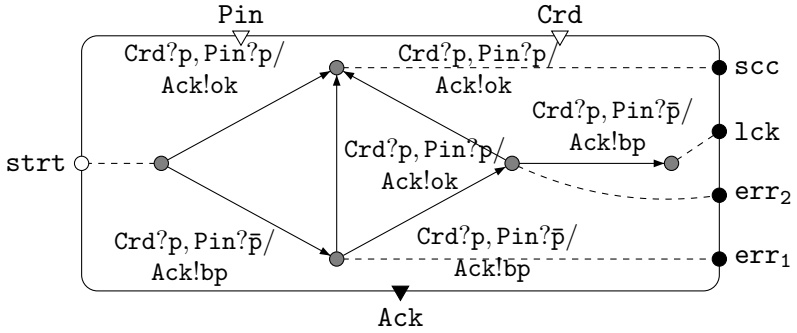

Fig. 3.2: State-Transition Specification of ATM's Process '`vrfy_PIN`'

3.2 Composing Processes

Although at some point in the development process each *elementary* process should have an internal implementation as depicted in Fig. 3.2, it is not further *decomposed* into sub-processes. On the other hand, to decompose a process into further sub-processes, the concept of a *compound process* is introduced, which is graphically indicated by a gray box as shown in Fig. 3.1. Compound processes introduce a hierarchical structure, relating a compound process with its *sub-processes*. This hierarchy is especially useful for modeling complex behaviors, since it introduces an additional level of abstraction by allowing to fold/unfold a work flow description. Thus, it is possible to abstract from irrelevant details and to enhance the readability of large process descriptions.

Moreover, formally defined hierarchies lead to more consistent process specifications, since the sub-process relation imposes additional *syntactic constraints* on the related processes which, *e.g.*, can be statically checked by a CASE tool. A typical constraint is, that the syntactic interface

of a compound process must expose a *hierarchical interface* concerning its sub-processes as defined in Sec. 4.1.6. Informally, this relation implies that the compound process is not allowed to provide any access in terms of I/O variables and control points, which is not supported by its sub-processes.

We describe how to compose and structure work flows for our running example from the medical domain, in which we completely model one phase of the lapCHE surgery. A detailed description of the abstract syntax of process models and the different possibilities for composing processes is given in Sec. 4.

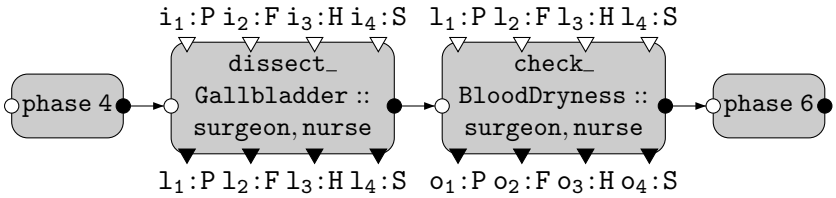

Fig. 3.3: Most Abstract Level of Execution of Surgery Phase 5

Fig. 3.3 depicts the most abstract modeling level of the surgery, *i.e.* all compound processes are folded and abstract from their internal structure and implementation (black-box view). The most abstract view of this surgery phase contains the four compound processes `phase 4`, `dissect_Gallbladder`, `check_BloodDryness`, and `phase 6`, which are all *composed in sequel*. The execution order of the depicted processes is expressed as a relation on the processes's control points – graphically depicted as arrows from exit points to entry points. *E.g.*, when `phase 4` terminates, it immediately transfers the control to process `dissect_Gallbladder`, which – after finishing its own execution – transfers control to process `check_BloodDryness`, and so on.

Processes communicate asynchronously over *shared* variables, which are identified via coincidence of variable names. Graphically, this data flow is not represented by an explicit communication link. As depicted in Fig. 3.3, process `dissect_Gallbladder` and process `check_BloodDryness` communicate via the four shared variables l_1 through l_4.

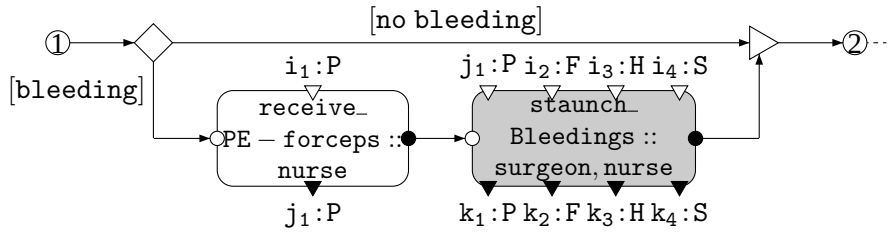

Fig. 3.4: First Phase of Process '`dissect_Gallbladder`' of Fig. 3.3

Fig. 3.4 *unfolds* the compound process `dissect_Gallbladder` and thus exposes its internal structure. The diamond and the triangle delimitate a *guarded process sequence* consisting of the elementary process `receive_PE − forceps` and its compound successor `staunch_Bleedings`, *i.e.* both processes are only executed in case the condition guarding their corresponding control relation evaluates to true ([`bleeding`]); otherwise the control flow is forwarded along the alternative path guarded by condition [`no bleeding`] without execut-

ing the both process. Note that receive_PE − forceps and staunch_Bleedings communicate over the *local variable* j_1, which is not accessible via the syntactic interface of process dissect_Gallbladder. On the other hand, the variables i_1 through i_4 depicted in Fig. 3.4 are contained within dissect_Gallbladder's interface.

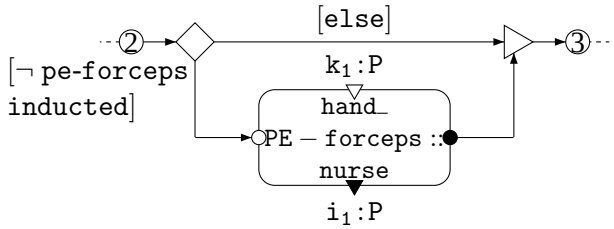

Fig. 3.5: Second Phase of Process 'dissect_Gallbladder' of Fig. 3.3

Fig. 3.5 depicts another guarded process called hand_PE − forceps which sequentially follows the process excerpt depicted in Fig. 3.4 (graphically indicated by the numbered cutting points). Fig. 3.6 depicts the continuation of process dissect_Gallbladder, in which two *alternative process sequences* are depicted, which are guarded by the condition [co] and [cut], indicating whether the surgeon demands the application of cutting (cu) or coagulation (co) current for preparing the patient's gallbladder. Alternative composition implies that *either* of the involved processes are executed, but *not both*.

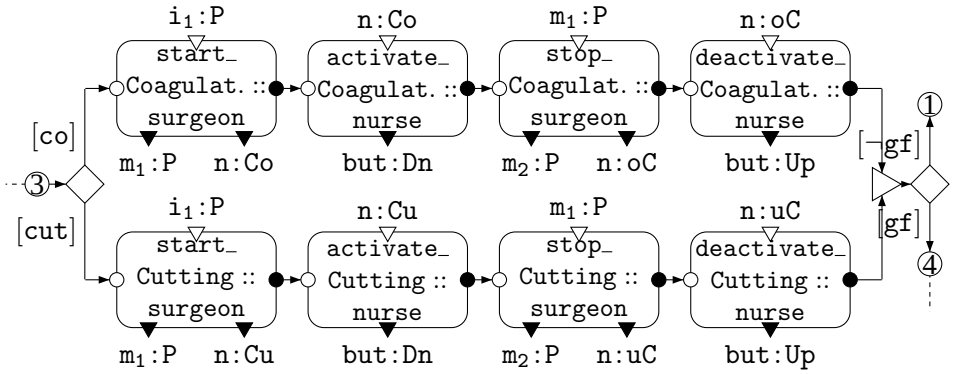

Fig. 3.6: Third Phase of Process 'dissect_Gallbladder' of Fig. 3.3

Note that input variable i_1 of process start_Coagulation in Fig. 3.6 is connected with *two* other I/O variables, namely with the eponymous input variable of super-process dissect_Gallbladder and the output variable of the preceding process hand_PE − forceps. This setting reflects the circumstance that – depending on whether process hand_PE − forceps was executed – the surgeon receives the pe-forceps at the beginning of process dissect_Gallbladder via input variable i_1 or not before the nurse has handed this instrument via hand_PE − forceps's output variable i_1. Thus, for modeling processes we *do not impose the disjointness of output variables*. This is in contrast to component-oriented specification techniques like FOCUS [13], which impose syntactic restrictions such as disjointness of output interfaces to ensure that the composition of components results in a component with input total behavior.

