# Semantics and Verification of UML Activity Diagrams for Workflow Modelling

*Promotiecommissie:*
Prof. dr. R. J. Wieringa (promotor)
Prof. dr. W. M. P. van der Aalst, Technische Universiteit Eindhoven
Prof. dr. G. Engels, Universität Paderborn
Prof. dr. H. Brinksma
Dr. M. M. Fokkinga (referent)
Dr. ir. P. W. P. J. Grefen
Prof. dr. W. H. M. Zijm (voorzitter)

The research reported in this thesis has been financially supported by the Netherlands Organisation for Scientific Research (NWO), within the scope of project nr. 612-62-02, Deontic Activity and Enterprise Modeling with an Object-oriented Notation (DAEMON).

SIKS Dissertation Series No. 2002-15
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

CTIT Ph.D.-thesis Series No. 02-44
Centre for Telematics and Information Technology (CTIT)
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands



ISBN: 90-365-1820-2
ISSN: 1381-3617; no. 02-44 (CTIT Ph.D.-thesis series)

# SEMANTICS AND VERIFICATION OF UML ACTIVITY DIAGRAMS FOR WORKFLOW MODELLING

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. F. A. van Vught,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 25 oktober 2002 te 15.00 uur.

door

**Hendrik Eshuis**

geboren op 17 september 1975
te Almelo

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. R. J. Wieringa

# Acknowledgements

Although I wrote this thesis, I couldn't have done it without the help of several people. First of all, I thank my promotor Roel Wieringa, who managed to find time to read my obscure manuscripts, and more importantly provided useful feedback to improve them. Roel gave me the freedom to pursue my own research interests and helped me to phrase the results in an understandable way.

I thank the other members of my promotion committee, Wil van der Aalst, Gregor Engels, Ed Brinksma, Maarten Fokkinga, and Paul Grefen, for their useful comments on a previous version of this thesis.

Besides Roel, several other people influenced the contents of this thesis. Paul Grefen and Wijnand Derks were always willing to have a good discussion on workflow modelling. Wil van der Aalst and Jörg Desel provided helpful criticism on a paper that forms the basis of Chapter 8. Working with Juliane Dehnert confronted me with a Petri net view on workflow modelling, which has been very illuminating for me. Mathematicians Maarten Fokkinga and David Jansen managed to increase the precision of my statements.

In 2001, I attended the FASE conference, part of ETAPS, to present a paper on the requirements-level semantics. In a keynote talk at this conference, Bran Selic rejected what he called 'platonic abstractions' like the perfect synchrony hypothesis, because they are, in his opinion, unrealistic and unimplementable. Even though I disagree with him, his talk triggered me to define a low-level implementation-level semantics that does not satisfy perfect synchrony, and to prove that the two semantics have similar behaviour.

Room mates David Jansen and Wijnand Derks, as well as the rest of the IS and DB group, provided a pleasant atmosphere to work in. Secretaries Sandra Westhoff, Suse Engbers and Els Bosch managed my travels by plane and train. Maarten Fokkinga and Rick van Rein guided the research I did for my Master's thesis in such a stimulating way that it made me think of doing a PhD.

Last but not least, I'd like to thank my friends and family. Omdat dit de afsluiting van mijn onderwijscarrière is, wil ik graag twee personen in het bijzonder bedanken: mijn vader en moeder hebben me door de jaren heen altijd gesteund in alles wat ik deed: *Pa en ma, bedankt!*

*Rik Eshuis, October 2002*

# Contents

# Chapter 1

# Introduction

In this thesis, we show how model checking can be used to verify functional requirements on workflow specifications. To specify workflows, we use UML activity diagrams. Since UML activity diagrams lack a formal semantics, we define a formal semantics for activity diagrams that is suitable for workflow modelling.

To define the problem more precisely, in Section 1.1 we introduce some terminology. Then, in Section 1.2, we define the problem. In Section 1.3, we explain the problem-solving approach. Section 1.4 gives an outline of the remainder of this thesis.

## 1.1 Background

A workflow is an operational business process. Workflow management is concerned with the control and coordination of workflows. Several computer-based systems have been developed that implement workflow management, either as a dedicated system or as part (component) of a larger system, for example as part of an Enterprise Resource Planning system. We call such systems workflow management systems[1] (WFMSs). Workflow management systems, once used, are vital for an organisation, since the processes that they support are usually primary and secondary processes. Malfunctioning of WFMSs hampers the functioning of organisations, and may lead to a decline in the quality of products and services that the organisation delivers to society. In recent years, there has been a trend to use WFMSs to integrate distributed systems which may be cross-organisational. In this case malfunctioning of one WFMS can affect more than one organisation, making the correct functioning of a WFMS even more critical than before.

An important function of WFMSs is to enforce certain ordering rules between

[1] In this thesis, we use the term 'workflow management system' to denote every computer system or part of a computer system that implements workflow management functionality, even though in literature the term is reserved for a dedicated computer system.

business activities. For instance, in a workflow that handles insurance claims, an example ordering rule could be that after a claim is registered, it is checked. Apart from ordering rules, a WFMS enforces other rules, for example allocation rules, which state to which actor in the organisation a WFMS may allocate an activity. In this thesis, however, we only consider ordering rules.

Rules that a WFMS must enforce, like ordering and allocation rules, are specified in a workflow specification. A workflow specification (synonyms: workflow design, workflow schema) defines a workflow that behaves according to the rules defined in its workflow specification.

> **Running example.** Throughout this thesis we use the workflow of a small production company as running example (adapted from an example provided by the Workflow Management Coalition [161]). The workflow begins when the company receives an order. Next, the departments Production and Finance are put to work. After the order has been received, Finance checks whether the customer's account limit is not exceeded by accepting the order. If Finance rejects the order, the whole workflow stops. Otherwise, Finance sends a bill to the customer and waits until the customer pays. If the customer does not pay within two weeks, Finance sends a reminder to the customer. If the payment arrives, Finance handles the payment. If the payment is ok, Finance has finished. Otherwise, the customer is notified and another bill is sent to the customer, and the subprocess described before is repeated. After the order has been received, Production checks whether the desired product is still in stock. If not, a production plan is made to produce the product. If according to Finance the order can be accepted, the product is either produced according to the made production plan, or taken from stock. If both Production and Finance have finished, the product is shipped to the customer and the workflow stops.

Like any design, a workflow specification needs to be communicated to different groups of people, like end users, managers, and technical staff. Hence, it is desirable that workflow specifications are written in a language that is known to all these groups. One way to achieve this is to use a standard language. Moreover, it is desirable that the workflow language is graphical. Experience shows that graphical languages are easier to understand by most people than textual ones (consider for example the large amount of graphical languages for specifying systems in engineering sciences [69] and computer science [157]).

Currently, no such graphical standard language for workflow modelling exists. The Workflow Management Coalition, an industrial standardisation body for Workflow Management, has proposed a textual standard language [160], but the purpose of that language is to facilitate the interchange of workflow specifications between software products, not between people. Consequently, models in this language are hard to read by people.

Some people have proposed to use Petri nets as a standard language for workflow modelling [2, 55, 58, 132]. Petri nets are graphical. With Petri nets, business activities and the ordering between business activities can be specified. This aspect

of workflows is called the control-flow dimension or process dimension of workflows in literature [2, 118]. Petri nets offer some support for modelling resources as well (the resource dimension of workflows), but they are not widely used for this purpose.

Recently, another modelling language has been proposed for describing and defining workflows, namely UML *activity diagrams* [150, 59, 71]. The Unified Modeling Language (UML) is a de-facto industry standard consisting of several graphical languages for representing software system designs [150]. The language of activity diagrams is one of these graphical languages. The notation of activity diagrams is based on Petri nets, flowcharts and statecharts. Like Petri nets, activity diagrams are useful for modelling business activities and the ordering between business activities. Activity diagrams offer some limited support for modelling resources, but they cannot model the resource usage of workflows very well. In this thesis, we use UML activity diagrams to model the process dimension of workflows. Figure 1.1 shows the activity diagram of the workflow of the running example. Ovals represent activity states, rounded rectangles represent wait states, and directed edges represent transitions. Further details on the notation are given

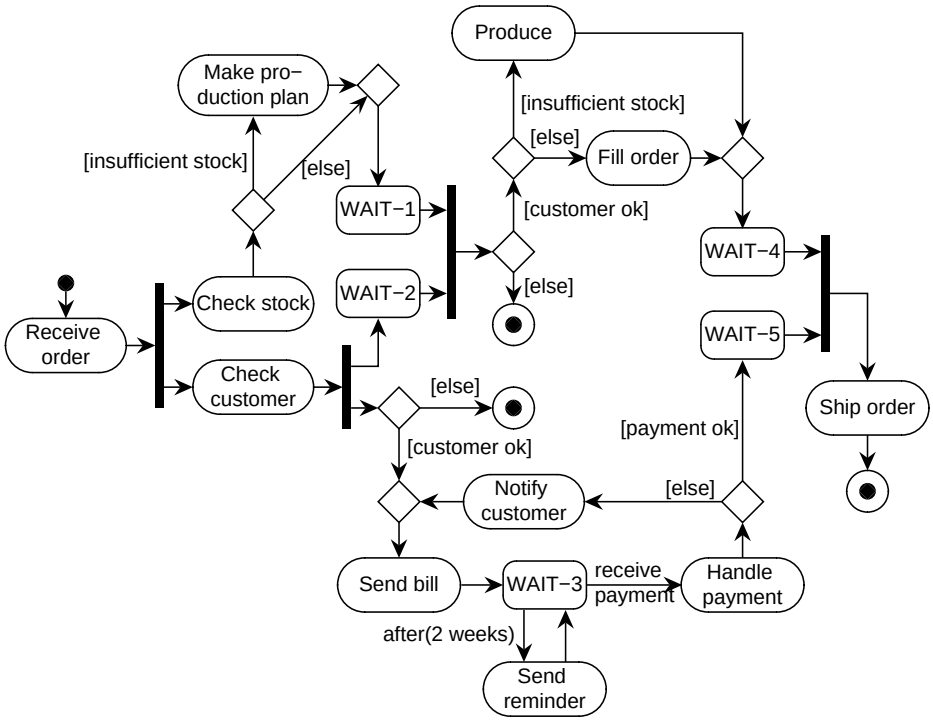



**Figure 1.1** *Workflow of production company*

in Chapter 3.

Any workflow specification has to satisfy several requirements, depending upon the workflow that is being modelled. For example, if the workflow specification is put to use in an organisation, an example requirement could be that its workflow instances do not deadlock. Or in the case of a workflow that handles insurance claims, that a claim is only accepted if it has been checked twice. Such requirements are called *functional requirements*. Functional requirements specify what should be done. They are contrasted to performance requirements, that specify how well something should be done. An example of a performance requirement is: "95% of all workflow instances are handled within one week". Every performance requirement presupposes a functional requirement, since specifying how well something should be done presupposes specifying what should be done as well. In this thesis, we only consider functional requirements.

Functional requirements of workflows can be checked at run-time by inspecting the behaviour of the WFMS, the organisation and the environment of the organisation, such as customers. If some requirement is seen to be violated, the corresponding workflow specification can be repaired. But not only the workflow specification needs to be repaired, the running workflow instances of this erroneous specification must be repaired as well. For example, some part of a running workflow instance may have to be redone, or perhaps the whole workflow instance must start all over. If a workflow specification has a lot of workflow instances or if performing a certain activity is very costly, repairing is very expensive. Moreover, it may displease customers. It is therefore desirable to detect errors in a workflow specification before the workflow specification is put to use.

Some errors can be found by thoroughly testing the workflow specification with some example scenarios before using it in a WFMS. But since a workflow specification can have infinitely many scenarios due to loops and data, not every possible scenario can be tested. The main disadvantage of testing is that, as Dijkstra pointed out [52], it shows the presence of errors, but not their absence. In other words, if the workflow specification passes all tests, it is still uncertain whether or not the workflow specification satisfies the requirement. Another disadvantage of testing is that repairing an erroneous model can still be very costly, since some previous design steps, like mapping a workflow specification to the input language of the workflow management system, may have to be redone completely.

It is therefore desirable to detect errors in a workflow specification at design time, not at run time. There are several ways to do this. The simplest way is to visually inspect a workflow specification. Drawback of inspection, however, is that like with testing not all errors may be spotted. Moreover, workflow specifications can be very complex due to the presence of parallelism, event-driven behaviour, real time, data and loops. If the workflow specification is too complex, visual inspection may not be possible or reliable anymore. In addition, the meaning of a workflow specification is usually ambiguous and unclear, that is, different people may attach a different meaning to a workflow specification. Consequently, different

people might answer the question whether the workflow specification satisfies a particular requirement differently.

A workflow specification can be made unambiguous and clear by attaching a formal, mathematical semantics to the workflow specification. A *formal semantics* maps a workflow specification to a mathematical structure[2]. This mathematical structure we call the formal semantics of the workflow specification. Advantage of a workflow specification with a formal semantics is that it has an unambiguous and clear meaning, because there is only one mathematical structure that is the meaning of the specification. Another advantage of a formal semantics is that it provides a basis for tool support. Tools can mechanically verify requirements on workflow specifications that are so complex that they cannot be verified manually by a person. A formal semantics makes clear what has to be implemented in a tool and does not leave tool implementors with any open issues in the semantics that they have to resolve in order to implement the semantics.

But not every formal semantics is suitable for a workflow specification. The mathematical structure represents the real-world behaviour of the workflow, so that the mathematical structure can be used to analyse the real workflow. So, the mathematical structure must represent the execution of the real workflow *accurately*, otherwise the mathematical structure may satisfy a particular requirement while the real workflow does not, or the mathematical structure may fail to satisfy a requirement whereas the workflow does satisfy this requirement. This relation between actual workflow execution and mathematical structure is depicted in Figure 1.2.

The main problem with UML activity diagrams is that they have no formal semantics. Although the OMG has provided a semantics for UML [150], this semantics is informal. As pointed out above, verification of functional requirements of workflow specifications requires a formal semantics. In addition, the OMG semantics of UML activity diagrams is not entirely suitable for workflow modelling. In other words, a formalisation of the OMG semantics of UML activity diagrams does not represent workflow execution accurately. We show this in Chapter 9.

As Petri nets have a formal semantics, they do not have this problem. In fact, some authors [73] have proposed to just use the Petri net token-game semantics for activity diagrams, as the syntax of activity diagrams resembles the syntax of Petri nets. The formal semantics of Petri nets has been defined, however, independently from workflow modelling. Although there is a lot work done on applying Petri nets to workflow modelling (see Chapter 8), we do not know of any work in literature in which the relation between a Petri net semantics and the real-world behaviour of workflows is sketched. It is therefore unclear whether there is a semantics of Petri nets that resembles the real-world behaviour of workflows. It is true that

[2] In systems engineering [20], this mathematical structure is usually called a 'model'. What we call 'workflow specification' would be called a 'workflow design' in systems engineering. In computer science, however, the term 'model' is used in a broader sense and includes the notion of a semi-formal design (specification) with a formal syntax but with an informal, intuitive semantics. To avoid confusion, we do not use the word 'model' in this thesis.

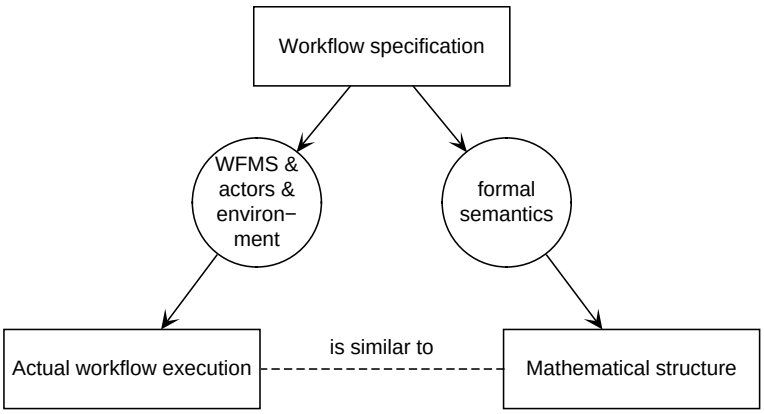



**Figure 1.2** *Relation between workflow execution and formal semantics*

there are some WFMSs that use the syntax of Petri nets for specifying workflows. It is doubtful, however, whether these WFMSs attach the token-game *semantics* of Petri nets to such specifications (Section 8.8). (See Chapter 8 for an extensive comparison of our activity diagram semantics with different Petri net semantics.)