If the gallbladder was successfully released from the liver-bed (condition [gf] in Fig. 3.6)

the nurse *concurrently* receives the seizing-forceps and the pe-forceps from the surgeon as depicted in Fig. 3.7. This expresses, that *(i)* both processes receive_Seizing − Forceps and receive_PE − forceps are simultaneously active, and *(i)* the exact order of their execution *may be irrelevant*. On the other hand, if the gallbladder could not be released (condition [¬gf] in Fig. 3.6), the control flow is forwarded to the beginning of process dissect_Gallbladder in a *guarded iteration*. Graphically, concurrent processes are delimited by two parallels splitting and merging the corresponding control flow of the concurrently executed processes. Note that concurrent process composition, just as any other form of composition in the presented approach, can be arbitrarily nested; *cf.* Fig. 3.8 in which two sequences of processes are composed concurrently.

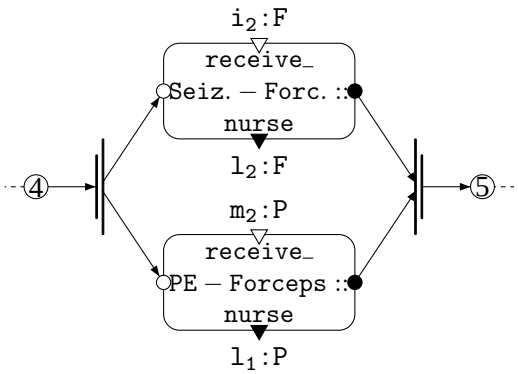

Fig. 3.7: Forth Phase of Process ‘dissect_Gallbladder’ of Fig. 3.3

Fig. 3.8 through 3.10 depict the refining processes of the compound process check_BloodDryness. However, since no new syntactic constructs are introduced within that phase of the surgery, we just present the corresponding process model for sake of completeness.

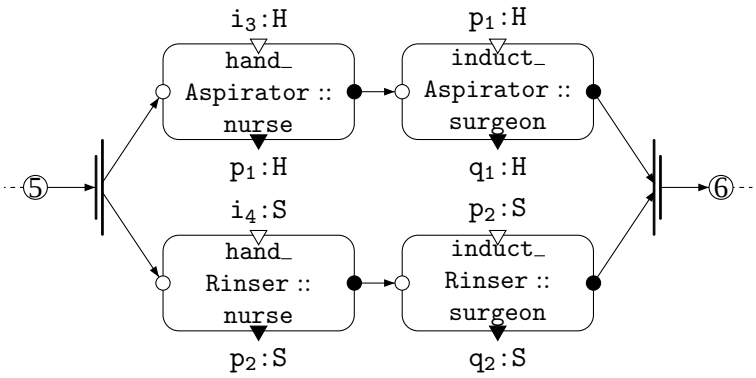

Fig. 3.8: First Phase of Process ‘check_BloodDryness’ of Fig. 3.3

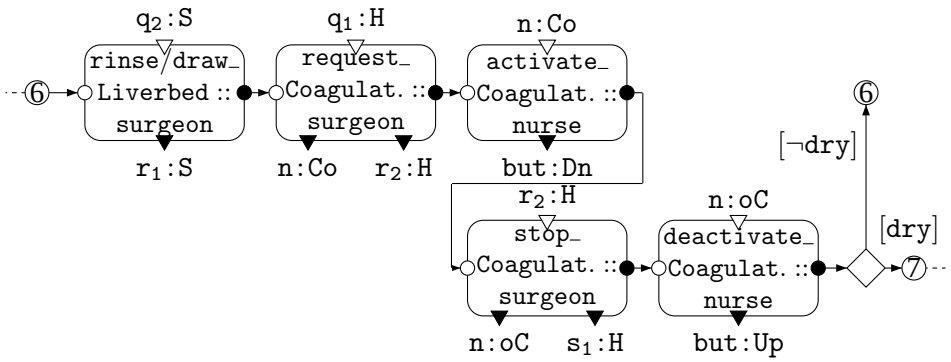

Fig. 3.9: Second Phase of Process ‘check_BloodDryness’ of Fig. 3.3

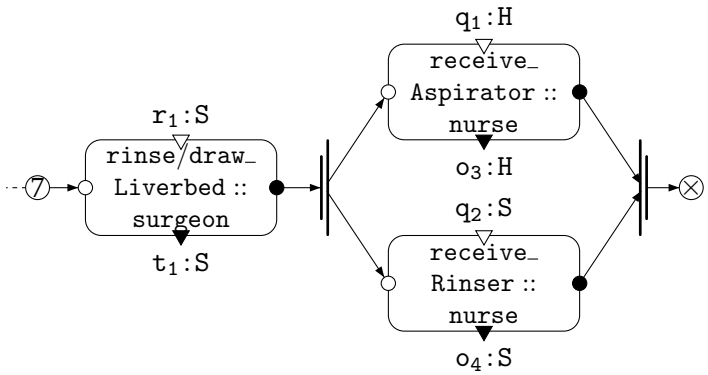

Fig. 3.10: Third Phase of Process ‘dissect_Gallbladder’ of Fig. 3.3

4 Modeling Processes

In this section we present a formal description technique for the specification of work flows. We introduce its abstract syntax together with a graphical representation in Sec. 4.1, before giving a denotational semantics for interpreting process behavior in Sec. 4.2.

4.1 Process Syntax

We use the abstract syntax format to define the '*appearance*' of our description technique in terms of a BNF notation. A work flow description is constructed in accordance with the syntactic equations in Fig. 4.1. Non-terminals are enclosed by $\langle\rangle$; the symbol | separates any two alternatives. The syntactic domain $\langle$proc$\rangle$ denotes a general process.

$$
\begin{array}{lll}
\langle \texttt{proc} \rangle ::= & & \texttt{empty} \\
\quad | & & \texttt{havoc} \\
\quad | & \langle \texttt{procid} \rangle = & \langle \texttt{eproc} \rangle \\
\quad | & [\, \langle \texttt{procid} \rangle =] & \langle \texttt{proc} \rangle \,;\, \langle \texttt{proc} \rangle \\
\quad | & [\, \langle \texttt{procid} \rangle =] & \langle \texttt{proc} \rangle \oplus \langle \texttt{proc} \rangle \\
\quad | & [\, \langle \texttt{procid} \rangle =] & \langle \texttt{proc} \rangle \parallel \langle \texttt{proc} \rangle \\
\quad | & [\, \langle \texttt{procid} \rangle =] & \langle \texttt{proc} \rangle \circlearrowleft_{\langle \texttt{lpspec} \rangle}
\end{array}
$$

Fig. 4.1: Abstract Process Syntax in BNF

The terminal symbols `empty` and `havoc` represent the absence and any form of work flow, resp. They are only of theoretical relevance, not for describing real work flows. Each of the syntactic domains listed above is explained within the next sections.

4.1.1 Elementary Process

The syntactic domain $\langle$eproc$\rangle$ in Fig. 4.1 denotes an elementary process, *i.e.* a process that is not decomposed into further sub-processes (black-box). Each process is identified by a name $\in \langle$procid$\rangle$ and interacts with its environment via its *interface*. We explicitly differentiate between the exchange of data and control flow between a process and its environment. Data flow models process *communication* while control flow models process *(de)activation*.

DEFINITION 1 (SYNTACTIC INTERFACE) *Formally, an* interface *of a process P is a tuple (I, O, S, E), denoted $Intf_P$, containing interface variables $I, O \subseteq Var$ and control points $S, E \subseteq Ctrl$. In order to emphasize the direction of the data flow, we distinguish between the set of input variables I observed by a process, and the set of output variables O controlled by a process.*

Analogously, to emphasize the direction of the control flow, we distinguish between the set of entry points S and exit points E, by which control enters and exits the process, resp. □

A variable has a name $\in V$ and a *type* $\in V \rightarrow TYPE$, where V is a set of variable identifiers and *TYPE* is the set of all types, which in our setting are simple datasets. Analogously, a control point has a name $\in C$, where C is a set of control point identifiers. Each process holds at least one *entry point* by which the process may be activated and start executing. By means of *exit points* the process deactivates and returns the control back to its environment; if no exit point exists, the process maintains the control forever.

A process is understood as an observable activity executed by one or several *actors*, which might be persons, components, technical systems or combinations thereof. To associate a process with its executing actors, we use the concept of *roles*. A role has a name $\in R$, where R is the set of role identifiers. *E.g.*, the role of a process is used to indicate on which component the process is executed. In other words, roles are used to relate logical system architectures with a set of processes describing their behavior.

To represent the concrete syntax of our description technique, we introduce a graphical notation as illustrated in Fig. 4.2. A process is represented by a rectangular data / control flow node with rounded corners. The data flow interface is indicated by labeled triangles connected to the process border. Hollow triangles pointing to the process denote input variables (`i : L`, `n : M`), filled triangles pointing to the environment denote output variables (`k : L`). Similarly, the control flow interface is described by labeled circles connected to the process border, whereby hollow and filled circles are used to distinguish between entry (`strt`) and exit (`end`) points, resp. A process is annotated by a name (`Proc`) and an optional role (`Role`).

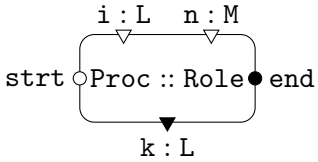

Fig. 4.2: Interface of Elementary Process

4.1.2 Sequential Process Composition

The sequential composition operator ; in Fig. 4.1 takes two processes and composes them in sequel, if at least one of their control point labels coincide. Moreover, if the sequentially composed processes are supposed to communicate over *shared* variables, these variables have to be *consistent*. This means that the considered variable names and their corresponding types must coincide. The exit points of the first process are connected with the coinciding entry points of the second process. In this sense, control points constitute the '*glue*' for composing processes, that determines the order in which the corresponding processes execute.

The graphical notation for composing two processes in sequel is illustrated in Fig. 4.3. Note, that we connect the control points and shared variables in accordance with their identifiers. Since the identifiers of linked control points must coincide, we occasionally omit labels such as `loc` of connected control points. Similarly, we occasionally omit to draw an explicit communication link between shared variables (`m : N`). Such a link is always assumed *implicitly*.

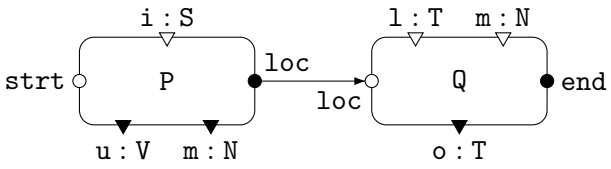

Fig. 4.3: Sequential Process Composition

4.1.3 Alternative Process Composition

The choice operator $\oplus$ in Fig. 4.1 takes two processes as operands and alternatively combines them in accordance with a guard associated to their common entry points. The diamond and the triangle symbols in Fig. 4.4 illustrate the branching of control flow. We assume that the control flow is *not split* by the diamond, *i.e.* the control always flows along at most one outgoing edge whose guard evaluates to true. If several guards evaluate to true, one of them is chosen *non-deterministically*.

When several processes are alternatively composed we assume that those processes do not depend on each other in the sense of control flow. To ensure this, we do not allow to connect control points between any of the alternative processes. This leads to alternative processes that execute in a *pairwise exclusive* fashion. In other words, it is impossible that any two alternative processes execute simultaneously. In Fig. 4.4, the common entry point `ent` of each alternative process are linked to the control flow branch (diamond) while the common exit point `ext` is linked to the optional merging node (triangle).

Note that the interface variables of alternative processes such as $\mathtt{i} : \mathtt{T}$ and $\mathtt{n} : \mathtt{N}$ need not be disjoint. However, the alternative composition of two processes is defined only if the input variables of one process and the output variables of the other do not coincide, *i.e.*

$$I_{\mathtt{P}} \cap O_{\mathtt{Q}} = I_{\mathtt{Q}} \cap O_{\mathtt{P}} = \varnothing.$$

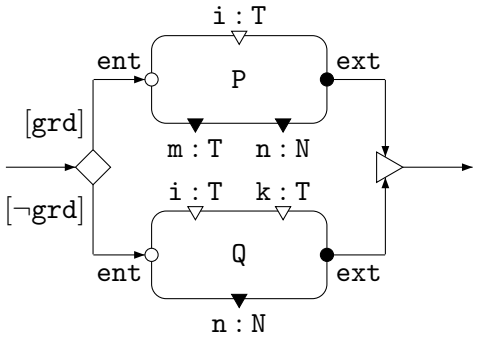

Fig. 4.4: Alternative Processes Composition

The alternative composition can be generalized to more than two processes in the obvious manner. In particular, the diamond can be extended to connect several exit points as incoming edges as depicted in Fig. 4.5.

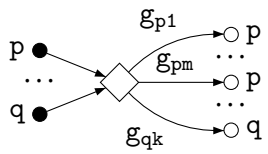

Fig. 4.5: Choice with Several Inputs

4.1.4 Parallel Process Composition

We introduce the parallel composition operator $\|$ in Fig. 4.1 to express that several processes execute *simultaneously*. When entered through their common entry point (*fork*), the control flow splits and simultaneously activates the concurrent processes, while a set of optional end points (*join*) 'synchronizes' their deactivation and returns control to the environment. As in the case of alternative composition, no control points are related between any of the concurrent processes. Instead, the entry points of each concurrent process are linked to the control flow fork (opening parallels) while the exit points may be linked to the optional join (closing parallels).

Interface variables may be shared between concurrent processes. Fig. 4.6 illustrates the graphical notation for composing two processes in parallel, whereby both processes communicate over the shared variable `n : N`. The composition can be generalized to more than two processes in the obvious manner.

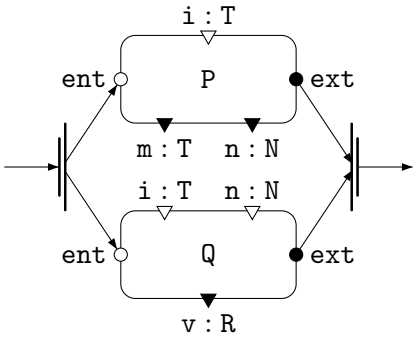

Fig. 4.6: Parallel Process Composition

4.1.5 Repetitive Process Composition

$\circlearrowleft_{\langle \texttt{lpspec} \rangle}$ in Fig. 4.1 represents a repetition operator for processes, *i.e.* depending on the evaluation of a loop specifier `lpspec`, the work flow is consecutively executed a *possibly indefinite* number of times. The loop specifier determines the number of repetitions. It is specified as a natural number $\in \mathbb{N} \cup \infty$ or in form of a guard. Fig. 4.7 illustrates the graphical representation of the repetition operator associated with a process.

Note that we understand a work flow as something *unique* which occurs at most once – like an execution trace of an automaton. However, for convenience we omit to describe each (part of the) work flow by a unique process. Consequently, we use the repetition operator as a shorthand for specifying a possibly infinite number of sequentially composed processes.

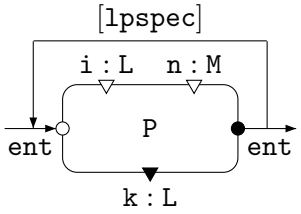

Fig. 4.7: Repetitive Process Composition

4.1.6 Process Hierarchies

We structure processes hierarchically and exploit this hierarchy for structuring the analysis of complex work flow descriptions. For the interface of a hierarchically structured process, certain syntactic constraints must hold which we formally define in the following. In order to relate the interface of a hierarchically structured process with the interfaces of its sub-processes, we first introduce the notion of a *union interface*.