A popular approach to the verification of functional requirements of hardware and software systems is model checking [42, 43]. *Model checking* is a technique for automatically verifying functional requirements of behavioural models. The functional requirements are specified in temporal logic. The checked behavioural models are mathematical structures and should not be confused with workflow specifications. Several tools, called model checkers, exist that implement model checking. Model checkers verify a functional requirement by searching the complete state space of the behavioural model. If the model checker does not find an error, the requirement is certain to hold. If the model checker does find an error, the model checker returns a counterexample in the form of a sequence of states that violates the requirement. For example, if a behavioural model fails to satisfy the functional requirement that it does not contain a deadlock, then the model checker returns a sequence of states that leads to the deadlock state. This feedback of the model checker can help the modeller in finding the error and repairing it.

The most important other technique for formal verification of functional properties is theorem proving [43]. Advantage of model checking over theorem proving is that with model checking, user requirements can be verified automatically without any user interaction, whereas with theorem proving, user interaction may be required. Also, if the functional requirement fails to hold, the model checker returns a counterexample whereas theorem provers do not do this. Finally, model checkers are faster than theorem provers [43].

Disadvantage of model checking is that the model checker can handle finite state spaces only, whereas theorem proving can handle both finite and infinite

state spaces. (The state space of the model must be finite, since model checking is an exhaustive search on the state space of the model.)

But even a finite state space may be too large in practice to be verified by a model checker. This problem is alleviated by symbolic model checkers, which represent the state space symbolically. A symbolic representation of the state space can be model checked more efficiently than an explicit representation. Symbolic model checking has been successfully applied to behavioural models that have over $10^{20}$ states [28].

In this thesis, we use model checking to verify functional requirements.

## 1.2 Problem statement

Having sketched the background of this thesis, we now formulate the goal of this thesis as follows.

> *The goal of this thesis is to define a formal semantics of activity diagrams that is suitable for workflow modelling. The semantics should allow verification of functional requirements using model checking.*

The goal sketched above is still rather vague. We make it more concrete by stating two requirements that must be satisfied by the formal semantics. These requirements help to evaluate the developed semantics and also give guidance in finding a suitable semantics.

*The formal semantics must represent workflow behaviour accurately.*

Since we use activity diagrams to model the process dimension of workflows, we only consider the process dimension of workflow behaviour in this thesis, not the resource dimension. By 'accurately', we mean that the formal semantics of an activity diagram as workflow specification is a realistic and faithful representation of the real-world behaviour of the corresponding workflow. From an inaccurate semantics, no reliable analysis results can be inferred. Then analysis is useless and ineffective. Therefore, the semantics must represent all relevant aspects of workflow behaviour.

*The formal semantics must be "easy to analyse" for a model checker.*

Ideally, to allow for efficient model checking, the state space of the semantics of an activity diagram must be as small as possible.

These two requirements on the semantics are conflicting. An accurate semantics of activity diagrams for workflow modelling deals with all relevant aspects of the execution of workflows in an organisation. It will therefore be more detailed and more difficult to analyse than an inaccurate semantics. Whereas a simple

semantics that is easy to analyse will contain less details, and will therefore be less accurate than a more complex semantics.

We do not put the requirement that the formal semantics must be "easy to use" by a workflow modeller. No matter how praiseworthy such a requirement is, it is hard to validate, and requires a large empirical evaluation of the semantics, which is beyond the scope of this thesis. Nevertheless, we think that a semantics satisfying the two requirements listed above, especially the first one, will likely be easier to use by workflow modellers than a semantics not satisfying these requirements.

An often heard claim is that a semantics should be "simple". The argument is that using a simple semantics complex constructs can be modelled in a structured way. But a simple semantics is not by definition a good semantics, because it can be *too* simple. Then the semantics is inaccurate. Or as the German architect Tessenow once said: "The simplest is not always the best, but the best is always simple". We therefore do not require the formal semantics to be simple.

We will not be concerned with the question how our semantics could be implemented by a state of the art WFMS. The problem with commercial state of the art WFMSs is that the semantics they attach to an input workflow specification is not known. Only by observing the behaviour of the WFMSs at run time, some of the meaning becomes known, but not all of it. And the textual standard language proposed by the Workflow Management Coalition [160] does not even have a formal semantics. It is therefore not possible to relate any formal execution semantics, not even the Petri net one, to the behaviour of a real-world WFMS. Nevertheless, in Chapter 9 we briefly relate the current state of the art WFMSs to our semantics, but not in detail.

## 1.3 Problem solving approach

We solve the problem in five steps.

1. We study the domain of workflow modelling in order to identify requirements on the formal semantics of activity diagrams. The requirements are characteristics of workflow executions that in our opinion should be reflected in the formal semantics. The study is based on literature and on several case studies that we did.

2. We define a formal semantics for activity diagrams that satisfies the requirements identified in the first step. We take the existing execution semantics of statecharts and Petri nets as starting point. Petri nets are investigated because they look like activity diagrams and because they are frequently used for formal workflow specification. Statecharts are investigated because they too look like activity diagrams, but in addition the current UML semantics of activity diagrams is defined in terms of statecharts. Statecharts are not used very often for formal workflow specification.

We define the semantics directly on the syntax of activity diagrams. The alternative is to translate the syntax of activity diagrams into the syntax of a formal language, that already has a formal semantics. This is the approach adopted in almost every other formalisation of UML activity diagrams (see Chapter 9). Drawback of this approach is that the design choices made in the semantics of the formal language are made to apply to UML activity diagram as well. Although in principle there is nothing wrong with this, it may have several awkward consequences for workflow modelling, as we show in Chapter 9 for several of these other formalisations.

3. In fact, we will define two semantics. The first one, called the *requirements-level semantics*, is easy to analyse but somewhat abstract, whereas the second one, called the *implementation-level semantics*, is difficult to analyse but an accurate representation of the behaviour of real workflows. Both semantics are based on existing semantics.

4. We study the relation between the two semantics in order to find out for which activity diagrams and which requirements the requirements-level semantics and implementation-level semantics give similar results. For such activity diagrams and requirements, we can safely use the requirements-level semantics and obtain verification results that also hold for the implementation-level semantics. In addition, we gain more insight in the differences and similarities between the two semantics.

5. We use the requirements-level semantics to verify functional requirements of workflow specifications using model checking. Thus we can evaluate how well that semantics can be used to verify requirements of workflow models. We focus on functional requirements and activity diagrams that are invariant under the requirements-level and implementation-level semantics in the sense as described in the previous item. Thus, the obtained verification results carry over to the implementation-level semantics, even though that semantics is considerably more complex than the requirements-level semantics.

## 1.4 Outline

The remainder of this thesis is structured according to the problem solving approach outlined above.

Chapter 2 explains and analyses the domain of workflow modelling. An interpretation of workflow specifications is presented. We will argue that a workflow specification prescribes the behaviour of a WFMS, but not of the organisation in which the WFMS is embedded.

Chapter 3 defines the syntax of activity diagrams. The underlying structure of an activity diagrams is formally defined. The specification of activities is discussed.

Chapter 4 discusses the design choices made in our semantics. The choices are motivated by the analysis of the domain of workflow modelling that was done in Chapter 2. Both the Petri net and statechart execution semantics are considered as a starting point for the semantics. The statechart execution is chosen, since it fits our purposes best. The requirements-level and implementation-level semantics are informally introduced.

Chapter 5 formally defines the requirements-level and implementation-level semantics. The semantics is defined in terms of transition systems.

Chapter 6 studies the similarities and differences between the requirements-level and implementation-level semantics. A class of activity diagrams that have similar behaviour, at a certain level of abstraction, in the requirements-level semantics and implementation-level semantics, is defined.

Chapter 7 discusses possible semantics for some constructs not considered in the previous chapters. Among others, object flows are discussed.

Chapter 8 compares the two semantics with several Petri net variants. The comparison focuses on the modelling of several aspects of workflow modelling.

Chapter 9 compares the two semantics with other related work, including the informal OMG semantics of UML activity diagrams and formalisations of that semantics. Other formal workflow modelling languages are also discussed.

Chapter 10 discusses tool support that we developed for verification of functional requirements on activity diagrams using model checking. The tool translates an activity diagrams into an input for a model checker according to our requirements-level semantics. If the model checker returns a counterexample, the tool highlights a corresponding path through the activity diagram. The tool allows the verification of workflow specifications that have event-driven behaviour, data, real time and loops. A class of functional requirements that are insensitive to the requirements-level and implementation-level semantics is defined. If a functional requirement in this class is verified against an activity diagram of the class defined in Chapter 6, the result of the verification under the requirements-level semantics is the same as the result under the implementation-level semantics.

Chapter 11 applies the verification approach of Chapter 10 to two real-life case studies, that are based on workflows actually in use in organisations. Lessons learned are discussed.

Chapter 12 contains conclusions, a summary of contributions, and gives directions for future work.

Preliminary results of the research reported in this thesis appear in the following publications [63, 64, 65, 66, 67].

# Chapter 2

# Workflow concepts

In this chapter we explain and analyse the domain of workflow modelling. First we focus on workflows in general, then we zoom in on workflow management systems, and finally we explain how we interpret workflow specifications. In the remainder of this thesis we use the analysis done in this chapter to define and motivate our semantics.

Section 2.1 explains what workflows are. Section 2.2 explains how workflow management systems coordinate workflows. Section 2.3 discusses the architecture of workflow management systems. Section 2.4 focuses on the class of software systems that workflow management systems belong to, the class of reactive systems. Finally, Section 2.5 discusses various possible interpretations of a workflow specification. The first three sections of this chapter are based on literature (among others [2, 31, 76, 81, 82, 103, 118, 135, 159, 161]) and several case studies that we did.

## 2.1 Workflows

The following concepts are important for workflow modelling (see Figure 2.1). A *business process* is an ordered set of business activities. The goal of a business process is to deliver a certain requested service or product. For example, if John Smith sends an insurance claim to his insurance company, then a business process that handles the insurance claim will be started in the insurance company.

A *business activity* or *activity* is an amount of work that is uninterruptible and that is performed in a non-zero span of time. For example, in an insurance company possible activities are *Check credit* and *Handle payment*. Activities are done by *actors*. Actors are persons or machines. A machine can be a computer. A person usually makes use of a computer (machine) to do his activities. If an actor is a machine, no interaction with a person is needed to do the activity. So then the activity is automatic.

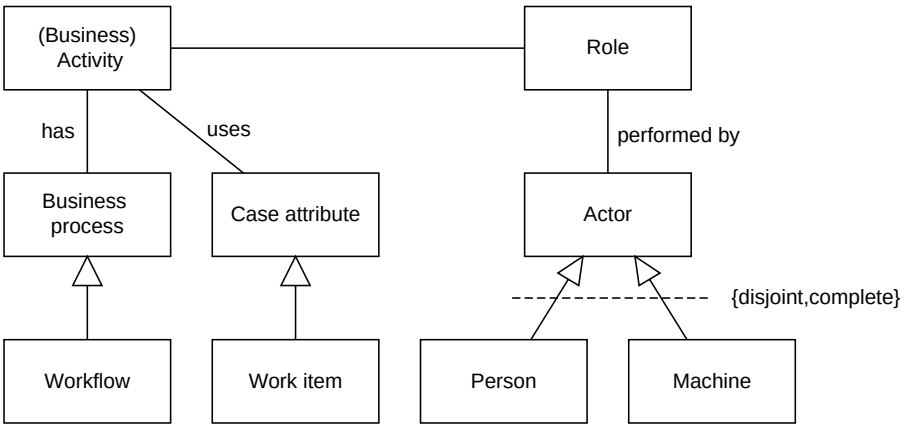



**Figure 2.1** *Workflow concepts and their relation*

A *workflow* is an operational business process. A *workflow specification* defines a workflow. A workflow specification is also called workflow schema or workflow design. The Workflow Management Coalition (WFMC) calls a workflow specification a process definition [161]. An instance of a workflow specification is called a *case*.

In a case, *work items* are passed and manipulated. An example of a case is the process that handles the insurance claim of John Smith. An example of a work item is the claim form of John Smith. More in general, in a case *case attributes* are read and updated. Case attributes are work items and other relevant data of the case. Actors update case attributes in activities. Case attributes, including work items, are usually stored in a database system; see Sections 2.2 and 2.3.

At any moment in time, the case may be distributed over several actors. A distributed part of the case we call a *thread*, borrowing terminology from UML [150]. Each thread has a *local state*. There are two kinds of local state.

- In an *activity state* an actor is executing an activity in a part of the case.
- In a *wait state* the case is waiting for some external event or temporal event.

We allow multiple instances of states to be active at the same time. For example, suppose that two activities $A, B$ that are active in parallel are each followed by the same activity $C$. That is, if $A$ ($B$) terminates, $C$ starts. If both $A$ and $B$ terminate at the same time, the result is that two instances of $C$ will be active at the same time. The *global state* of the case is therefore a bag (synonym: multiset), rather than a set, of local states of the distributed parts of the case.

The definition, creation, and management of cases is done by a workflow management system (WFMS), on the basis of workflow specifications. In this thesis, any computer system that implements workflow management functionality is called

a WFMS. In Sections 2.2 and 2.3, we explain how WFMSs control and coordinate cases using workflow specifications.

Workflow specifications have two important dimensions: the process or control-flow dimension and the resource dimension. The *process dimension* concerns the ordering of activities in time (what activities have to be done in what order). The *resource dimension* concerns the structure of the resources in the organisation (who has to do an activity).

In the process dimension, there are several ordering constructs possible between activities. The most obvious ones are sequence (activity $A$ is done before activity $B$), choice (either $A$ or $B$ is done but not both), parallel composition ($A$ and $B$ are done in parallel), and iteration ($A$ is done zero or more times, after which $B$ is done). Van der Aalst et al. [7] have found other ordering constructs that are also used in existing workflow specifications.

In the resource dimension, the relevant characteristics of the actors are modelled. Actors are grouped according to organisational units they belong to. Actors are also grouped according to roles. A *role* is a set of characteristics of actors [161]. For people, a role can refer to skills, responsibility, or authority. For example, in an insurance company an example role might be *senior officer*. For machines, a role can refer to computing capabilities, for example a machine with the Windows NT™operating system. For each activity a role is specified that states which actors are allowed to do the activity. For example, if an insurance claim of over € 5000 is processed, it might be specified that the activity *Check payment* is done by somebody with the senior officer role. Roles link activities and actors, and thus link the process and resource dimension.

Sometimes, the intermediate concept of a group is used. A group is a set of actors, who all can play the same role. By allocating an activity to a group of actors, rather than to a single actor, actors have more flexibility in selecting their work and a more efficient division of labour can be brought about. To keep the presentation simple, we do not consider groups in this thesis.

In this thesis, we focus on modelling the process dimension of workflows. When we use the term workflow specification, we refer to a specification that describes the process dimension of a workflow. In the remainder of this chapter, we sometimes refer to the resource dimension, i.e. actors, in order to explain the concept of workflow management more clearly.