DEFINITION 2 (UNION INTERFACE) *Given a set of processes with syntactic interfaces* $Intf_i = (I_i, O_i, S_i, E_i)$ *for* $i = 1, \ldots, n$ *with* $n \in \mathbb{N}$. *We construct their* union interface (I^*, O^*, S^*, E^*) *by unifying all constituents (variables, control points) contained in* $Intf_i$ *element wise, i.e.*

$$I^* = \bigcup_{i=1}^{n} I_i \ \wedge \ O^* = \bigcup_{i=1}^{n} O_i \ \wedge \ S^* = \bigcup_{i=1}^{n} S_i \ \wedge \ E^* = \bigcup_{i=1}^{n} E_i$$

We denote the union interface $\bigcup_{i=1}^{n} Intf_i \stackrel{\text{def}}{=} (I^*, O^*, S^*, E^*)$. □

However, to construct the interface $Intf_H$ of a hierarchically structured process H, we are interested in exactly those variables and control points of H's sub-process interfaces, which are not '*bound*' by coincidence of names. The notion of a *hierarchal interface* formalizes this idea.

DEFINITION 3 (HIERARCHICAL INTERFACE) *Given a union interface* $\bigcup_{i=1}^{n} Intf_i$ *constructed from interfaces* (I_i, O_i, S_i, E_i) *for* $i, j = 1, \ldots, n$ *with* $n \in \mathbb{N}$. *We call an interface* (I, O, S, E) *a* hierarchical interface, *iff*

$$I = I^* \setminus \{I_i \cap O_j\}, \quad O = O^* \setminus \{I_i \cap O_j\}$$

$$S = S^* \setminus \{S_i \cap E_j\}, \quad E = E^* \setminus \{S_i \cap E_j\}$$

We denote the hierarchical interface $\biguplus_{i=1}^{n} Intf_i = (I,O,S,E)$. □

Fig. 4.8 illustrates the hierarchical interface of a compound process `compProc`, which is constructed from the interfaces of its two sub-processes $\texttt{subProc}_1$ and $\texttt{subProc}_2$. We omitted the type declarations of variables for clarity. Note that the common control point `loc` and the shared variable `l` are not part of `compProc`'s hierarchical interface – both have been *hidden* from the environment as defined in Def. 3.

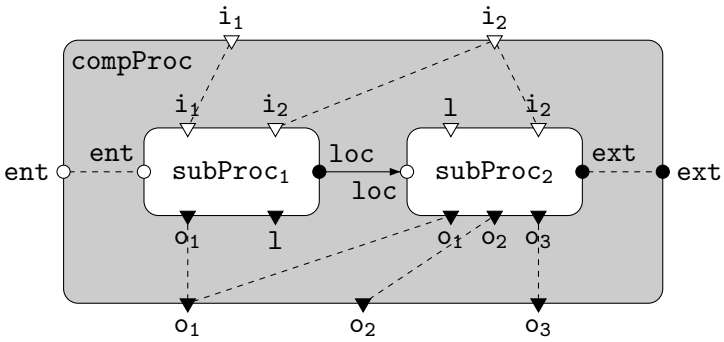

Fig. 4.8: Hierarchical Interface of `compProc`

4.2 Process Semantics

To that end, we introduced the syntactic aspects of process specifications without defining how to describe its *behavior* and its interpretation in terms of mathematical objects like sets, functions, relations, etc. We use the concept of a *labeled transition system* (*LTS*) in order to specify the behavior of a process. Moreover, we give a formal, mathematical interpretation for process behavior based on the set of *observations* induced by such a LTS.

4.2.1 Basic Semantic Concepts

Along the lines of [6, 7, 8], we use well-known concepts for interpreting processes in terms of *states*, *observations*, and *behaviors*. We use these concepts to declare the meaning of (i) an elementary process, (ii) sequential composition, (iii) alternative composition, (iv) parallel composition, and (v) repetitive composition of processes.

State A state $s \in Var \rightarrow Val$ maps variables to their current values, whereas $Var = L \cup I \cup O$ with local variables L, input variables I and output variables O.

Observation An observation is either a triple $(a, \langle t \rangle, b)$ consisting of a finite sequence $\langle t \rangle$ of states corresponding to an execution starting at control point a and ending at control point b, changing variables according to $\langle t \rangle$; or it is a pair $(a, \langle t \rangle)$ consisting of a finite sequence $\langle t \rangle$ of states, corresponding to a *partial* execution, starting at control point a.

Behavior The behavior of a process is the set *Obs* of all its observations, *i.e.* we consider finite behavior only.

4.2.2 Interpretations

System Model

We use processes to specify concurrent and distributed discrete event systems as found in software intensive systems and their operational environment. We reason about the behavior of a process P by considering the observations induced by P's automaton (LTS). This automaton communicates with its environment via its interface. In contrast to a component, the behavior of

a process needs not be totally defined. For a partial specification, it is possible to have a behavior of the environment where no behavior of the process is defined by the specification. By this, process behavior is inherently defined in an assumption/guarantee style: in case the environment violates this assumption and produces illegal inputs, the reaction of the process is *undefined*, *i.e.* the process exhibits the empty set of outputs (input disabled). In contrast, a deactivated process does not constrain any variables whatsoever. This interpretation reflects the *exemplary* nature of a process: a component is composed from partial process specifications until a totally defined component specification emerges, which defines a reaction to every possible input of the environment.

With this in mind, we describe how to interpret the behavior of an elementary process and the different forms of process composition, whereby composition is nothing else than structuring the resulting automaton into modular behavioral descriptions (sub-processes).

Throughout this section, we illustrate all concepts by means of a prominent example of a reactive system: the Automated Teller Machine, ATM.

Elementary Process

The 'structural' aspects of a process are defined by its syntactic interface (I, O, S, E) containing a set of interface variables $\subseteq I \cup O$, with $Var = I \cup O \cup L$, and a set of interface control points $\subseteq S \cup E$. The corresponding process behavior is specified in terms of a labeled transition system. Transitions are influenced by local and input variables and influence local and output variables. A single transition can be understood as the most basic form of a process. Fig. 4.9 depicts such an elementary process for verifying the Pin provided to an ATM – covering the case in which a bad Pin is entered once. When entered through its entry point $\texttt{ent} \in S$, process P reads the current values of its input variables $I = \{\texttt{Crd}, \texttt{Pin}\}$. Then P changes the variable state by writing a new value to its controlled variables $O = \{\texttt{Ack}\}$. When reaching its exit point $\texttt{err} \in E$, the process terminates and passes the control to its environment.

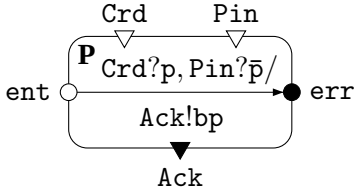

Fig. 4.9: Behavior of Elementary Process

P's behavior is specified via a labeled transition from ent to err. We use the notation described in [14] for labeling transitions, whereby ? and ! denote the *access* and *modification* of variables $\in$ *Var*. The transition has a label consisting of $\texttt{Crd?p}, \texttt{Pin?}\bar{\texttt{p}}/\texttt{Ack!bp}$. The pre-part of the label (before the /) states that, whenever message p is received via variable Crd and the differing message $\bar{\texttt{p}}$ is received via variable Pin, then the transition is *enabled*. If no transition is enabled for a given input, the behavior of P is *undefined* in the sense of our system model. The post-part of the label states that, whenever the transition is executed, in the next state, *i.e. strictly causal* [15], value bp is written to Ack. Fig. 4.9 shows that the transition is depicted by an arrow linking the corresponding control points ent and err.

We interpret each transition label as the logical conjunction of its pre- and post-part, *i.e.* $\texttt{Crd?p} \wedge \texttt{Pin?}\bar{\texttt{p}} \wedge \texttt{Ack!bp}$ for the transition label in Fig. 4.9. This constitutes an *atomic step*, *i.e.* a non-interruptible pair of states (σ, ρ), whereby observation σ and ρ are called the *source* and the *sink* of the step, resp. The step w is *successive* to the step u if the sink of u is equal to the source of w.

We use a primed version of variables and states to argue about the current and the next state within such a step, *i.e.* we use variables $v \in \mathit{Var}$ for values of v prior to the execution of the transition, and variables v' for values of v after its execution. Priming of states yields a mapping of equally valued primed variables, *i.e.* for a given $\sigma \in \mathit{Var} \to \mathit{Val}$, σ' is defined by $\forall\, v \in \mathit{Var} : \sigma(v) = \sigma'(v')$. A step is not allowed to constrain primed input variables and unprimed output variables. By this, we disallow a process to constrain its future inputs and read its own outputs, resp.

For a state $s \in \mathit{Var} \to \mathit{Val}$ with $\mathit{Var}^* \subseteq \mathit{Var}$ we use the notation $s©\mathit{Var}^*$ for *restrictions* $(s©\mathit{Var}^*)(v) = s(v)$ for all $v \in \mathit{Var}^*$, *i.e.* valuations of variables $\notin \mathit{Var}^*$ are ignored.

We extend this restriction to sequences of states through element wise application. For sequences $\langle s\rangle$ and $\langle t\rangle$ we use the notation $\langle s\rangle \circ \langle t\rangle$ to describe their concatenation. Formally, $\langle s_1, \ldots, s_n\rangle \circ \langle t_1, \ldots, t_m\rangle \stackrel{\text{def}}{=} \langle s_1, \ldots, s_n, t_1, \ldots, t_m\rangle$.

DEFINITION 4 *The* behavior of an elementary process *is the set containing all observations* $(a, \langle t\rangle, b)$ *and* $(a, \langle s\rangle)$ *with entry point a, exit point b, and* $\langle s\rangle$ *being any prefix of the finite sequence* $\langle t\rangle$ *of successive steps. The behavior of a process P is* prefix-closed *to ensure that an observation can be operationally generated in a stepwise manner:*

$$(a, \langle t\rangle, b) \in \mathit{Obs}_P \Rightarrow (a, \langle t\rangle) \in \mathit{Obs}_P \Rightarrow (a, \langle s\rangle) \in \mathit{Obs}_P$$

□

Consequently, the behavior of process P in Fig. 4.9 is the set consisting of all observations *(i)* $(\texttt{ent}, \langle(\sigma, \rho)\rangle, \texttt{err})$, *(ii)* $(\texttt{ent}, \langle(\sigma, \rho)\rangle)$, and *(iii)* $(\texttt{ent}, \langle\rangle)$, such that $\sigma(\texttt{Crd}) = \texttt{p}$, $\sigma(\texttt{Pin}) \neq \texttt{p}$, and $\rho'(\texttt{Ack}') = \texttt{bp}$.

Sequential Composition

We interpret the *sequential composition* $P\ ;\ Q$ in an end-to-start manner *w.r.t.* control flow, *i.e.* after process P terminates via one of its exit points, the control *immediately* transfers to the coinciding entry point and activates process Q.

DEFINITION 5 *The sequential composition of two processes P and Q results in a compound process* $P\ ;\ Q$

(i) whose hierarchical interface is $\biguplus_{i=P,Q} \mathit{Intf}_i$*, and*

(ii) exhibits the behavior of either process, with the restriction that Q is not activated before P, i.e. $\exists\, b$*, such that*

$(a, \langle s \circ t\rangle, c) \in \mathit{Obs}_{P;\,Q} \Leftrightarrow (a, \langle s\rangle©\mathit{Var}_P, b) \in \mathit{Obs}_P \ \wedge\ (b, \langle t\rangle©\mathit{Var}_Q, c) \in \mathit{Obs}_Q$*;*

$(a, \langle s \circ t\rangle) \in \mathit{Obs}_{P;\,Q} \Leftrightarrow (a, \langle s\rangle©\mathit{Var}_P, b) \in \mathit{Obs}_P \wedge (b, \langle t\rangle©\mathit{Var}_Q) \in \mathit{Obs}_Q$*;*

$(a, \langle s\rangle) \in \mathit{Obs}_{P;\,Q} \Leftrightarrow (a, \langle s\rangle©\mathit{Var}_P) \in \mathit{Obs}_P$*.*

In other words, the sequential composition acts concatenatively on traces. Note that $\langle s \rangle$ and $\langle t \rangle$ denote sequences of successive steps. Consequently, the last state in $\langle s \rangle$ and the first state in $\langle t \rangle$ coincide. □

If process P within the sequential composition P ; Q does not terminate, Q is never activated. Fig. 4.10 depicts the sequential composition of process `P` and `Q` – representing a negative Pin evaluation of the ATM followed by a positive one – resulting in the compound process `P ; Q`. The behavior of `P ; Q` is the set consisting of all observations *(i)* $(\texttt{ent}, \langle(\sigma,\rho),(\rho,\phi)\rangle, \texttt{scc})$, *(ii)* $(\texttt{ent}, \langle(\sigma,\rho),(\rho,\phi)\rangle)$, *(iii)* $(\texttt{ent}, \langle(\sigma,\rho)\rangle)$, and *(iv)* $(\texttt{ent}, \langle\rangle)$, such that $\sigma(\texttt{Crd}) = \texttt{p}, \sigma(\texttt{Pin}) \neq \texttt{p}$, $\rho'(\texttt{Ack}') = \texttt{bp}$, $\rho(\texttt{Crd}) = \rho(\texttt{Pin}) = \texttt{p}$, $\phi'(\texttt{Ack}') = \texttt{ok}$.

Note that the control point `err` does not occur within any of the above observations, *i.e.* it is *hidden* by the sequential composition. `P` and `Q` are allowed to communicate over shared variables. If those variables should also be hidden from the process environment, *variable hiding* can be used as described next.

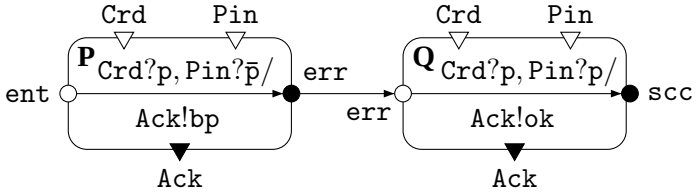

Fig. 4.10: Behavior of Sequential Processes

Variable Hiding

Hiding an interface variable of a process renders the variable inaccessible from the outside. By hiding a variable v of a process P we obtain a process described by $P \setminus v$, that accesses the same control points as P and uses the input and output variables of P *excluding* v:

$$Var_{P\setminus v} = Var_P \setminus \{v\}.$$

Moreover, $P \setminus v$ exhibits the same behavior as P, *i.e.*

$$(a, \langle t \rangle \copyright Var_P \setminus \{v\}, b) \in Obs_{P\setminus v} \Leftrightarrow (a, \langle t \rangle, b) \in Obs_P,$$

and

$$(a, \langle t \rangle \copyright Var_P \setminus \{v\}) \in Obs_{P\setminus v} \Leftrightarrow (a, \langle t \rangle) \in Obs_P.$$

Alternative Composition

By *alternatively* composing two processes P and Q, we express that *either P or Q* is executed, but *not both*. Hence, we interpret the alternative composition $P \oplus Q$ as an '*exclusive or*' relation between processes. More precisely, when control resides in their common entry point a, either process P or Q is activated depending on the *guarding conditions* of a, but not both. A guard is simply a predicate $\in S \to \mathbb{B}$ over the process's state space $S \subseteq Var \to Val$, *i.e.* a's guards evaluate $P \oplus Q$'s observed variables and either activate P or Q. In case several guards evaluate to true, one process is activated non-deterministically.