## 2.2 Workflow management

Before we explain how workflow management systems (WFMSs) coordinate workflows, we discuss the subtle difference between a WFMS on the one hand, and a workflow system (WFS) on the other hand. A WFMS is a general purpose machine; it needs a workflow specification to make it useful for a particular purpose. Compare this to a computer that needs a program to compute something specific and a database management system that needs a database schema to store some

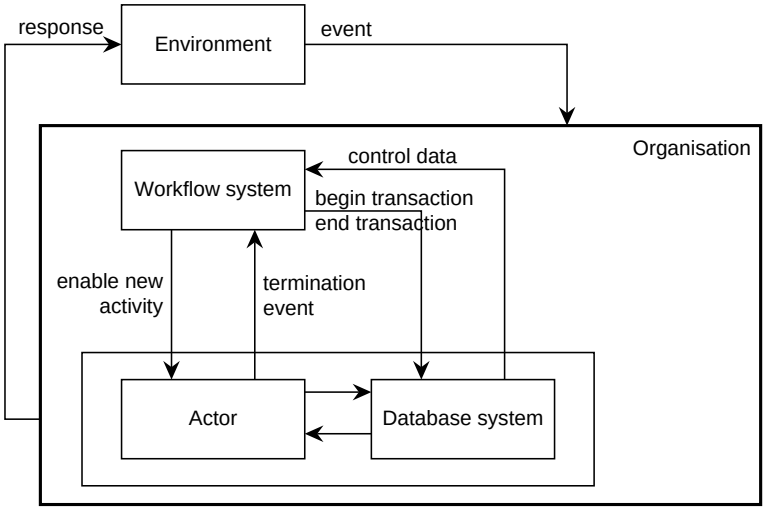



**Figure 2.2** *Function of workflow system. Arrows denote communication*

specific data. A WFMS that is instantiated with one or more workflow specifications is called a workflow system (WFS), just like a database management system instantiated with one or more database schemas is called a database system. Until now, when we used the term 'WFMS', we sometimes intended a WFS. From now on, we will use the term 'WFS' whenever we mean a WFMS instantiated with a workflow specification.

Cases are controlled and coordinated by workflow systems (WFSs). For each case that a WFS controls and coordinates, the WFS ensures that the actors do the activities of that case in the right order, by informing the right actors at the right time that they have to do some activities. Which order is right and which actor is right is specified in the workflow specification with which the WFS was instantiated. Enforcement of rules in a workflow specification by a WFS is called *enactment* [161].

Figure 2.2 shows how a WFS interacts with actors and a database system. Actors, database system and WFS together are part of the organisation. The organisation has an environment with which it interacts. The most important entities in the environment are the customers to which the organisation delivers its products and services.

A WFS informs actors that they may start some new activities. When an actor completes, it notifies the WFS that it has finished its activity. An actor is under the social obligation to only start an activity once the WFS has notified him that he may start the activity. Moreover, an actor should finish an activity he is working on. These may seem obvious constraints or assumptions; nevertheless we state them here explicitly for better understanding of workflow management.

A typical input event for a WFS is an activity termination event. If a WFS receives an activity termination event, the WFS must decide what activities have to be done next. In order to decide this, the WFS needs to know what activities are currently being executed. In other words, the WFS needs to know the state of the case. In the example workflow of the production company (Figure 1.1), the WFS must know that it is in state Receive order when the termination event of activity *Receive order* occurs, in order to make the right decision[1].

In addition, the WFS needs to know what transitions are possible from the current state to another state. The WFS makes a decision by choosing one of the possible transitions and taking it. By taking a transition the state of the case is changed. For example, in the workflow of the production company, there is only one transition leaving state Receive order. Thus, if activity *Receive order* terminates, the WFS can only decide to next enable activities *Check stock* and *Check customer* and update the state of the case from Receive order to Check stock, Check customer.

Next, the WFS needs to allocate the activities to some actors. The WFS first looks at the role specified for each new activity and chooses an actor that belongs to this role. It then allocates the activity to this actor, and informs the actor that it should do the activity.

Both the possible states of a case and transitions between these states are defined in a workflow specification. In addition, the workflow specification contains allocation rules, specified in terms of roles and actors, but we leave them out of the discussion as they do not belong to the process dimension. Using the workflow specification, the WFS maintains the state of each case. If an event occurs, the WFS *routes* the case: it changes the state of the case, i.e., it takes a transition, and informs the relevant actors what activities have to be done next. Typical input events are activity termination events, but other events (e.g. external, temporal) are also possible (see Section 2.4). We come back to the issue of state and transition in Section 2.4

Every case has some data associated with it, for example the name and address of the applicant of an insurance claim. The data is used and changed by actors in activities, and maintained by a database system. A database system ensures that all data it maintains is consistent and stays consistent [56, 152]. For example, if the name of a client is changed, the database system removes the previous name, in order to ensure that each client in the database system only has one name. Moreover, a database system ensures that the data is maintained in a reliable way. Reliability means that if the database system fails or goes down, the data can be recovered.

The unit of consistent and reliable database access is called a *transaction* [56, 152]. An (ACID) transaction [56, 152] is a sequence of update (change) and read (use) operations on a database that must be executed atomically and in isolation

[1] Throughout this thesis, names of activities are written in *italic* whereas names of nodes and states are written in sans serif.

from each other. Execution of an activity corresponds to a transaction: During execution of an activity, the database is accessed by the actor, in order to read or update some data. There are many possible links between an activity and a transaction [79]. One transaction can correspond to one or more activities or even the whole workflow. In this thesis, we make the choice that every activity corresponds to a unique ACID transaction. The ACID transaction is started by the WFS when the activity starts executing; the ACID transaction is ended by the WFS when the activity has terminated. More advanced transaction concepts with relaxed atomicity and isolation, are also possible for workflows, see for example Grefen et al. [83], but we do not consider those here.

Two important properties of transactions are atomicity and isolation. Atomicity means that a transaction is not done partially: either it completes or is undone. Isolation means that two parallel transactions do not interfere with each other: intermediate results of each transaction are hidden from other transactions. An example of interference is when two parallel transactions both update the same address. The WFS can prevent interference by forbidding two interfering transactions, i.e., their corresponding activities, to be active at the same time.

Unlike actors, the WFS does not update data, since it does not *do* activities, but *coordinates* them. Sometimes, however, the WFS uses (reads) data variables to decide which activity should be started next. These variables are stored in the database system. For example, if *Check stock* terminates, the WFS uses boolean insufficient stock in order to decide whether or not *Make production plan* needs to be done. Such data is called *control data* [102]. All data other than control data is called *production data* [102]. A typical example of production data are work items. As production data is not used by a WFS, we do not consider it here. So case attributes represent control data.

The whole organisation is an open system that interacts with its environment. Therefore, the environment can interact with each of the parts of the organisation, as shown in Figure 2.3. For example, a customer of the production company can interact with the company, with employees of the company (the actors), with a database system, and even with the WFS, for example to retrieve the current state of his order.

## 2.3 Architecture of workflow systems

Figure 2.3 shows the architecture of an abstract workflow system; it decomposes the workflow system in Figure 2.2. The architecture is based upon several existing architectures of WFSs [31, 81, 118, 159]. Note that the architecture focuses on the process dimension of workflows; the resource dimension is left out. In this thesis we assume that the WFS controls a single case. A generalisation to a WFS that controls multiple cases is straightforward under the assumption that cases do not interact with each other. The main components of the WFS are the queue and the router. The environment interacts with the WFS by putting events in the queue.

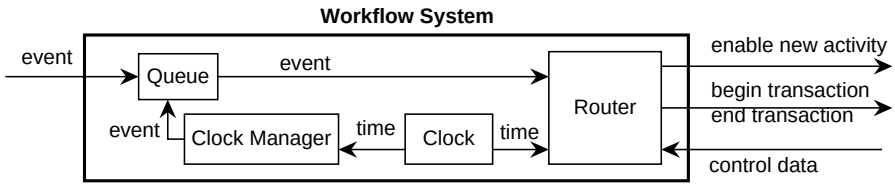



**Figure 2.3** *Abstract workflow system architecture*

On basis of these events and the current state of the case, the router component of the WFS routes the case as prescribed by the workflow specification of the case. As a result, some new activity instances can be started. All attributes of a case, i.e., both production and control data, are stored in the database (see Figure 2.2). The state of the case is maintained by the WFS itself. In this thesis, we do not treat the state of the case as a case attribute; instead, it is a kind of meta attribute of the case. Scheduled timeouts are maintained and raised by the clock manager on basis of an internal clock.

Note that the case attributes are updated during an activity by the actors, not by the WFS. For example, an actor may update a work item by editing it with a word processor. But the transitions between the states of the case (activity or wait) are performed by the WFS, not by an actor. By taking these transitions the WFS routes the case.

## 2.4 Reactive systems

Workflow systems are reactive systems. A *reactive system* is a software system that maintains an ongoing interaction with its environment [91, 122, 157]. The goal of a reactive system is to create certain desirable effects in its environment by reacting to input events in a certain way. For a WFS, characteristic input events are activity termination events, in Figure 1.1 for example the termination of activity *Receive order*. A WFS uses other events as well, that we will describe below. And a characteristic desirable effect of a WFS to an activity termination event is the enabling of new activity instances, for example enabling *Check stock* and *Check customer* if *Receive order* terminates, and informing the appropriate actors that they have to do some new activities.

The reaction of a reactive system often depends upon the current state of the system and the input event that is responded to. As we saw in Section 2.2, this is also true for WFSs. For example, in Figure 1.1, if event receive payment occurs, then activity *Handle payment* will only be started if the case is in state WAIT-3. Moreover, the reaction may leave the system in a different state than it was before. So in the activity diagram of Figure 1.1, if event receive payment occurs while the case is in state WAIT-3, the next state of the case will be Handle payment. The

| Event | State | Condition | Action | Next state |
|---|---|---|---|---|
| *Receive order* terminates | Receive order | | enable *Check stock*, *Check customer* | Check stock, Check customer |
| *Check stock* terminates | Check stock | insufficient stock | enable *Make production plan* | Make production plan |
| *Check stock* terminates | Check stock | ¬ insufficient stock | | WAIT-1 |
| receive payment | WAIT-3 | | enable *Handle payment* | Handle payment |

**Table 2.1** *Some ECA rules for workflow of production company (Figure 1.1)*

WFS reaches this state by routing the case.

Reactive systems are often contrasted to transformational systems [91, 122, 157]. A *transformational system* computes an output from an input. The output only depends upon the input and not upon some internal state of the system. The output does not have any intended effect upon the environment. A transformational system only interacts with its environment to collect input and to deliver output. During computation, the interaction of a transformational system with its environment is not important.

Reactive systems should also be contrasted with active systems. An *active system* decides itself when it will do something and what it will do then: it does not respond to its environment. An active system has therefore no interaction with its environment, i.e., it is closed, whereas a reactive system is open. Nevertheless, parts of the active system can interact with each other, in other words, they can react to each other. At a suitable level of abstraction, a reactive system composed with an environment *is* a closed, active system.

The behaviour of a reactive system can be described using reaction rules of the form event-condition-action (ECA) [157]. An ECA rule e-c-a has the following meaning:

> if event e occurs and condition c is true, then do action a

The condition can refer to an internal state of the system. Actions can update the internal state of the system. Table 2.1 shows some ECA rules for the workflow of the production company. Since we are only concerned with the process dimension of workflows, the selection and notification of actors (the resource dimension) is not shown as part of the action.

Note that second and third entry of Table 2.1 together represent a choice. Although the outcome of this choice is presumably influenced by the environment of the WFS, i.e., attribute insufficient stock is updated in one of the activities, the choice is made by the WFS.

ECA rules implement the ordering rules mentioned in Section 2.1, thus enforc-

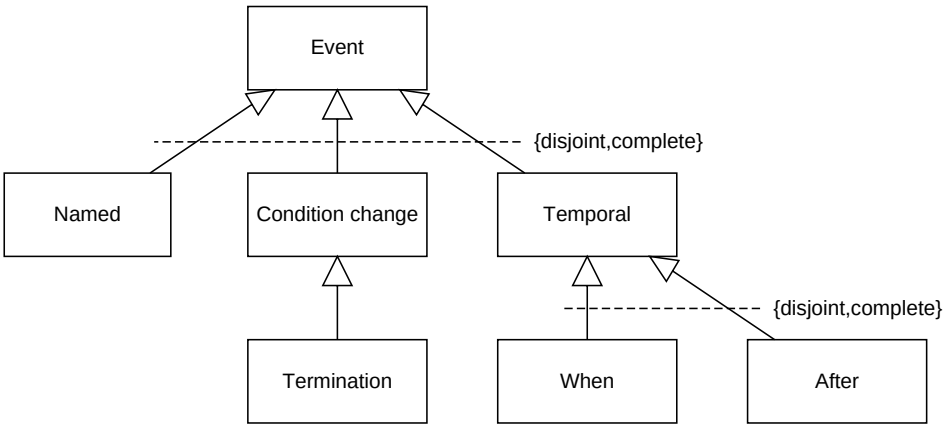



**Figure 2.4** *Event classes*

ing sequence. In Chapter 3, we will see that ECA rules are also used in activity diagrams to label the edges.

We now describe the three components of an ECA rule in more detail.

**Events.** An *event* is some instantaneous, discrete change in the state of the world [157]. Examples of events are pushing a button, receiving a letter, picking up the phone. Since events are instantaneous, they occur at a point in time; they do not have any duration. An event can also be modelled in UML; there it is called a *signal* [150].

There are several kinds of events, some general and some particular for workflows (see Figure 2.4). First, the change that the event signifies can be referred to by giving a name to the change itself or to the condition that changes [157]. This gives us two kinds of events. Both kinds can be specified in UML [150].

- A *named event* is an event that is given an unique name [150, 157]. For example, in Figure 1.1 receive payment is an external event.
- A *condition change event* is an event that represents that a boolean condition has become true[150, 157]. To represent condition change events, UML uses the keyword when. For example, when(c) is the condition change event that represents that condition c becomes true.

Second, a *temporal event* is a moment in time to which the system is expected to respond, i.e. some deadline [150, 157]. In the UML, there are two kinds of temporal events, relative and absolute. Relative temporal events are specified with the after keyword whereas absolute temporal events are specified with the when keyword. The after($n$) event expression, where $n$ is a positive integer, means that $n$ time units after the source of the edge was entered a timeout is generated.

For example in Figure 1.1, two weeks after node WAIT-3 is entered, a timeout is generated. The when($t$) event expression, where $t$ represents a set of points in time, means that the first timepoint from now that is in $t$, a temporal event is generated. For example, if today is March 15, then the temporal event of when(first of month) will be generated at April 1.

Third, a typical kind of event in a workflow is a *termination event*, which denotes that a certain activity has terminated (it is not important who the actor was). Above, we have already seen examples of termination events. Termination events can be seen as a special kind of condition change event; it will be convenient, however, to treat them separately from condition change events. Termination events cannot be denoted in UML.

A useful distinction between events is whether they are generated by the environment or by the system. If an event is generated by the environment of the system, so outside the system, it is *external*. If an event is generated by the system, it is *internal*. An event is either external or internal but not both.

Finally, an event can be either broadcast or point-to-point. A *broadcast event* $e_b$ can trigger arbitrarily many ECA rules with event label $e_b$ at the same time, whereas a *point-to-point event* $e_p$ can trigger at most one ECA rule with label $e_p$.

**Conditions.** Conditions are boolean expressions. They can only refer to the internal state of the case and to control data, i.e., to some case attributes. Evaluating a condition should not have a side effect.

**Actions.** The only actions we consider are changing the state of the case, notifying the appropriate actors that they have to do some new activities, and generating internal events. In Section 2.5, we will see that the informing of actors can be abstracted from. We do not consider update actions on case attributes, as case attributes are updated by actors in activities, not by a WFS.

## 2.5 Interpreting workflow specifications

In this thesis we use activity diagrams to specify workflows. Since the goal of the thesis is to define a semantics for activity diagrams, we must devote some attention to the question what actually the meaning of a workflow specification is. In other words, what does a workflow specification specify?

As explained in Section 2.3, a workflow specification defines ordering rules, in this case ECA rules, between business activities. The ordering rules prescribe how a WFS must behave in order to ensure that activities are done in the right order. The meaning of a workflow specification should therefore be defined in terms of a WFS.

Adopting this interpretation has some subtle consequences. We can illustrate this best by referring to the definition of activity state. As explained in Section 2.1, in an activity state of a workflow, an actor is busy executing an activity. This

description takes the perspective of the organisation. From the perspective of the WFS, however, an activity state of a workflow is a state in which the WFS has enabled some activity by informing some actor that he can start the activity. The WFS *waits* in the activity state for an activity termination event, that signifies that the actor has completed the activity.

Consequently, for the WFS, the activity is a kind of black box. Only the outcome of an activity has an effect on the execution of the workflow; how this outcome is reached is not interesting from the perspective of a WFS. Activities are therefore specified declaratively. An imperative specification would imply that the WFS does the activity. But the outcome of an activity is not computed by the WFS. We therefore specify activities declaratively as black boxes.

Another consequence is that we abstract from the fact that an actor may not be immediately available to do a certain activity. If a WFS enters an activity state, it merely informs the relevant actor that he should do the corresponding activity, but the actor does not have to start immediately. During analysis, we will assume that the actor behaves in a fair way: he will eventually execute the activity.

Putting this one step further, we altogether abstract from actors. We just assume that an activity, once enabled, will terminate. That an actor is needed to perform an activity is irrelevant for the process dimension of workflows.

The most important consequence, however, is that the workflow specification specifies a reactive system, since a WFS is a reactive. Usually another interpretation is adopted, namely that a workflow specification specifies an active system. For example, if a Petri net is used to model a workflow, representing activities by transitions, it is tacitly assumed that an active system is modelled: the transitions in a Petri net are active transitions, not reactive ones (see Chapter 8). Activities then are white boxes rather than black boxes. Under such an interpretation, a workflow specification presumably specifies the behaviour of the entire organisation, including the WFS, *and* its environment, rather than only the WFS. (A Petri net cannot model only the organisation, since even an organisation is reactive: it interacts with its environment, for example customers.) It may be clear that it is far more difficult to model a complete organisation and its environment accurately than to model a single WFS accurately. We therefore do not interpret a workflow specification as specifying an active system.

Summarising, states in a workflow specification are states of a WFS during which it waits for some events. When the events occur, the ECA rules of the workflow specification tell the WFS what it should do. Activities are specified declaratively as black boxes. Actors are abstracted from.

# Chapter 3

# Syntax of activity diagrams

This chapter introduces the syntax of activity diagrams, as described in the current version (1.4) of UML [150]. The semantics, the mapping from the syntax to a mathematical structure, will be defined in the next chapters.

Section 3.1 explains the syntactic constructs that can be used in UML activity diagrams. Section 3.2 introduces the notion of an activity hypergraph as the underlying syntactic structure of a UML activity diagram. Section 3.3 explains how an activity diagram maps into an activity hypergraph. Finally, Section 3.4 explains how activities are specified. The first section is based on the UML standard [150].

## 3.1 Syntactic constructs

An activity diagram is a directed graph, consisting of nodes and directed edges. The most commonly used nodes are shown in Figure 3.1. An activity diagram models the behaviour of a system. A node represents a state of the system. (Our terminology differs slightly from literature, where the term 'state' is used instead of 'node'; and likewise 'transition' instead of 'edge'. Since the terms 'state' and

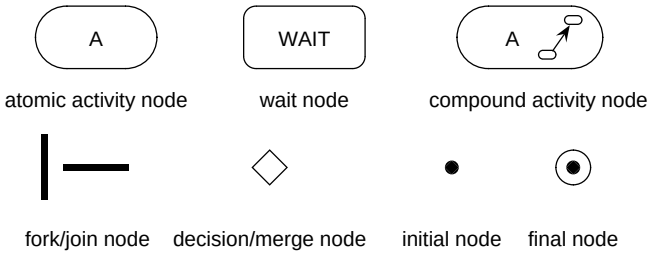



**Figure 3.1** *Activity diagram nodes*

'transition' denote a semantic concept, not a syntactic one, we prefer the terms 'node' and 'edge'.)

In an atomic activity state[1] the system waits for termination of an activity that has been enabled upon entry of the state. In a wait state the system waits for the occurrence of an event, e.g. some deadline occurs or a customer sends some additional information. A wait state is also used for synchronisation of a thread with other parallel threads; in the wait state the system then waits for the completion of the other parallel threads. We come back to this issue on page 28. In a compound activity state another activity diagram is executed. This other activity diagram is started when the compound state is entered. When the activity diagram finishes, the compound activity state is left.

The system starts in the initial state and ends in one or more final states. A final state means local termination of the corresponding thread; other parallel threads can still continue to execute. Using a fork (a bar with one incoming edge and more than one outgoing edge) a thread can be split in several parallel threads. Using a join (a bar with more than one incoming edge and one outgoing edge) multiple parallel threads can be merged into one thread. In a decision (a diamond with one incoming edge and more than one outgoing edge) one of the outgoing edges is chosen, if the incoming edge is taken. In a merge (a diamond with more than one incoming edge and one outgoing edge) the outgoing edge is taken, if one of the incoming edges is taken.

Nodes are linked by directed edges, that represent sequence. We will use the term 'edge' throughout this thesis to stand for directed edge. The node that the edge leaves is called the *source*; the node that the edge enters is called the *target*. The edge always points at the target. An edge is labelled with an ECA rule $e[c]/a$, where $e$ is an event expression, $c$ a guard condition expression, and $a$ an action expression. Events are also called signals in UML. Each of these three components is optional. An edge with label $e[c]/a$ has the following meaning: If the system is in the source state, the event $e$ occurs, and the guard condition $c$ evaluates to true, then the source state is left, the actions $a$ are performed, and the target state is entered. Since the transition is triggered by the occurrence of $e$, event $e$ is called the *trigger event* or simply *trigger* of the edge in UML [150]. Note the similarity with the ECA rules discussed in Section 2.4.

Sometimes we give an edge a name for ease of identification. We write this name in front of the ECA label, followed by a colon. For example, $e1 : e[c]/a$ labels an edge with name $e1$ and ECA label $e[c]/a$.

We now discuss events, guards, and actions in more detail. The events that can be specified in activity diagrams we already listed in Section 2.4. Termination events are not specified explicitly in an activity diagram. We use the convention that an edge leaving an activity node $\mathsf{A}$ and having no visible event label, is implicitly labelled with a termination event that signifies that the enabled activity $A$ has terminated. Below, we will forbid that an edge that leaves an activity state

[1] UML 1.4 calls an activity node an action state [150].

has any other event expression in its label, since that would denote an interrupt, whereas an activity cannot be interrupted, since it is atomic.

A guard expression is a boolean expression that can refer to local variables of the activity diagram. The local variables of an activity diagram are booleans, integers and strings. Guard expressions can be combined using the logical operators $\wedge$, $\vee$ and $\neg$ . Special guard expressions are the in and else predicates. Predicate in(*name*), where *name* is a node name, is true if and only if the system is in state *name*. Predicate else can only be used to label an edge that leaves a decision node (represented by diamond). It cannot be combined with other guard expressions, so for example [else $\wedge$ x=10] is not a guard expression. Predicate else abbreviates the negation of the disjunction of the guard labels of the other edges that leave the decision node. For example in Figure 1.1, the else predicate on the edge that enters node Notify customer abbreviates not(payment ok). If an edge does not have a guard expression, the edge is implicitly labelled with guard expression [true].

Actions are done by the system if and when it takes the edge. The only action expressions we allow are sets of send event action expressions. The events in the set are generated if the edge is taken. Other action expressions in an edge label would change the case attributes. But case attributes are changed by actors in activities, not by the WFS. We therefore do not allow any other action expressions.

The UML defines some extra symbols for event sending and reception. The sending of an event can be represented by a convex pentagon that looks like a rectangle with a triangular point on one side. The label of the symbol specifies which event is sent. Figure 3.2(a) shows an example. The event sending symbol is similar to a wait node whose outgoing edge generates the event. The activity diagram in Figure 3.2(a) is equivalent to the activity diagram in Figure 3.2(b).

The reception of an event can be represent by a concave pentagon that looks like a rectangle with a triangular notch in one of its sides. The label of the symbol specifies which event is received. Figure 3.2(c) shows an example. This symbol is similar to a wait node whose outgoing edge is triggered by the event. The activity diagram in Figure 3.2(c) is equivalent to the activity diagram in Figure 3.2(d).

The notation presumes that a wait node only has one outgoing edge. If a wait node has more than one outgoing edge, such as node WAIT-3 in Figure 1.1, then the notation cannot be used for this node.

We will not use these extra symbols in the remainder of this thesis.

**UML constructs removed.**

- Update actions on edges. As explained above, an update action would denote an update on a case attribute. But case attributes are updated in activities by actors, not in transitions by the WFS. We therefore do not have update actions on edges. Note that technically speaking, update actions can be added without a problem to the syntax and semantics, along similar lines as in existing semantics for statecharts [46, 62].

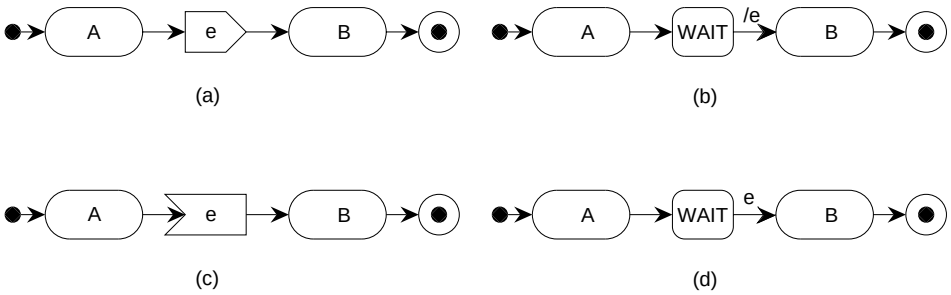



**Figure 3.2** *Abbreviations for event sending and event reception*

Note that the state of the case is not represented as a case attribute. The WFS updates the state of the case while routing the case.

- Synchronisation states synchronise parallel threads. A synchronisation state is similar to a wait state; the only difference is that it has a bound whereas as a wait state has not. The bound limits the number of outgoing edges that can be taken, but does not limit the number of times the wait state is entered [150].

  Synchronisation states can be added without a problem, but we do not consider them here. We did not need them in our case studies.

- Do-activities in UML 1.4 are activities performed by the software system. In our case, the software system is a WFS. The activity states in an activity diagram then represent execution of business activities. Business activities are done by actors, not by the WFS. So we do not use do-activities to model business activities.

- Swimlanes allocate activities to software packages, actors, or organisational units. We do not consider swimlanes, since they are part of the resource dimension of workflow modelling and do not impact the execution semantics of activity diagrams. We here assume that enough resources are available to carry out every enabled activity eventually.

- An object flow node represents an object in a certain state. Objects are processed in activities. An object can be input to an activity, represented by a dashed edge from an object flow node to an activity node, or output from an activity, represented by a dashed edge from an activity node to an object flow node.

  The semantics of object flows and object flow nodes in the new UML version (2.0) will differ considerably from the semantics in version 1.4 of UML, current at the time of writing (2002). We explain this in Chapter 7. Objects are not control data. So object flows and object flow nodes are not needed

to model the process dimension of workflows. We come back to object flows in Chapter 7.

- Other constructs, like deferred events and dynamic concurrency, we discuss in Chapter 7.

**Pseudo nodes and compound edges.** Although we stated at the beginning of this section that nodes represent states, this is not true for all nodes in an activity diagram. The fork and join node and the decision and merge node are called *pseudo nodes* in UML[2] [150]. Pseudo nodes do not represent system states but are syntactic sugar used to glue edges together. A set of edges that is glued together is called a *compound edge* in UML [150].

Compound edges have the following two properties. First, a compound edge is atomic: either all edges in the compound edge are taken, or none. Second, all edges in the compound edge are taken at the same time. That is, it is not allowed to first take the first part of a compound edge, then wait for some event, and then take the second part.

The intended meaning of a fork or join node and decision or merge node is as follows. Both a fork node and a join node are AND nodes. All incoming and all outgoing edges of an AND node belong to the same compound edge and thus are taken simultaneously. Both a decision node and a merge node are OR nodes. Precisely one of the incoming and one of the outgoing edges of the OR node belong to the same compound edge and thus are taken simultaneously. In Section 3.3 we discuss this in more detail.

Like an ordinary edge, a compound edge has an ECA label. The ECA label is derived from the edges contained in the compound edge as follows.

- The trigger event of the compound edge is the trigger event of one of the component edges. Below we require that at most one edge in the compound edge has a trigger event.
- The guard of the compound edge is the conjunction of the guard conditions of all the edges in the compound edge.
- The set of send actions of the compound edge is the union of each individual set of send actions of every edge in the compound edge.

By taking the edges in a compound edge, some non-pseudo nodes are left (the sources) and other non-pseudo nodes are entered (the targets). We call these nodes the sources and targets of the compound edge. To describe the effect of taking a compound edge only the sources and targets of the compound edge need to be known. The edges that are part of the compound edge and the pseudo nodes that link them can be abstracted from. So, we can model a compound edge as a

[2]UML defines initial and final nodes also as pseudo nodes, but since these do not glue edges together, we regard them as real nodes in this thesis.

transition from some sources to some targets. Such a transition is called a directed hyperedge, or *hyperedge* for short.

The difference between a (directed) hyperedge and a (directed) edge is that a hyperedge can have more than one source and more than one target whereas an edge always has a single source and a single target. So an edge can be seen as a hyperedge with a single source and a single target. Like an edge, a hyperedge is labelled with an ECA rule. The ECA rule of a hyperedge is the ECA rule of the corresponding compound edge.

Due to compound edges, the syntactic structure underlying a UML activity diagram is not a graph, but a hypergraph. A hypergraph is graph with hyperedges instead of edges. In Section 3.2, we will formally define the hypergraph structure that underlies a UML activity diagram. In Section 3.3, we will explain how an activity diagram maps into a hypergraph.

**Concurrent termination.** With a join, two or more parallel threads can be merged into one thread. Both the incoming and outgoing edges of the join are part of the same compound edge and thus are taken at the same time.

If more than one of the sources is an activity node, this may lead to awkward situations. As explained above, the intended meaning of an activity node is that the system waits for termination of an activity that has been enabled upon entry of the state; the activity node is left when the activity terminates. But when two activity nodes are both sources of the same compound edge, this meaning no longer holds.

For example, in Figure 3.3(a) activities $A$ and $B$ are active in parallel. Both need not terminate at the same time. So suppose $A$ terminates before $B$. Then ideally node A should be left, since in A the system waits for the termination of activity $A$. But the other edge that leaves B is not yet enabled.

There are two options for the system: either staying in A, or leaving A and not yet entering the join node. With the first option, the system is still in A even though $A$ has terminated. That is not desirable, since the intended meaning of an activity state is that the corresponding activity has not yet terminated; the state is left when the corresponding activity terminates. We therefore reject the first option. With the second option, the system must stay somewhere between A

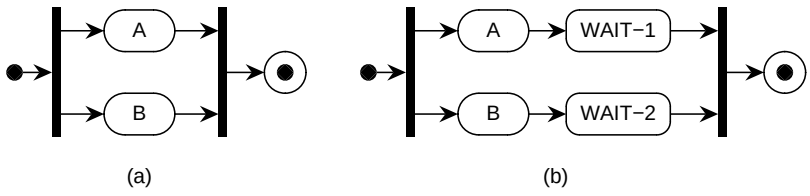



**Figure 3.3** *Concurrent termination*

and the join. Then, the state of the system is apparently the edge leaving A and entering the join node. But an edge is not a state: a system cannot 'be' in an edge. An edge merely denotes sequence between states, but is not a state itself. We therefore reject the second option too.