DEFINITION 6 *The* alternative composition *of two processes P and Q results in a compound process* $P \oplus Q$

(i) whose hierarchical interface is $\biguplus_{i=P,Q} Intf_i$*, and*

(ii) exhibits either the behavior of process P or Q, i.e.

$$(a, \langle t \rangle, b) \in Obs_{P\oplus Q} \Leftrightarrow (a, \langle t \rangle \copyright Var_P, b) \in Obs_P \vee (a, \langle t \rangle \copyright Var_Q, b) \in Obs_Q;$$

$$(a, \langle t \rangle) \in Obs_{P\oplus Q} \Leftrightarrow (a, \langle t \rangle \copyright Var_P) \in Obs_P \vee (a, \langle t \rangle \copyright Var_Q) \in Obs_Q.$$

In other words, the alternative composition acts disjunctively on traces. □

Fig. 4.11 depicts the alternative composition of processes P and Q from above – representing the alternative execution of a positive and negative Pin evaluation of the ATM – resulting in the compound process $\mathtt{P} \oplus \mathtt{Q}$. Its behavior contains all observations *(i)* $(\mathtt{ent}, \langle(\sigma_\mathtt{P}, \rho_\mathtt{P})\rangle, \mathtt{err})$, *(ii)* $(\mathtt{ent}, \langle(\sigma_\mathtt{P}, \rho_\mathtt{P})\rangle)$, *(iii)* $(\mathtt{ent}, \langle(\sigma_\mathtt{Q}, \rho_\mathtt{Q})\rangle, \mathtt{scc})$, *(iv)* $(\mathtt{ent}, \langle(\sigma_\mathtt{Q}, \rho_\mathtt{Q})\rangle)$, and *(v)* $(\mathtt{ent}, \langle\rangle)$, such that $\sigma_\mathtt{P}(\mathtt{Crd}) = \mathtt{p}$, $\sigma_\mathtt{P}(\mathtt{Pin}) \neq \mathtt{p}$, $\rho'_\mathtt{P}(\mathtt{Ack}') = \mathtt{bp}$, $\sigma_\mathtt{Q}(\mathtt{Crd}) = \sigma_\mathtt{Q}(\mathtt{Pin}) = \mathtt{p}$, and $\rho'_\mathtt{Q}(\mathtt{Ack}') = \mathtt{ok}$.

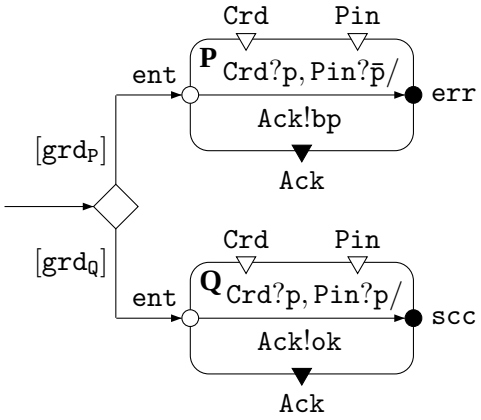

Fig. 4.11: Behavior of Alternative Processes

Note that alternative processes do not need to have a common exit point, and that the guards of alternative processes are *inherently* contained in the observations in Def. 6. *E.g.*, the guards of process $\mathtt{P} \oplus \mathtt{Q}$'s common entry point ent are defined by the corresponding pre-parts of the transition labels with ent as starting point, *e.g.*,

$$\mathtt{grd_P} \stackrel{\text{def}}{=} (\sigma_\mathtt{P}(\mathtt{Crd}) = \mathtt{p} \wedge \sigma_\mathtt{P}(\mathtt{Pin}) \neq \mathtt{p}).$$

For methodical reasons, we allow to annotate the diamond with guards $[\mathtt{grd_i}]$ with $\mathtt{i} \in \{\mathtt{P}, \mathtt{Q}\}$ in the graphical representation, even though they actually denote the pre-part of the transition label leaving the related entry point ent.

Parallel Composition

By composing two processes *P* and *Q in parallel*, we express that *both P and Q* execute simultaneously. When entered through their common entry point *a*, the control flow is split so that both processes *P* and *Q* exhibit their joint behavior.

DEFINITION 7 *The parallel composition of two processes P and Q results in a compound process $P \parallel Q$*

(i) whose hierarchical interface is $\biguplus_{i=P,Q} Intf_i$, and

(ii) exhibits the combined behavior of each process, i.e.

$$(a, \langle t \rangle, b) \in Obs_{P\parallel Q} \Leftrightarrow (a, \langle t \rangle \copyright Var_P, b) \in Obs_P \wedge (a, \langle t \rangle \copyright Var_Q, b) \in Obs_Q;$$
$$(a, \langle t \rangle) \in Obs_{P\parallel Q} \Leftrightarrow (a, \langle t \rangle \copyright Var_P) \in Obs_P \wedge (a, \langle t \rangle \copyright Var_Q).$$

In other words, the parallel composition acts conjunctively on traces. In particular, each step of P corresponds to a concurrent step of Q. □

To terminate the compound process $P \parallel Q$, both P and Q need to terminate via their common exit point. Note that the parallel composition is *strict* in the sense that undefined behavior of one process 'knocks out' defined behavior of the other process. This is motivated from a methodical point of view, since we can not rely on undefined behavior. More precisely, unless we require P and Q to have disjoint output variables, the parallel composition may lead to conflicting valuations of output variables and the introduction of additional undefined behavior. Moreover, if either P or Q receives illegal inputs, the behavior of $P \parallel Q$ is undefined.

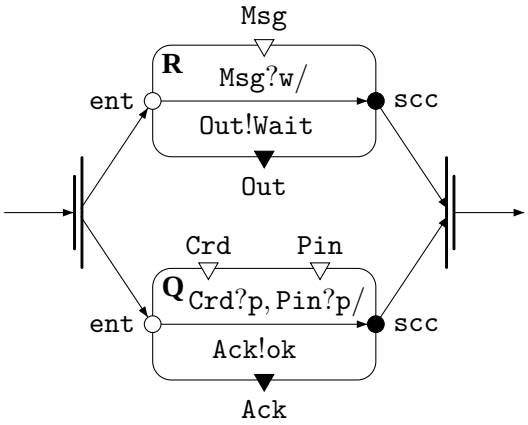

Fig. 4.12: Behavior of Parallel Composition

Fig. 4.12 depicts the parallel composition of process `Q` from above and another process `R` – representing an output to the ATM's user interface – resulting in the compound process `R` $\parallel$ `Q`. Its behavior contains all observations *(i)* $(\texttt{ent}, \langle(\sigma, \rho)\rangle, \texttt{scc})$, *(ii)* $(\texttt{ent}, \langle(\sigma, \rho)\rangle)$, and *(iii)* $(\texttt{ent}, \langle\rangle)$, such that $\sigma(\texttt{Msg}) = \texttt{w}$, $\sigma(\texttt{Crd}) = \texttt{p}$, $\sigma(\texttt{Pin}) = \texttt{p}$, $\rho'(\texttt{Out}') = \texttt{Wait}$, $\rho'(\texttt{Ack}') = \texttt{ok}$.

Repetitive Composition

By composing a processes *P repetitively*, we express that P is executed *sequentially* a possibly indefinite number of times. This number is either determined by a constant or a guarding condition of P's entry point. Consequently, repetitive composition is just a shorthand for repeatedly applying sequential composition. Hence we do not give an explicit behavioral interpretation, since composition can be arbitrarily nested anyhow.

Note that in our approach a process describes exemplary behavior which is typically *finite*. This is reflected in the corresponding definitions of process behavior, in which all observations denote finite sequences of variable valuations. By introducing the repetition operator $\circlearrowleft_\mu$, we extend this view in the sense that a work flow may be described by a process $P \circlearrowleft_\mu$, which induces a possibly infinite behavior. However, the behavior of each sub-process of $P \circlearrowleft_\mu$ is *finite* and *strictly causal* [15]. As well-known, unique fixed points for strictly causal behaviors always exist.

5 Application: Surgery Assistance

This section outlines a model-based approach for the formal, structured specification of system behavior on basis of user work flows and context information. We illustrate the application of our modeling approach using the lapCHE case study. The approach structures the development along the lines of three views, which capture all aspects necessary for the design of the system's behavior. We describe in parts the *work flow*, *contextual*, and *functional view* of a model specifying a context-aware *Surgery Assistance System* (*SAS*). The SAS assists an operating team during the surgical intervention.

In a nutshell, the SAS 'observes' an ongoing lapCHE by means of sensors installed within the operating room (OR), *e.g.*, sensors for currently used instruments, table position, room lights, etc. Additionally, it accesses a data model of the lapCHE comprising its work flow and context view. On basis of both information, it *tracks* the actual surgery progress and adapts its behavior accordingly, *e.g.*, by providing the estimated remaining surgery duration, recommending the instruments needed for the next surgery step, or indicating critical surgery situations by means of an early warning system.

The model of the lapCHE case study serves as a running example throughout this section and illustrates how a concrete work flow, contextual and functional view of an actual system looks like. Fig. 5.1 illustrates the involved system views which are presented in the following three sections. In Sec. 5.4, we relate our approach to a requirements engineering methodology [10] with a similar purpose, namely the integrated elaboration of functional requirements (*cf. functional view*) and contextual information (*cf. work flow* and *contextual view*).