Hence we forbid the activity diagram in Figure 3.3(a). A corrected version is shown in Figure 3.3(b). When either $A$ or $B$ terminates in the corrected workflow model, immediately the corresponding activity node is left. When both activities have terminated the final state can be reached. It is possible to introduce the convention that Figure 3.3(a) abbreviates Figure 3.3(b). But to be totally explicit about the semantics of activity diagrams, we will not do that.

We therefore require that if a compound edge leaves an activity node, it does not leave any other node. The constraint ensures that if an activity terminates, the corresponding activity node can be left by taking a compound edge. An activity diagram not meeting this constraint can be easily transformed in an activity diagram that does meet these constraints by inserting an extra wait node after every appropriate activity node, as illustrated in Figure 3.3. In the next section we will formalise this constraint. The UML standard [150] does not impose this constraint.

**Constraints on syntax.** Constraints labelled **UML** are defined in the UML standard [150]. All other constraints are defined by us.

1. **UML** Every edge that leaves an activity node or a compound activity node is only triggered by a termination event, i.e., it does not have a visible event label.

   Event expressions other than termination events would denote an interrupt, whereas an activity cannot be interrupted, since it is atomic.

2. **UML** If two edges both leave the same decision node, then they do not both have label [else].

   Otherwise the predicate else would be ill-defined.

3. **UML** Every pseudo node has at least one incoming and at least one outgoing edge, except the initial node (only has outgoing edges) and the final node (only has incoming edges).

   This constraint prevents that the system gets stuck while taking a compound edge.

4. **UML** Every edge that leaves a pseudo node does not have a trigger event.

   **UML** Every edge that enters a join node does not have a trigger event.

   These constraints ensure that a compound edge is triggered by a single event.

5. **UML** Every edge that leaves an initial node does not have a trigger event.

The initial state should be immediately left when the activity diagram is started. If an edge that leaves an initial node would have a trigger event, the initial state may not be immediately left, since the trigger event may not occur.

6. Between a join or fork node and any other kind of node, there can be at most one edge. The UML has a softer constraint: it is allowed to have more than one edge between a join or fork node and another pseudo node.

   This constraint rules out activity diagrams in which a join gets stuck. For example, in Figure 3.4 the join requires that both edges leaving the decision node are taken whereas the decision only allows one edge to be taken. We therefore forbid the activity diagram in Figure 3.4, even though the UML accepts it.

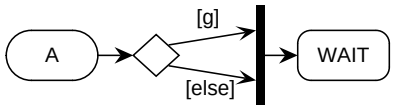



**Figure 3.4** *Ill-defined join*

This constraint also rules out activity diagrams in which by taking a fork node the same node is entered twice. For example, the activity diagram Figure 3.5(a) is forbidden, since the same node B is entered twice. But this is not a serious restriction. Figure 3.5(b) shows an activity diagram that does not violate this rule and that has the same intended meaning as the activity diagram Figure 3.5(a): after activity $A$ completes, two instances of $B$ are started. In general, activity diagrams in which taking a fork enters a certain node $n$ twice can be simulated by activity diagrams in which the fork enters $n$ and a copy of $n$ with the same label as $n$.

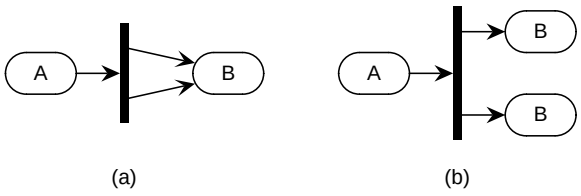



**Figure 3.5** *Example of forbidden fork and allowed fork*

UML 1.4 has adopted a constraint that every activity diagram must have balanced forks and joins, that is, every fork must be eventually followed by a join, and multiple layers of forks and joins must be well nested. This constraint ensures

that every activity diagram can be translated into a UML statechart, which gives the semantics of an activity diagram in UML 1.4. UML 2.0, under development at the time of writing, will not adopt this constraint, as it will define the semantics of activity diagrams independently from statecharts. We too have not adopted such a constraint, as it restricts the concurrency (parallelism) that can be expressed in an activity diagram (see Section 9.1).

## 3.2 Activity hypergraphs

In the previous section (page 27), we saw that a compound edge corresponds closely to a hyperedge. Thus, we can model an activity diagram with its compound edges as a hypergraph: a graph with hyperedges instead of edges. In this section, we define such a hypergraph structure, called an activity hypergraph. This structure is not defined in the UML standard. The activity hypergraph is the syntactic structure for which we will define two execution semantics. Figure 3.6 shows the activity hypergraph that underlies the activity diagram in Figure 1.1. Section 3.3 explains how an activity diagram maps into an activity hypergraph.

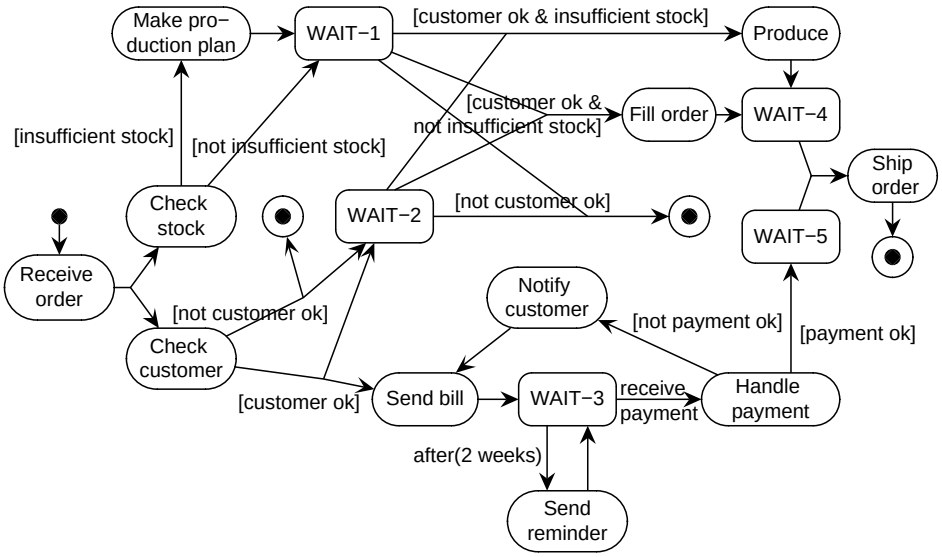



**Figure 3.6** *Activity hypergraph of activity diagram in Figure 1.1*

**Syntax of activity hypergraphs.** We assume there is a set of activities *Activities* and a set of events *Events*. Set *NamedInternalEvents* is defined below; it is a subset of *Events*.

An *activity hypergraph* is a tuple $(Nodes, Events, Guards, HyperEdges, LVar)$ where:

- *Nodes* is the set of nodes,
- *Events* is the set of event expressions,
- *Guards* is the set of guard expressions,
- *HyperEdges* is the transition relation between the nodes of the activity diagram,
- *LVar* the set of local variables. Every variable in a guard expression is a local variable.

We now discuss these elements in more detail. Set *Nodes* is partitioned into set *AN* of activity nodes, set *WN* of wait nodes, set *FN* of final nodes, and one initial node *initial*. Every activity node has an associated activity, denoted by the function $act : AN \longrightarrow Activities$, that is enabled upon entry of the node. In Section 3.4 we discuss how activities can be specified. To distinguish between an activity and an activity node, activities are written in *italic* whereas activity nodes are written in sans serif. We use the convention that in the activity diagram, an activity node $a$ is labelled with the (name of the) activity $act(a)$ it controls. So we have for example $act(\mathsf{Check\ stock}) = \mathit{Check\ stock}$. Note that different activity nodes may have the same label, since they may enable the same activity. Wait nodes are labelled WAIT with an additional number for ease of reference.

Set *Events* is partitioned as follows. There are several kinds of events: named external events, named internal events, condition change events, termination events, and temporal events (cf. Section 2.4). These event types are mutually exclusive: an event has only one type. We partition set *Events* in six subsets: *NamedExternalEvents*, *NamedInternalEvents*, *ConditionChangeEvents*, *TerminationEvents*, *WhenEvents* and *AfterEvents*. Sets *WhenEvents* and *AfterEvents* together comprise the set of temporal events. Note that an event in *WhenEvents* is a temporal event, not a condition change event.

A termination event denotes that a certain activity node has terminated. By function *act*, the termination event indirectly also denotes which activity has terminated. The bijective function *term* defines for each termination event the activity node that has terminated.

$$term : TerminationEvents \rightarrowtail\!\!\!\!\!\rightarrow AN$$

A termination event refers to an activity node and not directly to a terminated activity for the following reason. An activity can belong to more than one activity

node, i.e., more than one activity node can enable the same activity. So, more than one instance of the same activity can be active at the same time. Then if one activity instance completes, only knowing which activity terminates does not provide enough information to decide which node should be left.

As explained in Section 2.4, events can be either external or internal, but not both. Whether an event is external or internal depends upon its type. Named events can be either external or internal. Condition change events and termination events are always external. Temporal events are a special case: they could be considered either as external or internal. We consider them as external here.

We also explained in Section 2.4 that an event can either be broadcast, triggering possibly more than one compound edge, or point-to-point, triggering at most one compound edge. Function *sendtype* : *Events* $\longrightarrow \{\mathsf{bc}, \mathsf{p2p}\}$ specifies for each event whether it is broadcast or point-to-point.

Every activity termination event is point-to-point, since it denotes completion of one instance of an activity node. Every timeout event (after) is point-to-point, since the timeout is generated some finite amount of time after the corresponding compound edge became relevant. Every global temporal event (when) is broadcast. Named events and condition change events can be either point-to-point or broadcast. In this thesis, we use the convention, however, that they are broadcast. This convention is also adopted in UML.

Table 3.1 summarises the kind of events and their properties. Completion events denote completion of a wait node. They are only needed in the implementation-level semantics, which we define in Section 5.4. For a motivation why they are needed, see Section 5.4. Although completion events are not used in the syntax of activity hypergraphs, for the sake of completeness, we have included them here.

Set *Guards* is the set of all boolean expressions on set *LVar*, using boolean connectors $\wedge$, $\vee$, $\neg$ . Special guard expression is in(*name*), which is true if and only if the system is in node *name*.

Hyperedges have several parameters. For every hyperedge $h \in$ *HyperEdges*, *source*($h$) is a non-empty set of source nodes, from which the hyperedge $h$ departs. The source nodes are left if $h$ is taken. Symmetrically, *target*($h$) is a non-empty

| Event type | external | internal | broadcast | point-to-point |
|---|---|---|---|---|
| named | x | x | x | (x) |
| condition change | x | | x | (x) |
| termination | x | | | x |
| when (temporal) | x | | x | |
| after | x | | | x |
| completion | | x | x | (x) |

**Table 3.1** *Event properties. An entry '(x)' is possible, but not considered in this thesis*

set of target nodes, which are entered if the hyperedge $h$ is taken.

Like every edge in an activity diagram, every hyperedge $h$ in an activity hypergraph is labelled with an ECA rule. The trigger event of $h$ is denoted by $event(h)$. Below, we will put the constraint that a hyperedge cannot be triggered by more than one event. Thus, a hyperedge has either no trigger event or only one trigger event. We use symbol $\perp$ as a special event label to denote that a hyperedge has no trigger event. So $event(h) \in Events \cup \{\perp\}$.

Every hyperedge $h$ has a guard expression (possibly [true]), denoted $guard(h)$. Guard expression $guard(h)$ should be valid, i.e. $guard(h) \in Guards$.

Moreover, every hyperedge $h$ has an action expression. The only action expressions we allow in hyperedges are sets of send event actions; other action expressions would change the case attributes, which is undesirable, as case attributes are changed in activities. Given a hyperedge $h$, the set of events generated by taking $h$ is denoted $sendactions(h)$. Every hyperedge $h$ can only generate named internal events: $sendactions(h) \subseteq NamedInternalEvents$.

Local variables in $LVar$ represent the case attributes. As explained on page 16, case attributes represent control data.

**Note.** In the remainder of this thesis, we will use the term 'activity diagram' when we actually mean an activity hypergraph. And we will show activity diagrams when we actually intend to show activity hypergraphs. This is not harmful, however, as each activity diagram maps into a unique activity hypergraph (see Section 3.3).

**Constraints on activity hypergraphs.**

1. For every hyperedge $h$ that has activity node $a$ as source, $a$ is the only source of $h$. Consequently, if a hyperedge has more than one source, none of its sources is an activity node.

$$\forall\, h \in HyperEdges\, \forall\, a \in AN \bullet a \in source(h) \;\Rightarrow\; source(h) = \{a\}$$

   We motivated this constraint already on page 28.

2. Every hyperedge that leaves an activity node is labelled with a corresponding activity termination event.

$$\begin{aligned}&\forall\, h \in HyperEdges\, \forall\, a \in AN \bullet a \in source(h) \Rightarrow \\ &\quad (\; event(h) \in TerminationEvents \;\wedge\; term(event(e)) = a \;)\end{aligned}$$

3. The initial node is only part of the sources of a hyperedge. Moreover, if it is source of a hyperedge, it is the only source of that hyperedge.

$$\begin{aligned}&\forall\, h \in HyperEdges \bullet initial \notin target(h) \\ &\qquad\qquad \wedge\; (\; initial \in source(h) \;\Rightarrow\; source(h) = \{initial\}\;)\end{aligned}$$

A final node is only part of the targets of a hyperedge. Moreover, if it is target of a hyperedge, no non-final node is target of that hyperedge.

$$\forall\, h \in \mathit{HyperEdges} \bullet \mathit{FN} \cap \mathit{source}(h) = \varnothing \\ \wedge\ (\ \mathit{FN} \cap \mathit{target}(h) \neq \varnothing\ \Rightarrow\ \mathit{target}(h) \subseteq \mathit{FN}\ )$$

4. Hyperedges leaving the initial node have no trigger events, and the disjunction of their guard expressions is a tautology.

$$\forall\, h \in \mathit{HyperEdges} \bullet \mathit{source}(h) = \{\mathit{initial}\}\ \Rightarrow \\ \mathit{event}(h) = \bot \\ \wedge \bigvee \{\ \mathit{guard}(h) \mid h \in \mathit{HyperEdges} \wedge \mathit{source}(h) = \{\mathit{initial}\}\ \}$$

This constraint ensures that the initial state can be left immediately (see Constraint 5 on page 29).

## 3.3 From activity diagram to activity hypergraph

The mapping from activity diagrams to activity hypergraphs consists of three steps: (1) rewriting of some syntactic expressions, (2) eliminating hierarchy (compound activity nodes), and (3) computing hyperedges. Steps 1 and 3 have been implemented in TCM [60].

**Rewriting of some syntactic expressions.** First, else is replaced by the expression it abbreviates. Second, we replace every after($t$) label of hyperedge $h$ with after($t$,$h$). This replacement makes the after constraint unique for two hyperedges, ensuring that if two hyperedges have the same after label, there is no confusion as to for which hyperedge the timeout is meant. Third, edges leaving some activity node $a$ and having no trigger event are labelled with $\mathit{term}^{-1}(a)$.

**Eliminating hierarchy.** An activity diagram can contain compound activity nodes. Every compound activity node is decomposed into another activity diagram that specifies the behaviour of the compound activity node. We require that the transitive closure of the decomposition relation be acyclic. We eliminate a compound activity node n by replacing n by its corresponding activity diagram. The initial and final nodes of the corresponding activity diagram are OR nodes, which are eliminated as follows. Every compound edge that enters n is glued together with every compound edge that leaves the initial node. Every compound edge that leaves n is glued together with every compound edge that enters one of the final nodes.

**Computing hyperedges.** We first define a compound edge. A *compound edge* is a set of edges that are linked by AND (fork/join) and OR (decision/merge) nodes, satisfying the following constraints.

- If an edge in a compound edge *ce* enters or leaves an AND node, then every edge that leaves or enters the AND node is part of *ce*.
- If an edge in a compound edge *ce* enters (leaves) an OR node, then there is one edge in *ce* that leaves (enters) the OR node.

Not every compound edge that satisfies above rules is well-defined. Compound edges are not well-defined if they contain cycles. Cycles are bad for the following reasons.

- Sometimes a compound edge with a cycle is unreachable: the compound edge does not start with an non-pseudo node. For example, in Figure 3.7(a) there are two compound edges, namely {e1,e2,e4} and {e2,e3}. The compound edge {e2,e3} is not well-defined since it is unreachable.
- Sometimes a compound edge with a cycle is reachable but cannot be executed. For example, in Figure 3.7(b) there is one compound edge, namely {e5,e6,e7,e8}. This compound edge is reachable but not executable: Edge e6 can only be taken after e7 has been taken, whereas edge e7 can only be taken after e6 has been taken (remember that an edge denotes sequence). So e6 and e7 cannot be taken, and therefore the compound edge cannot be taken.

Above discussion can be summarised by quoting the definition of a (well-defined) compound edge in the UML standard [150]: "a [well-defined] compound edge is an acyclical, unbroken chain of edges".

As explained on page 28, the effect of a compound edge can be specified with a hyperedge. The hyperedge specifies which nodes are left and which are entered if the compound edge is taken. The hyperedge neither refers to the edges contained in the compound edge, nor to the pseudo nodes that link them.


**Figure 3.7** *Two ill-defined compound edges*

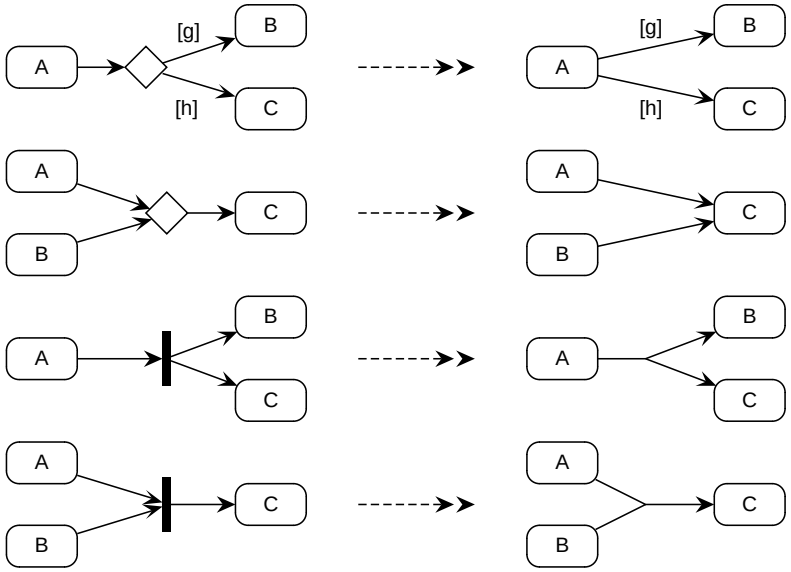



**Figure 3.8** *Example computations of hyperedges*

We compute hyperedges by processing the pseudo nodes of the activity diagram one by one. Figure 3.8 shows the most simple mappings. An OR node with $n$ incoming or $n$ outgoing edges maps into $n$ new hyperedges. An AND node maps into one new hyperedge.

The only difficulty that arises is when an AND node is connected with an OR node. Then the order of processing is significant: processing an AND node before an OR node gives a different result than processing an OR node before an AND node. For example, the pseudo nodes in the activity diagram on the lefthand side in Figure 3.9 can be processed in two ways. Processing the OR node before the AND nodes gives the hypergraph in the top right, whereas processing the AND node before the OR node gives the hypergraph in the bottom right. The intended mapping is the one on the bottom right: either node W3 or W4 is entered but not both. So, AND nodes should be processed before OR nodes.

By Constraint 6 on activity diagrams, listed on page 30, we have that computation of a compound edge always produces a set of source nodes and a set of target nodes, rather than a bag.

Like an edge, a hyperedge has an ECA label. The ECA label of the hyperedge can be derived from the labels of the edges that are glued together. By Constraint 4 on activity diagrams, listed on page 29, at most one edge in a compound edge has a trigger event. This trigger event is the trigger event for the hyperedge. The guard condition of the hyperedge is the conjunction of the guard conditions of the edges that are part of the corresponding compound edge. The events generated

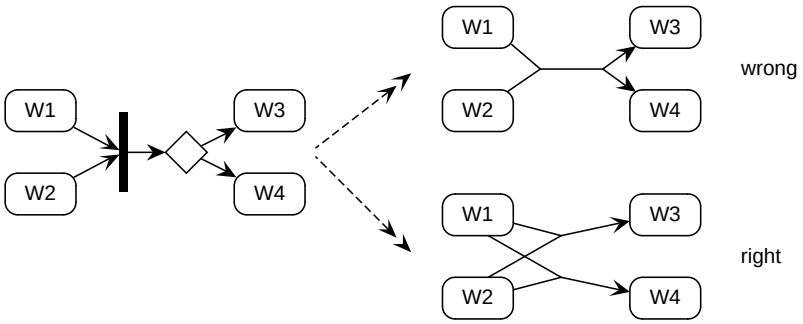



**Figure 3.9** *Two possible computations of hyperedges*

by the hyperedge are the events generated by the edges of the compound edge.

## 3.4 Specifying activities

Above, we focused on possible semantics for edges in an activity diagram. We now focus on the semantics of nodes, more precisely the semantics of activity nodes.

We explained in Section 2.5 that the actual execution of activities falls outside the scope of the system being modelled, a WFS, since activities are done by actors. Moreover, we do not model actors; we simply assume that an activity, once it becomes enabled, will terminate sometime in the future. The only relevant aspect of the execution of an activity is that when it terminates, some of the control data may have been changed. This affects the routing of the case.

We therefore specify the effect of an activity declaratively using pre and post-conditions. A *precondition* is a logical expression that is used by the WFS to decide when the activity may be started. A *postcondition* is a logical expression that is used by the WFS to decide when the activity has been completed. Both pre and postconditions refer to control data.

Unfortunately, pre and postconditions do not have a standard semantics as pointed out by Bussler [29]. That is, they can be interpreted differently by different WFSs. For example, if in the workflow of Figure 3.10(a) activity $A$ completes and $x$ is 5, then some WFSs will stop the whole workflow, i.e, the workflow of Figure 3.10(a) behaves similar to the workflow of Figure 3.10(b). Other WFSs, however, will wait until $x$ gets a value greater than 10 (Figure 3.10(c)). Yet other WFSs will skip $B$ and proceed with $C$ (Figure 3.10(d)). Similarly, a postcondition has different interpretations too. If a postcondition fails to hold, the WFS can decide that the activity has to be redone, or that the whole workflow is stopped.

Since pre and postconditions have such an unclear semantics, we decide not to model them implicitly as in Figure 3.10(a) but explicitly using guard conditions as in Figures 3.10(b)-(c). Then the semantics of the pre and postcondition follows

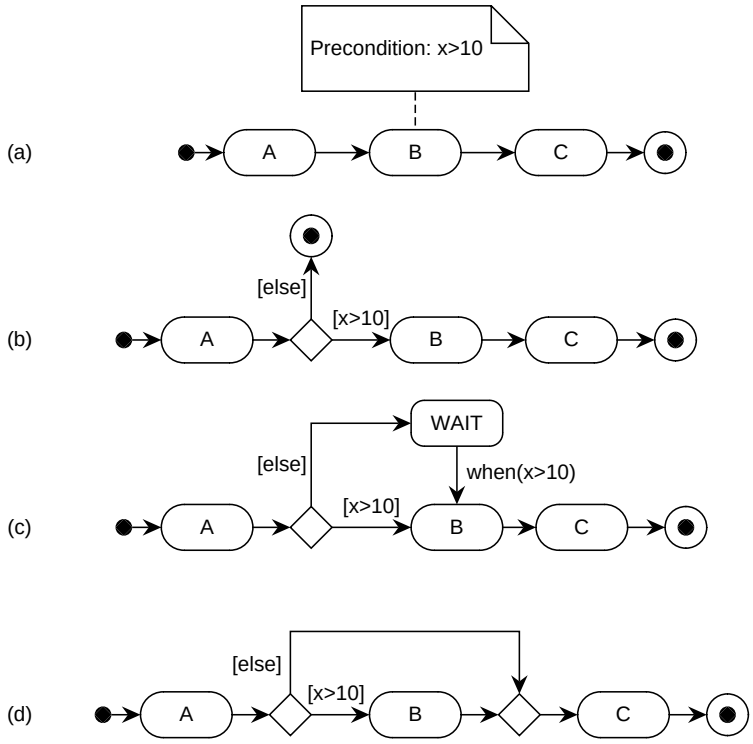



**Figure 3.10** *Workflow with precondition and three possible interpretations*

immediately from the workflow specification.

In Section 2.2 we explained that a WFS can prevent interfering activities from being active simultaneously, thus enforcing isolation between activities. In order to do so, the WFS needs to know what activities interfere with each other. The interference information can be defined either explicitly by the workflow designer, or the WFS can derive it automatically from the specification of activities. In this thesis we only consider the last option.

In database theory, a useful criterion for detecting interference between transactions is serialisability [56, 152]. Roughly speaking, if two transactions both simultaneously access (observe/update) the same variable and in addition one updates the variable, then the two transactions are not serialisable (commutative) and thus they interfere with each other. We take a similar approach and specify for each activity the variables it observes (reads) and the variables it updates. From this information, the WFS can derive whether or not two activities interfere with each other.

Formally, in every activity $a \in \mathit{Activities}$ that is controlled by an activity diagram, some local variables may be *observed* or *updated*. We denote the observed

variables by $Obs(a) \subseteq LVar$, and the updated variables by $Upd(a) \subseteq LVar$. We require these two sets to be disjoint for each activity. Note however that it is possible to have $Obs(a) = Upd(b)$, if $a \neq b$.

Two activities are in *conflict* or *interfere*, if one of them observes or updates a local variable that the other one updates. (This definition is similar to the definition of conflict equivalence in database theory [56, 152].)

$$\begin{aligned} A \lightning B \quad \Leftrightarrow \quad & (Obs(A) \cup Upd(A)) \cap Upd(B) \neq \varnothing \\ & \vee\, (Obs(B) \cup Upd(B)) \cap Upd(A) \neq \varnothing \end{aligned}$$

Note that this particularly implies that we only allow autoconcurrency (two instances of the same activity that are active at the same time) if the autoconcurrent activity does not update any variables.

The environment of the organisation may also change or update some variables, denoted $Obs(env)$ and $Upd(env)$ where $env$ is a special symbol, $env \notin Activities$, representing the environment. The WFS cannot prevent the environment from interfering with an activity that accesses the same variable; such a prevention is the job of the database system.

For our running example (see Figure 1.1 on page 3), we specify that activity *Check stock* updates variable insufficient stock, that *Check customer* updates customer ok, and that *Handle payment* updates payment ok. The environment does not update any of these variables.

# Chapter 4

# Design choices in semantics of activity diagrams

We use UML activity diagrams to specify workflows. As explained in Chapter 2, a workflow specification prescribes how a workflow system should behave. We therefore motivate and define the execution semantics of activity diagrams in terms of WFSs (see Section 2.5). In this chapter we discuss the design choices we make in our semantics for activity diagrams. We give an informal introduction to two semantics that both satisfy the same design choices, but are otherwise completely different. One is a high-level semantics which assumes that a WFS is infinitely fast and reacts immediately to events, whereas the other one is a low-level semantics which does not make this assumption. In the next chapter, we formally define these two semantics. A comparison of both semantics with Petri nets can be found in Chapter 8. A comparison with other related work can be found in Chapter 9.

Section 4.1 explains what (a state of) the mathematical structure looks like. The design choices that we made for the semantics are motivated by the domain of workflow systems as explained in Chapter 2. Section 4.2 looks at how an activity diagram changes states. We study two existing execution semantics, the Petri net token-game semantics and the statechart semantics. Since the statechart semantics fits our purposes best, we take that semantics as a starting point. Section 4.3 lists several issues in the semantics that have to be resolved, and for each issue we show the choice that we make in order to resolve the issue. Section 4.4 informally introduces two semantics that both satisfy these choices but that are otherwise completely different from each other.

## 4.1 Mathematical structure

From the discussion in Chapter 2, we conclude that workflow systems have the following characteristics.

- **A WFS is reactive.** A *reactive system* runs in parallel with its environment and responds (reacts) to input events by creating certain desirable effects in the environment [91, 157]. For a WFS, typical input events are activity termination events, in Figure 1.1 for example that the Receive order activity node terminates. Other events are also possible (cf. Figure 2.4 on page 19). And characteristic desirable effects for a WFS are the enabling of new activity instances.

- **A WFS has coordination functionality.** A WFS does not execute the activities themselves, but it merely coordinates the execution of the activities by the actors (people or machines)[1]. For example, in Figure 1.1 (page 3) the WFS does not check a customer itself, but merely tells the relevant actors that one of them can start checking the customer. Case attributes are only changed in activities by actors, not by the WFS.

The semantics of activity diagrams must be able to represent these aspects accurately.

As explained in Chapter 1, every formal semantics is a mapping of a syntactic structure into a mathematical structure. The structure we use in this thesis is that of a run (to be precise, a set of runs). A *run* (or a trace or scenario) is a sequence of states connected by state changes. A state is a condition of the world, for example: activity *Receive order* is being executed. A state change is a change in condition of the world, for example: activity *Receive order* terminates. State changes are instantaneous. Non-instantaneous state changes can always be split into an instantaneous begin and end state change. So time only elapses in a state.

Runs are frequently used to give a semantics to reactive systems [90, 122]. They are also used as a semantic domain in model checking [42]. So, runs are suitable to represent the behaviour of workflows and in addition can be used for model checking functional requirements. Moreover, the UML action semantics [12] employs a similar notion as semantic domain.

Since a run is a possible behaviour of a WFS, states of the run are states of the WFS. Keeping in mind the characteristics of a WFS, we define that a state of a run consists of the following components:

- the state of the case (i.e, which nodes in the activity diagram are active, possibly multiple times),

- the queue of input events of the WFS,

- the case attributes and their values, and

- the scheduled timeouts and the value of the internal clock.

[1] Sometimes, there can be confusion if the WFS runs on the same computer as the application software that performs an activity; but even then the application software differs from the WFS.

A queue of input events is needed because of the first characteristic: a WFS is a reactive system. In a reactive system state changes are caused by input events. This means that the WFS must have some interface with the environment to observe the input events. We therefore use an input queue in which events are kept. The case attributes are needed to evaluate the guard conditions on the hyperedges, i.e., they are only used for routing the case. Timeouts are raised by the WFS itself, on the basis of an internal clock.

The second WFS characteristic, coordination, has several implications. First, activities are done by the environment in states of the WFS, i.e, during an activity state the WFS waits for an activity to complete (see Section 2.5). Activities take non-zero time to execute. Second, an activity is specified declaratively, in particular its postcondition. This was explained already in Section 2.5. Third, in a reaction case attributes are not changed. Instead, changing (updating) of case attributes is done by the environment of the WFS. Also, the WFS does not maintain the case attributes, this is done by its environment (usually by a database system).

## 4.2 Petri net token-game semantics versus statechart semantics

We now have to decide how an activity diagram changes state. In other words, we have to define the execution semantics of an activity diagram. Since in an activity diagram a change of state is modelled by an edge (a hyperedge in an activity hypergraph), we look at the semantics of a (hyper)edge in this section. We keep in mind that an activity diagram prescribes how a WFS behaves.