5.1 Work Flow View

In our approach, processes form the building blocks for specifying data and control flows in the environment (work flow view) and within the system (functional view). Our approach starts with the specification of (abstract) user work flows in terms of processes. Ex. 8 gives an example for a work flow view, whereby the used description technique has been described previously in Sec. 4. Although not presented in the work flow view depicted in Fig. 5.1, work flow behavior can also be modeled in terms of labeled transition systems. However, in our running example we abstract from the exact behavior of user work flows and only provide a name, a role, and an interface for each process describing the work flow parts of the lapCHE.

EXAMPLE 8 The *work flow view* in Fig. 5.1 contains three processes describing an extract of the lapCHE surgery. The depicted work flow represents a scenario, in which the `surgeon` (*role*) in a first step cuts the patient's navel – indicated by the *process* named `cut_navel`. After the `surgeon` has finished this activity, he hands the scalpel (variable $\texttt{Scl}'$) to a `nurse` in process

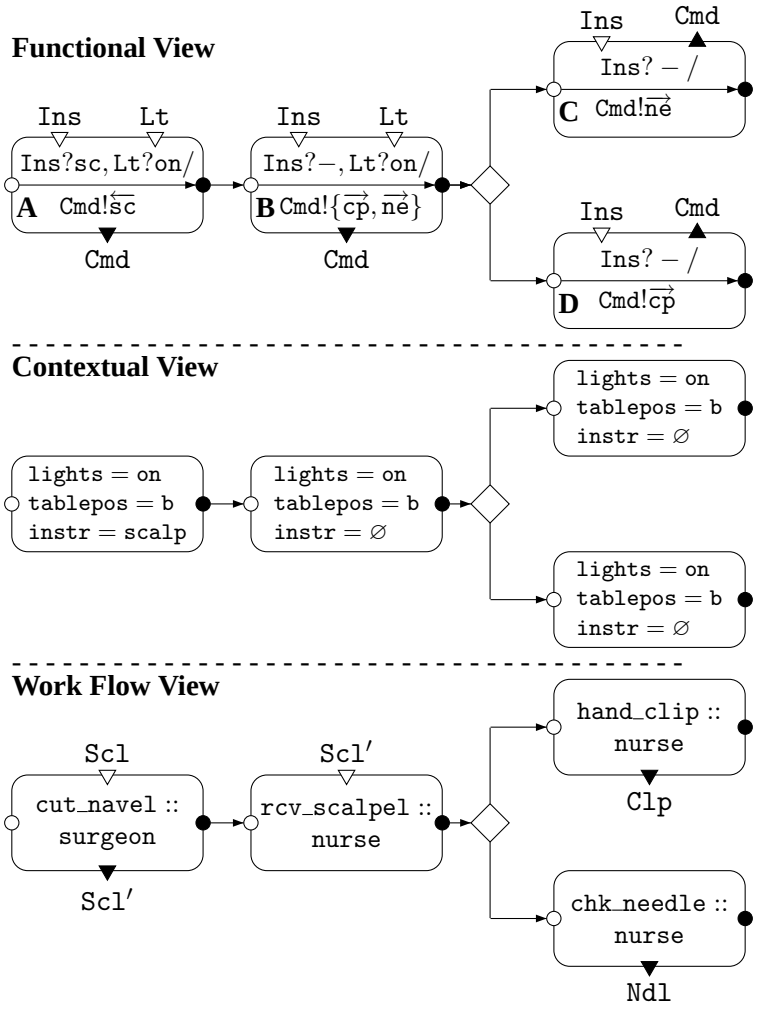

Fig. 5.1: Integrated System Views of LapCHE Extract

`rcv_scalpel`. Subsequently, the `nurse` either hands the clip to the surgeon (`hand_clip`) or alternatively checks the needle (`chk_needle`). □

5.2 Contextual View

The *contextual view* describes for each process contained in the work flow view, how this process can be characterized by context information observable within the system environment. As indicated in the contextual view in Fig. 5.1, several processes may exhibit the same context values. The context is represented in terms of a Data Type Definition (DTD). In case of the context in Fig. 5.1 denoted `lights`, basic data types like *Bool* are sufficient to express that the lights in the OR are switched `on` or `off`. Other context information require more complex types: consider the *duration* of a process, that is specified by an interval ranging over the natural numbers $\mathbb{N}$ to express that, *e.g.*, process `cut_navel` takes between 30 and 60 seconds to execute. If operations

over these data types are required, one can also consider to introduce an *algebraic specification* [16] for each context type.

In case of the lapCHE's contextual view, we use eight context information with simple data types such as *Bool*, enumerations and (intervals over) $\mathbb{N}$. Each context information corresponds to one sensor installed within the OR. The context determines the nominal values of the sensor within each process of the lapCHE's work flow view and statistically bases on a survey of approx. 200 surgeries with patients of different gender, age, and medical records.

EXAMPLE 9 The associated context of process `cut_navel` in Fig. 5.1 is interpreted as follows: When cutting the patient's navel, we suppose that (i) the `lights` in the OR are `on`, (ii) the `tableposition` of the operating table is `balanced`, and (iii) the `surgeon`'s currently used `instrument` is a `scalpel`. □

We use these nominal context values to determine the (de)activation and the triggering conditions of each process's LTS contained in the functional view. The fundamental notion of the first two views is the following: given a context-aware system that gets as input *(i)* the work flow of the user in terms of a *work flow view*, *(ii)* the nominal context values of each process in the work flow view in terms of a *contextual view*, and *(iii)* the *current context* information measured by an appropriate set of sensors. Then, the system is able to *match* the nominal against the currently measured context values, and – on basis of this information – is able to *track* the current position within a user work flow like the lapCHE surgery.

5.3 Functional View

The work flow and the contextual view enable a system like the SAS to track the actual position within the user's work flow. On the other hand, the *functional view* describes which functionality the system provides for each of the user's activities contained in the work flow. In other words, if the system successfully tracks the user's current activity, *e.g.*, by recognizing that the surgeon is currently cutting the patient's navel, the functional view determines how the system should react to that *situation* in the most appropriate way, *e.g.*, by activating a certain system function needed in that situation. The system behavior required in each part of the user's work flow is specified in the same description technique as the work flow itself, *i.e.* in terms of a LTS. Note, that we abstracted from this behavior in the work flow view in Fig. 5.1. The hierarchical decomposition given by the work flow view imposes the basic structure of this LTS in terms of processes.

EXAMPLE 10 Process `B` in Fig. 5.1 specifies that the system recommends the nurse to keep the clips and the needle on hand after the surgeon finished cutting the patient's navel. When activated through its entry point, no instrument is used (`Ins`$?-$) and the lights are on (`Lt`$?$`on`), then the process displays the surgery clips (`cp`) and the needle (`ne`) as the next instruments to hand over ($\rightarrow$) on the monitor within the OR (`Cmd`$!\{\overrightarrow{\texttt{cp}}, \overrightarrow{\texttt{ne}}\}$), and terminates via its exit point.□

Note that the context information defined in the contextual view can be used to formulate the triggering conditions for the LTS in the functional view. In particular, we use context values to (de)activate a system function. *E.g.*, process `B` in Fig. 5.1 is activated exclusively on basis

of contextual information (`instrument`, `room lights`). In this connection, we also speak of *context-triggered* processes / transitions. The possibility to group together states within modules can also be found in other description techniques like Statecharts [17], where such hierarchical states, denoted *modes*, are used to express alternative, sequential behavior. Thus, processes and their context-triggered (de)activation are a convenient form to specify reconfigurable behavior occurring, *e.g.*, in context-aware systems like the SAS.

5.4 Contextual Requirement Chunks

Since the specification of the functional view contains all information relevant for the functional design of the system, the contextual and work flow view are only relevant from a methodical point of view. They structure the construction and analysis of the desired system behavior.