So the question to answer is: When is a hyperedge taken? There are several existing semantics for graphical notations that answer this question differently. The most relevant ones for activity diagrams are Petri nets, since they look like activity diagrams, and statecharts, because they too look like activity diagrams and in addition the current UML semantics of activity diagrams is defined in terms of statecharts, even though this will be changed in the future. Various semantics for Petri nets have been defined, but we focus on the standard token-game semantics used for low-level Petri nets. Timed and stochastic Petri nets have a more complicated semantics that is too complex for our purposes. Statecharts are notorious for their many semantics; in 1994, Von der Beeck [18] listed already over twenty different semantics, without considering object-oriented statechart variants. The introduction of UML has increased the number of semantics of statecharts considerably. All the different statechart semantics agree, however, on the semantics of hyperedges that we sketch below. Introductions to Petri nets and their semantics are by Peterson [136], Reisig [141] and Murata [129]. Introductions to statecharts and their semantics are by Harel [88], Harel and Naamad [90] and Wieringa [157].

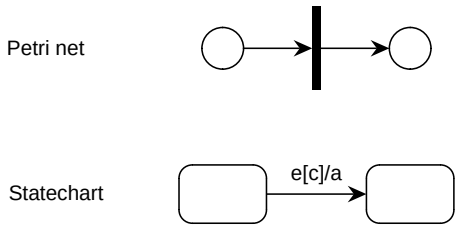



**Figure 4.1** *Differences between Petri net token-game and statechart semantics*

There are two major differences between the Petri net token-game semantics and statechart semantics of hyperedges[2]. First, in a Petri net, a hyperedge (transition) is enabled once its input places are in the current marking. In a Petri net, a label on a hyperedge does not influence the enabledness of that hyperedge; the label is not interpreted. So, in Figure 4.1 the hyperedge in the Petri net is enabled iff its input place is filled with a token. In a statechart, a hyperedge (transition) is enabled once the system is in the input states of the hyperedge *and* both its trigger event occurs and its guard condition is true. The trigger event and guard condition are specified in the label of the hyperedge; the label is interpreted. So, in Figure 4.1 the hyperedge in the statechart is enabled iff its input state is active *and* event `e` occurs and the condition `c` is true.

Second, in a Petri net an enabled hyperedge *may* be taken, but it does not have to be taken. In a statechart, an enabled hyperedge *must* be taken, since the event occurrence must be responded to. Consequently, in Figure 4.1 if the hyperedge in the Petri net is enabled, we still do not know whether or not the hyperedge in the Petri net is taken. Whereas if the hyperedge in the statechart is enabled, we do know that it is taken.

These differences in semantics of hyperedges are due to a difference in intended use of the notation. Petri nets, under the token-game semantics, model the resource usage of closed, active systems. Resources are represented by tokens. The presence of a token in a place gives information about the possibility to perform certain transitions, i.e., to use the resources in a certain way. A closed, active system decides itself when it should do something, i.e., take a transition; there is no environment to provide stimuli to the closed, active system. By contrast, statecharts model open, reactive systems. The environment of an open, reactive system provides the system with input events. The system must react to these events. The current state of a statechart gives information about what will happen if certain events occur. We conclude that the statechart semantics of hyperedges matches the domain of reactive systems best.

Based on this analysis we draw the following conclusion. Since we use an activity

[2]In both Petri nets and statecharts a hyperedge is called a transition, but we prefer the term 'hyperedge' since 'transition' denotes a semantic concept, as explained on page 23.

diagram as workflow specification, and a workflow specification prescribes how a WFS behaves, edges in an activity diagram (hyperedges in an activity hypergraph) are transitions by the WFS. A WFS is a reactive system. The statechart semantics represents reactive behaviour of a system more accurately than the Petri net token-game semantics. We therefore take the statechart semantics as starting point of our semantics.

After having defined our formal semantics in Chapter 5, we will study in Chapter 8 whether and how Petri nets can be used to simulate our reactive semantics of activity diagrams.

## 4.3 Issues in reactive semantics

In the statechart semantics, or rather any reactive semantics, the system takes a hyperedge, i.e., makes a transition to another state, in response to some input events. There are several issues that have to be decided upon in defining a semantics for reactive systems. We label each choice that we make with a checked symbol ✓.

First of all, can more than one event occur at the same time? We make two observations. First, although the chance of two events occurring simultaneously is rather small, it is not equal to zero. Second, the reactive system (WFS) will respond to events by inspecting the contents of the queue. If no two events can occur simultaneously, the rate at which events occur in the environment must be slower that the rate at which the WFS reads input events (sampling rate). We do not want to impose such a restriction upon the environment and therefore do not make such an assumption.

✓ Events can occur simultaneously.

From this choice, it follows that two event occurrences are either simultaneous, or some time elapses between them.

Second, can an event trigger one or more than one hyperedge at the same time? We allow an event in general to be either broadcast, triggering arbitrarily many hyperedges, or point-to-point, triggering at most one hyperedge (see Section 2.4). As explained in Section 3.2, however, some events have a fixed sendtype, for example activity termination events. It does not make sense to broadcast an activity termination event of activity $A$, leaving arbitrarily many activity nodes that enable $A$, since only one instance of $A$ has terminated and therefore only one activity node should be left. If an event can both be broadcast and point-to-point, we use the broadcast interpretation as default (see Table 3.1 on page 33).

✓ Events can be either broadcast or point-to-point.

From these two choices, it follows that more than one hyperedge can be taken at the same time. Two hyperedges $h$ and $h'$ can be taken at the same time either

because their trigger events occur at the same time, or they have the same trigger event which is broadcast. The collection of hyperedges that is taken at the same time is called a *step* in statechart terminology [90, 150].

Third, during execution of the system, the event queue is filled with events. The system reads the events from the event queue and reacts to them. There should however be some removal policy. If an event is not removed after it is processed, it would continue to have an effect, which is undesirable. Since the result of the event occurrences is the taking of some hyperedges (a step), the events should be removed after these hyperedges have been taken.

✓ An event is removed after the step in which it is processed has been taken.

Fourth, there are some obvious constraints that steps must satisfy. For example, every hyperedge in the step must be triggered by one of the input events, and no two hyperedges can leave the same node instance at the same time. In the next chapter, we will formalise these constraints. One not so obvious, but very important constraint is that a step is *maximal*. This constraint is satisfied by every statechart semantics. Not imposing this constraint would imply that some hyperedges that are enabled would not have to be part of the step, so would not have to be taken. Since an event is removed from the input after the subsequent step has been taken, this would mean that some input events would not cause all their effects, although, according to the workflow model, they *should* have these effect (enabled hyperedges should be taken). In other words, then the WFS would not react fully to these input events. That is why we require that a step be maximal.

✓ Steps are maximal.

Fifth, events can be processed either immediately upon arrival or at fixed points in time, that is, at ticks of the clock, for example once every hour. The former is called an event-driven model whereas the latter is called a time-driven model. In STATEMATE both response regimes are supported: there the event-driven model is called an asynchronous time model, since it is asynchronous with respect to the system clock, whereas the time-driven is called a synchronous time model, since it is synchronous with ticks of the system clock. In previous work [62, 63, 64], we used the terms 'clock-asynchronous' and 'clock-synchronous' semantics. The terminology we use here is borrowed from control theory [17].

For workflow modelling, the event-driven model is more appropriate. The time-driven model is useful for embedded systems.

✓ Events are processed when they arrive (event-driven model).

Sixth, it must be decided what should happen if an event occurs that has no effect in the current state of the system. In statecharts, as in control theory in general, the assumption is made that input events cannot be blocked. Instead, the system should be able to respond to every possible event occurrence. If no effect is

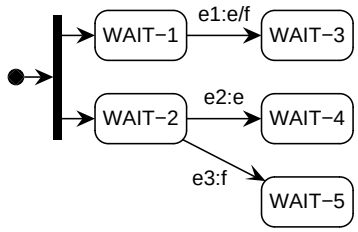



**Figure 4.2** *Event generation*

specified, nothing happens. We adopt this assumption too. UML statecharts also adopt this assumption, but in addition allow for the deferring of an event that has no effect in the current state. As soon as the system reaches a state in which the event is not deferred, the event is responded to. We come back to the deferring of events in Chapter 7.

The alternative assumption is to block an unexpected event reception until the system is ready to respond to it. This latter choice is made in process algebra [97, 127], but it puts a constraint upon the environment. We do not want to put such a constraint upon the environment.

✓ Event receptions cannot be blocked.

Finally, we discuss what happens when events are generated. There are two different ways of interpreting event generation. The first one is to let the generated events have an effect in the current step (chosen in the fixpoint statechart semantics of Pnueli and Shalev [140]), the second one is to let the generated events have an effect in the next step (chosen in the STATEMATE statechart semantics [90]). To illustrate the difference between these two options, suppose in Figure 4.2 the current configuration is [WAIT-1,WAIT-2] and event e occurs. If edge e1 is taken, then according to the fixpoint semantics event f is immediately available and consequently edge e3 can be taken simultaneously with edge e1. Whereas in the STATEMATE semantics, event f can only be sensed *after* the step in which it is generated is taken, so *after* edge e1 is taken. Consequently, if the current configuration is [WAIT-1,WAIT-2], and event e occurs, in the fixpoint semantics either step [e1,e2] or [e1,e3] is taken, but in the STATEMATE semantics, step [e1,e2] is taken. Step [e1,e3] is counterintuitive here, since it seems that event $e$ is ignored in node WAIT-2. So there are circumstances in which the fixpoint semantics computes a counterintuitive step (this was first pointed out by Leveson et al. [117] using a similar example, but they mistakenly attribute the fixpoint semantics to STATEMATE). That is why in practice the STATEMATE approach is taken, even in the UML. We adopt the STATEMATE interpretation for event generation as well, since it is also adopted by UML.

✓ Generated events are sensed in the next step.

| |
|---|
| ✓ Events can occur simultaneously. |
| ✓ Events can be either broadcast or point-to-point. |
| ✓ An event is removed after the step in which it is processed has been taken. |
| ✓ Steps are maximal. |
| ✓ Events are processed when they arrive (event-driven model). |
| ✓ Event receptions cannot be blocked. |
| ✓ Generated events are sensed in the next step. |

**Table 4.1** *General choices made in our reactive activity diagram semantics*

As an aside, note in this interpretation too, there are anomalies. One may for example get infinite loops of event generation that trigger each other, as pointed out by Leveson et al. [117]; see Figure 4.3 and the corresponding discussion on page 50.

Table 4.1 summarises the general choices that we made. The first six assumptions are made in every statechart semantics, including STATEMATE and UML. (STATEMATE supports both an event-driven (asynchronous time) and time-driven (synchronous time) model [90].) Some of the choices are also made in other formal methods, like process algebra, but none of them makes the same choices. The last choice is made by both STATEMATE [90] and UML [150]. The fixpoint semantics of Pnueli and Shalev [140], in which the alternative property is adopted, is widely studied by theoretical computer scientists, but we do not know of any practical application of this semantics. There is no tool that implements it.

Although we have made these choices, we have left open a lot of other issues. Many different semantics can be defined that satisfy the properties listed above. In this thesis we will define two semantics, which are completely different from each other, even though both satisfy all the properties listed in Table 4.1. We introduce these two semantics in the next section.

## 4.4 Two reactive semantics

Having resolved some general issues in the previous section, we still have a lot of other issues to decide upon. The main issues we did not deal with is whether the processing of events, i.e. reacting to them, takes time or not. This question is answered differently in various statechart semantics. In the classical statechart semantics, such as the fixpoint semantics of Pnueli and Shalev [140] and the STATEMATE semantics of Harel and Naamad [90], the assumption is made that event processing ("taking a hyperedge") does not take time.

This assumption is an example of what is called the perfect technology assumption in software engineering. The *perfect technology assumption*, introduced by McMenamin and Palmer [124] and subsequently adopted in Yourdon's structured analysis [162], states that the system under development has infinite processing power: It reacts infinitely fast and has infinitely many resources. The perfect

technology assumption is adopted in order to focus on specifying the interaction of a system with its environment, without being bothered by limitations of the implementation platform on top of which the system is implemented. In other words, the perfect technology assumption is adopted to focus on specifying what the requirements of a system are, so what effects the system should achieve in its environment, rather than on specifying how well the system achieves these effects, for example how fast a response is. Models satisfying perfect technology are called essential-level models in structured analysis [124, 162], but we prefer the term *requirements-level* models.

In modern statechart semantics, especially the object-oriented variants, the perfect technology assumption is dropped. It is dropped because it is considered to be unrealistic. The most important statechart semantics in this group are the (informal) UML statechart semantics, and its predecessor, the ROOM statechart semantics [146]. Models that do not make the perfect technology assumption we call *implementation-level* models.

As an aside, note that although the perfect technology assumption is usually made in structured analysis statecharts and not in object-oriented statecharts, this does not mean that an implementation-level semantics cannot be used in structured analysis, or a requirements-level semantics in object-oriented statecharts. For example, in earlier work [62] we defined a requirements-level semantics for UML statecharts.

In this thesis, we will define a requirements-level and an implementation-level semantics for activity diagrams, both in terms of WFSs.

**Requirements-level semantics.** The requirements-level semantics we define is based upon the STATEMATE semantics of statecharts [90]. The perfect technology assumption abstracts from internal implementation details of the WFS. In the requirements-level semantics the WFS is therefore considered as a black box.

If the processing of events does not take time (by perfect technology), and events are processed upon arrival (event-driven model), then the logical choice is to let events be processed immediately. So, if an event occurs, the system (WFS) starts immediately processing the event and does not take time to process the events. In other words, an event occurs simultaneously with the subsequent reaction of the system. This assumption is called the *perfect synchrony hypothesis*. It was introduced by Berry and Gonthier [19] for the synchronous programming language ESTEREL and it was also adopted for other synchronous languages [86], including the asynchronous (event-driven) semantics of STATEMATE [90].

One may wonder how the perfect synchrony hypothesis is implemented in practice. For sure, no system will react immediately to an event. In synchronous languages, the hypothesis translates into the constraint that the system is fast enough to be ready before the next event occurs [19]. So the system reaction may take time, but it is always in time, before the next event arrives. Note that this actually puts a constraint upon the environment of the system, rather than the

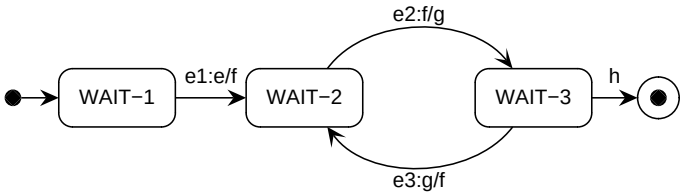



**Figure 4.3** *Diverging activity diagram*

system itself.

We will pursue a different justification of the requirements-level semantics, by relating it (in Chapters 6 and 10) to another semantics, the implementation-level semantics discussed below, that does not have the perfect synchrony hypothesis.

As explained in the previous section, the system reacts to events by taking a step. If the system enters a new state by taking a step, some internal events can be generated and some hyperedges can be enabled in this new state, because their guard is true and either their trigger events were generated in the previous step or they do not have a trigger event. In that case the new state is *unstable*. By the perfect synchrony hypothesis, in such an unstable state immediately a new step is taken and another state is entered. If this other state is also unstable, again a step is taken, and a new state is entered. This sequence of taking a step and entering a new state is repeated until finally a state is reached in which there are no events in the queue and there are no enabled hyperedges. Such a state is *stable*. The sequence of steps that is taken is called a *superstep* [90].

The superstep may be nonterminating, because some hyperedges may enable each other. If a superstep does not terminate, we say the superstep *diverges*. A simple example of a diverging superstep is shown in Figure 4.3. If the system is in state WAIT-1 and event e occurs, then edge e1 is taken and event f is generated. The resulting state is unstable because edge e2 is enabled. Then, if edge e2 is taken, event g is generated and edge e3 becomes enabled. And taking edge e3 generates event f, so edge e2 becomes enabled again. The superstep then diverges: it never terminates. This drawback of the STATEMATE semantics was first pointed out by Leveson et al. [117].

One final issue that we must decide upon is whether a single event is processed at a time or all events are processed in parallel. Under the perfect synchrony hypothesis, it makes more sense to process all events at the same time. We therefore assume that all events are processed in parallel. This assumption is also made in STATEMATE [90].