Similar to the proposed model-based approach, [10] introduces an informal approach for the integrated elicitation of a system's functional requirements and usage context. This approach results in a concise, text-based table containing the relevant contextual information characterizing a usage situation together with the associated functional requirements. The rows within this table are denoted *Contextual Requirement Chunks* (CRCs). Such a chunk is a tuple

$$CRC \in Req \times Sit \times Sce$$

consisting of a functional requirement *Req*, a context instance *Sit* characterizing the usage situation, and an illustrating scenario *Sce*. By means of CRCs, functional requirements are related to the context in which they are valid. Basically, a CRC expresses the following: '*If* a certain usage situation *Sit* is present, *then* the associated requirement *Req* is valid, which is illustrated by scenario *Sce*'.

The experience gathered since the introduction of CRCs in [10] and its application in several case studies motivated the model-based approach presented here. Actually, it can be understood as the formalization of this idea. Due to its formal semantics in terms of labeled transition systems, the model-based approach facilitates the specification of *history-dependent* behavior, which can be encoded in the LTS. Moreover, interesting properties such as conformance *w.r.t.* a reference model, consistency and completeness of the involved system specifications can be verified. By this, the model-based approach enables an efficient development process with (automatic) support for verification.

6 Conclusion

We conclude this paper with a short summary. Furthermore, we discuss our ongoing work, and outline further promising research directions.

6.1 Summary & Evaluation

We formally defined a class of entities called *processes*. The intended use of processes is to provide a formal, structured model for describing the control and data flows occurring in software-intensive systems and their environment. The model is formal in that it defines a process as a mathematical object, which can be analyzed. The model is structured in that it permits the hierarchical definition of a process. This hierarchy can be exploited for structuring the analysis. Processes constitute the basic building blocks for describing data and control flows, that can be composed sequentially, alternatively, and in parallel – arbitrarily nested. The mathematical semantics of a process is given by its interface and its set of observations.

We evaluated the practicalness of our description technique in terms of a case study from the medical domain, in which we specified the work flow of a minimal invasive surgery, denoted laparoscopic cholecystectomy (lapCHE) [12]. The surgery is a highly standardized work flow in which the patient's gallbladder is removed under general anesthesia. The overall process describing the lapCHE contains about 270 elementary sub-processes which are structured into four hierarchy levels with eight hierarchical processes on the most abstract level.

6.2 Outlook

Process-Based Development The work presented in this paper only constitutes the first steps towards a process-based, integrated development approach for the design of reactive systems, which is outlined in the following.

(i) Starting with the formalization of exemplary user work flows and the associated functional requirements in terms of structured labeled transition systems, denoted processes, we specify the interaction between the system and its environment.

(ii) Due to the underlying formalism, these specifications can be automatically checked for interesting properties such as, *e.g.*, conformance, consistency and completeness.

(iii) On basis of these specifications and a predefined mapping of processes to software components via *roles*, we synthesize component behavior. More precisely, we synthesize a totally defined component specification for each role within the process specification, such that each of these component LTSs respects the behavioral restrictions imposed by the underlying process specification.

To put it in a nutshell, we transform a complex process LTS into a behavioral equivalent / refined set of component LTSs, which can then be implemented independently. Fig. 6.1 illustrates the fundamental idea behind the process-based development of software system, whereby solid and dotted lines represent data and control flow, resp. We compose partial process behavior until the behavior of each component in the logical architecture is completely defined. In this light, component A depicted in Fig. 6.1 implements the behavior of process $(\mathtt{P}\ ;\ \mathtt{Q})\circlearrowleft$, whereas component B implements the behavior of process $(\mathtt{R} \oplus \mathtt{S})\circlearrowleft$.

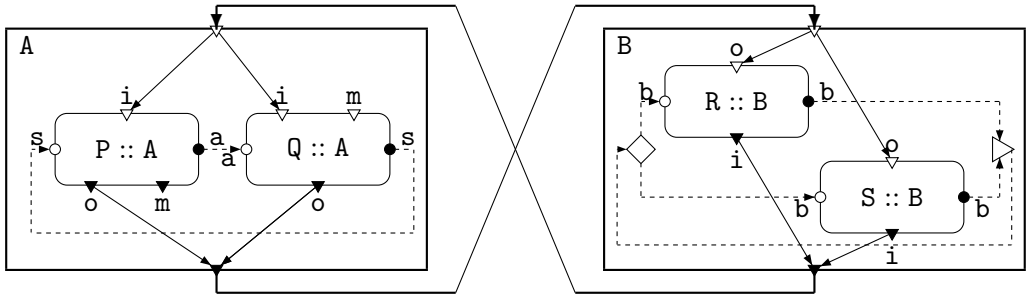

Fig. 6.1: Process-Based Development of Component Architectures

To enable an integrated development process, it is necessary to establish an implementation relation between the specifications created during the different stages of system development. Our current work addresses the definition of such an implementation relation which relates concrete and abstract processes as well as partially defined processes and totally defined components, resp. Obviously, simple implementation relations based on trace inclusion are inappropriate for partial specifications, since they do not reflect the reduction of undefined behavior required for the refinement [18]. Moreover, to effectively use such an implementation relation in a sound development process, (automatic) support for its verification is necessary.

Tool Support We are currently integrating our process-based approach into an existing CASE tool. AutoFOCUS[1] is a tool for the component-based development of reactive systems. It supports the graphical description of the system using different integrated diagram types.

Our current work includes the extension of this tool by a perspective dealing with the process-based specification of the system and its environment. This perspective should offer three different views. In the *Project Explorer* view, processes are hierarchically structured. In the *Process Structure Diagram* view, syntactic interfaces are defined. The *State Transition Diagram* view describes the behavior of each process in terms of a LTS. The existing simulation and verification environments of AutoFOCUS should be adapted to cope with the presented process semantics. An additional synthesizing functionality should automatically construct *(i)* an AutoFOCUS *System Structure Diagram* by combining the syntactic process interfaces to component interface via roles, and *(ii)* a *State Transition Diagram* for each component by composing the underlying process LTSs with coinciding roles. Then, this component LTS can be verified for completeness.

[1] http : //af3.in.tum.de/

Bibliography

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Reading Mass, 1999.

[2] Wolfgang Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[3] Khodakaram Salimifard and Mike Wright. Petri net-based modelling of workflow systems: An overview. *European Journal of Operational Research*, 134(3):664 – 676, 2001.

[4] H. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, Univ. of Twente, November 2002.

[5] V. Thurner. A Formally Founded Description Technique for Business Processes. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Software Engineering for Parallel and Distributed Systems, PDSE98*, pages 254 – 261. IEEE Computer Society, 1998.

[6] Thomas A. Henzinger. Masaccio: A formal model for embedded components. In *TCS '00: Proceedings of the International Conference IFIP on Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics*, pages 549–563, London, UK, 2000. Springer-Verlag.

[7] Bernhard Schätz. Building components from functions. *Electr. Notes Theor. Comput. Sci.*, 160:321–334, 2006.

[8] Bernhard Schätz. Modular functional descriptions. In *Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2007)*, 2007.

[9] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[10] Wassiou Sitou and Bernd Spanfelner. Towards Requirements Engineering for Context Adaptive Systems. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 2, pages 593–600, Beijing, China, July 2007. IEEE Computer Society.

[11] M. Broy, C. Leuxner, W. Sitou, B. Spanfelner, and S. Winter. Formalizing the notion of adaptive system behavior. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1029–1033, New York, NY, USA, 2009. ACM.

[12] H. Feussner, A. Ungeheuer, L. Lehr, and JR. Siewert. Technique of laparoscopic cholecystectomy. *Langenbecks Arch Chir.*, 376(6):367–74, 1991.

[13] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[14] Franz Huber, Bernhard Schätz, and Geralf Einert. Consistent Graphical Specification of Distributed Systems. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME '97: 4th International Symposium of Formal Methods Europe, Lecture Notes in Computer Science 1313*, pages 122 – 141. Springer, 1997.

[15] M. Broy. *Engineering Theories of Software Intensive Systems*, chapter Service-oriented Systems Engineering: Specification and Design of Services and Layered Architectures - The JANUS Approach, pages 47 – 81. Springer Verlag, July 2005.

[16] Martin Wirsing. Algebraic specification languages: An overview. In *COMPASS/ADT*, pages 81–115, 1994.

[17] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[18] Michael von der Beeck. Behaviour specifications: Equivalence and refinement. In H. Giese and S. Phillippi, editors, *Visuelle Verhaltensmodellierung verteilter und nebenläufiger Software- Systeme*, 2000.