Figure 4.4 shows an example run of the activity diagram in Figure 1.1 (activity hypergraph in Figure 3.6) under this semantics. In each state, the set of running activities is shown, as well as the variables that are updated by one of these running activities. The end value of a case attribute that is updated in an activity $a$ can

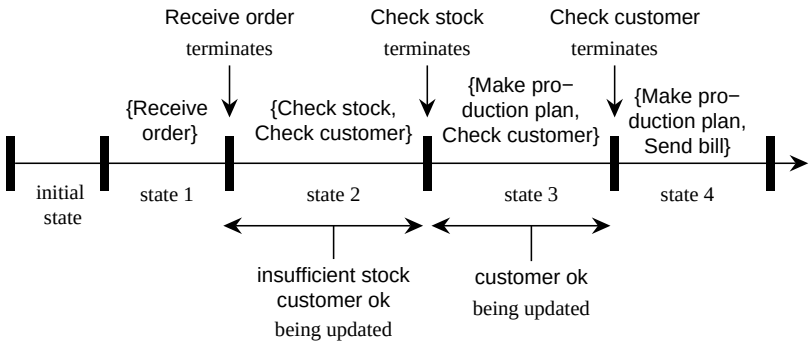



**Figure 4.4** *Example run in requirements-level semantics*

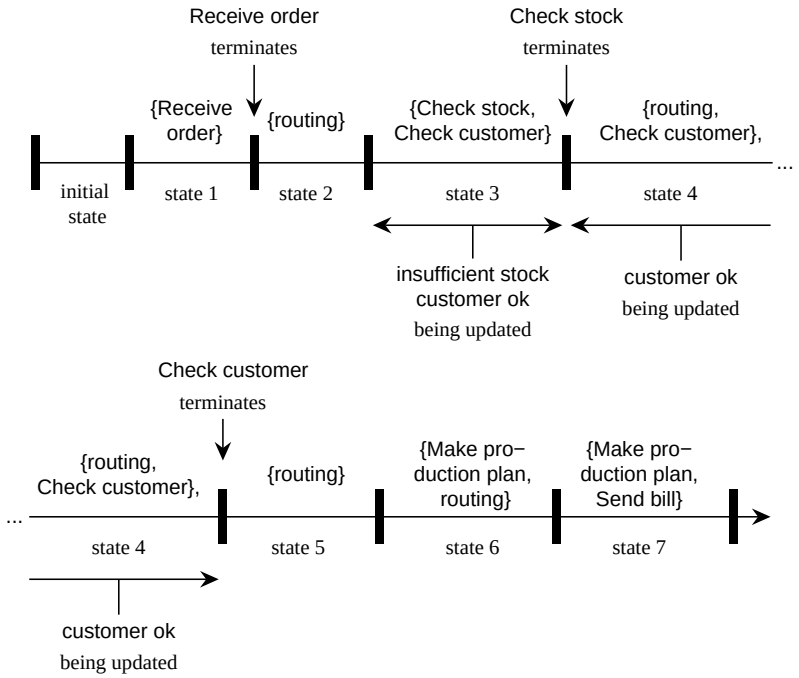



**Figure 4.5** *Example run in implementation-level semantics*

be derived from the activity that is started when $a$ terminates.

**Implementation-level semantics.** In the second semantics, the perfect technology assumption is dropped. Hence, the perfect synchrony hypothesis is dropped as well. So a reaction of the WFS takes time, and input events are not immediately reacted to. In particular, while the WFS is busy reacting, the next events can already occur. The semantics is based upon the OMG semantics of UML statecharts [150]. In the implementation-level semantics, the WFS is considered as a white box, consisting of the components shown in Figure 2.3 on page 17. The Router component is responsible for producing the desired reaction: routing the case to the new state, enabling some new activity instances to start. Figure 2.3 resembles the architecture of workflow systems [31, 81, 118, 159], and also the informal UML definition of state machines, underlying UML statecharts [150].

Since the perfect technology assumption is not made, the Router component has limited capacity. So it takes time to process an event, whereas in the requirements-level semantics the WFS is infinitely fast. Moreover, we will assume that a Router processes one event at a time, rather than arbitrarily many as in the requirements-level semantics. Harel and Gery [89] call this *single-event processing*. The informal OMG semantics of UML statecharts [150] also uses single-event processing.

When the Router starts routing, it picks some input event from the queue. It routes the case by updating the state of the case, enabling some new activities to start, scheduling some new timeouts, and removing some scheduled timeouts, because they have become irrelevant in the new state of the case. Afterwards, the Router starts processing the next event from the queue. Since the next input events might have arrived while the Router was busy routing the case, the content of the queue might have changed during routing. Note that this is impossible in the requirements-level semantics, since there routing is instantaneous.

Figure 4.5 shows an example run of the activity diagram in Figure 1.1 (activity hypergraph in Figure 3.6) under this semantics. In Figure 4.5, the term routing denotes that the Router is busy routing the case. The imperfect WFS in Figure 4.5 has the same input events as the perfect WFS of Figure 4.4. Note that the run in the implementation-level semantics (Figure 4.5) has twice as many states as the comparable run in the requirements-level semantics (Figure 4.4). In general, implementation-level runs have many more states than requirements-level runs.

Figure 4.6 shows in more detail how the state components of the WFS change state during a run in the implementation-level semantics. Note that also in this semantics state changes are instantaneous. The scheduled timeouts are not shown in the figure. A state change in a reactive system is either caused by (1) the occurrence of some input events, (2) picking of an event from the queue by the Router, (3) the reading of the current values of the case attributes by the Router, or (4) changing of the state of a case, leaving or entering states, by the Router. A state change can also consist of a combination of (1) with one of (2), (3) or

| **requirements-level semantics** | **implementation-level semantics** |
|---|---|
| - perfect technology | - imperfect technology |
| - parallel-event processing | - single-event processing |
| - event is immediately responded to | - event is responded to at some later time |

**Table 4.2** *Differences between requirements and implementation-level semantics*

(4). In the requirements-level semantics, these four state changes always happen simultaneously.

The first and second type of state change only affect the state of the queue: the events are added to the queue (first) and removed (second state change). The third state change only affects the case attributes: the current value of each case attributes is read in order to evaluate the guard conditions on hyperedges. Note that case attributes are maintained by the database, not by the WFS. So, the WFS merely needs a copy of them in order to evaluate guard conditions and route the case to the desired next state. The fourth type of state change only affect the state of the case: some nodes become active, others become inactive.

**Summary.** Table 4.2 sums up the differences between the two semantics. Note that although in the implementation-level semantics one event at a time is processed, still more than one event can occur at the same time. But in the requirements-level semantics events occurring in parallel are processed in parallel, whereas in the implementation-level semantics these events are processed one by one (single-event processing).

Evaluating the two semantics, the requirements-level semantics is easy to analyse, but not completely accurate, because no real WFS will satisfy the perfect synchrony hypothesis. The implementation-level semantics is more accurate, but also more difficult to analyse than the requirements-level semantics. Runs in the

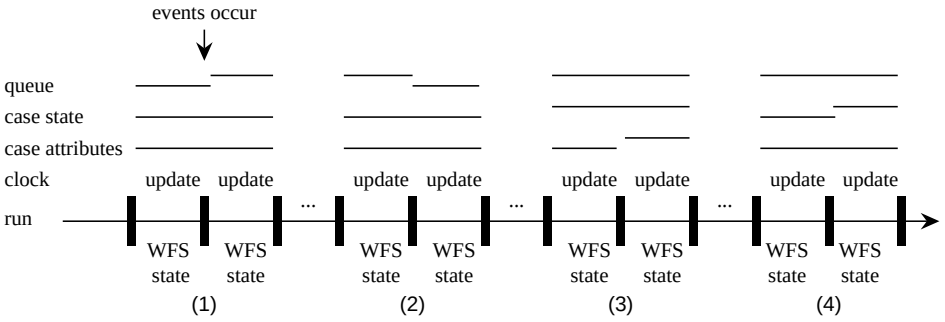



**Figure 4.6** *Structure of run in implementation-level semantics*

implementation-level semantics are harder to relate to the original activity diagram than runs in the requirements-level semantics, because in the implementation-level semantics there is a delay between the occurrence of an event and the subsequent reaction of the WFS to that event occurrence, whereas in the requirements-level semantics, there is no such delay. For example, the run in Figure 4.5 is harder to match with the activity diagram of Figure 1.1 than the run in Figure 4.4. Moreover, the implementation-level semantics is more difficult to analyse for a verification tool, because there will be more states due to the delay in response to event occurrences.

In this thesis, we focus on analysis of functional properties of workflows, for example the absence of deadlock. As we will show in Chapter 10, for such properties, it does not matter whether the requirements-level semantics or implementation-level semantics is chosen: if a workflow specification contains for example a deadlock in one semantics, it also will have the deadlock in the other semantics and vice versa. So we can use the requirements-level semantics for analysis of such properties with the assurance that the analysis result will also hold when the workflow specification is executed under the implementation-level semantics.

Note that we do not specify how the environment behaves. In general, the exact behaviour of the environment is unknown. In our model checking semantics, we have simply assumed that the environment can behave in every possible way, i.e., chaotically (see Chapter 5 for more details). For analysis purposes, we assume that the environment behaves in a fair way; see Chapter 10.

# Chapter 5

# Two formal semantics of activity diagrams

In the previous chapter, we informally introduced a requirements-level semantics and an implementation-level semantics for activity diagrams. In this chapter we define these two semantics formally. In the next chapter we will study similarities and differences between the two semantics.

Section 5.1 defines the semantic structure that we use, a Clocked Transition System. Section 5.2 defines the semantics of steps. As explained in the previous chapter, a step is the basic unit of reaction. The step semantics is used in both the requirements-level semantics and implementation-level semantics. Sections 5.3 and 5.4 define the requirements-level semantics and the implementation-level semantics. Each semantics maps an activity hypergraph to a Clocked Transition System. The requirements-level semantics has been implemented in a software tool (see Chapter 10). We use a variant of the Z notation [148]; Appendix A explains the notational conventions we use.

To stress the difference of both semantics with a Petri net token-game semantics, at the end of this chapter we define a Petri net-like token-game semantics for activity hypergraphs in an appendix.

Apart from the work that has been done on Clocked Transition Systems, mentioned below in Section 5.1, another source of inspiration has been the work of Damm et al. [46], in which a formal semantics of STATEMATE statecharts is defined in terms of a Clocked Transition System-like model. The layout of formulas has been heavily influenced by the writings of Leslie Lamport [114].

## 5.1 Clocked Transition System

A transition system specifies states and transitions between these states. In this thesis, states are assignments of values to variables. Such an assignment is called

a *valuation*. A valuation maps a variable to a value. From now on, we will use the term 'state' as a synonym for 'valuation'. For example, if $x$ is an integer, a possible valuation $\sigma$ could define $\sigma(x) = 10$. A transition from one state to another represents that some variables are assigned a different value, i.e., the valuation changes. Assume some variables *Var* and a (typed) data domain $\mathcal{D}$. Formally, a valuation $\sigma$ is a total, type preserving mapping from *Var* to $\mathcal{D}$:

$$\sigma : Var \longrightarrow \mathcal{D}$$

The set of all valuations on set *Var* is denoted $\Sigma(Var)$. A transition system whose states are valuations is called a *Kripke structure* [42].

A Clocked Transition System (CTS) [109, 123] extends a transition system with some extra variables, clocks, that measure the passage of time. Clocks increase uniformly whenever time progresses. They can be reset in system transitions. Clock variables always have type real.

There is one special clock variable $MC$, that represents the master clock, i.e. the global time. The master clock $MC$ can never be reset.

The other clocks are represented by set $RT$ of clock variables, the Running Timers. We will change the content of the set $RT$ at run time by adding and removing clocks. We therefore assume that there always "enough" timers available, i.e. not in $RT$, such that they can be added to $RT$. The reservoir of available clocks we call *ClockReservoir*.

Non-clock variables are called *discrete* variables in CTS jargon [109, 123]. They are represented by a set *Disc*.

Formally, a *Clocked Transition System* (CTS) is a tuple $(Var, \rightarrow, \sigma_{init})$ where:

- $Var = Disc \cup RT \cup \{MC\}$ is a finite set of variables,
- $\rightarrow \subseteq \Sigma(Var) \times \Sigma(Var)$ is the transition relation,
- $\sigma_{init} \in \Sigma(Var)$ is the initial valuation.

Instead of writing $(\sigma, \sigma') \in \rightarrow$, we write $\sigma \rightarrow \sigma'$.

This definition differs slightly from the original definition of a Clocked Transition System [109, 123]. In the original definition, Clocked Transition Systems have a constraint on all clocks, called the time progress condition or clock invariant, from which a special transition relation $\rightarrow_{tick}$ is implicitly derived that represents the passage of time. We will explicitly define a similar transition relation to represent the passage of time; in this definition we will enforce a constraint similar to the clock invariant.

The transition relation $\rightarrow$ is partitioned in two sets, *data transitions*, in which the clocks do not increase but can be reset and in which discrete variables can change arbitrarily, and *time transitions* in which clocks increase but discrete variables do not change. So data transitions are instantaneous, but time transitions are not. Note there is no transition possible in which both clocks are increased and some discrete variables change value.

A *path* of a CTS is an infinite sequence $\pi$ of valuations, $\pi = \sigma_0\sigma_1 \ldots$ satisfying:

- Initiation: $\sigma_0 = \sigma_{init}$,
- Consecution: for every $i = 0, 1, \ldots$, the valuation $\sigma_{i+1}$ is a $\longrightarrow$ successor of $\sigma_i$, i.e., $\sigma_i \longrightarrow \sigma_{i+1}$.

A *run* is a path satisfying

- Time divergence: The sequence $\sigma_0(MC)\sigma_1(MC)\ldots$ grows beyond any bound, i.e., the value of $MC$ increases beyond any bound.

Thus a run cannot have Zeno behaviour.

## 5.2 Step semantics

**Computing a step.** A node can become active or inactive during execution. If it is entered, it becomes active: the system is in the corresponding state. If the node is left, it becomes inactive: the system is not in the corresponding state. Since activity diagrams allow for the specification of parallelism (fork and join), more than one node can be active at the same time. All the active nodes together represent the global state, called the *configuration*. Since we use activity diagrams to model workflows, a configuration is a global state of the case (cf. page 12).

Formally, the configuration is a bag of nodes. The configuration is not a set, because a node can be active more than once at the same time. Figure 5.1 shows an example to illustrate this. If a customer wants some goods and not enough goods are in stock, the company can already send the goods in stock to the customer and produce the remaining goods in parallel. So Produce partial order and Take partial order from stock are active at the same time. If Take partial order from stock terminates, node Send partial shipment is entered. If next Produce partial order terminates, node Send partial shipment is entered again. Then two instances of node Send partial shipment are active at the same time, because the two shipments are processed in parallel. Thus, the configuration is a bag of nodes, rather than a set.

Let $C$ denote the current configuration, $C$ : bag *Nodes* and let $E$ be the bag of input events, $E$ : bag *Events*, to which the system responds by taking a step.

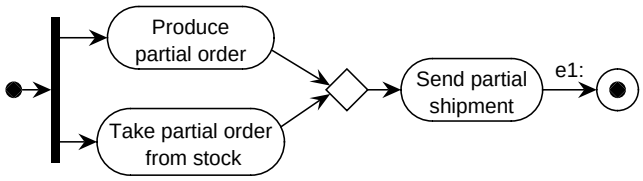



**Figure 5.1** *Multiple simultaneous instantiations of* Send partial shipment

A hyperedge is *relevant* in $C$ if its sources are contained in the current configuration. A hyperedge can be relevant more than once, since its source nodes can be more than once in the configuration. For example, if in Figure 5.1 the current configuration $C$ is [Send partial shipment, Send partial shipment], then two instances of e1 are relevant. A hyperedge cannot be relevant more often than the number of times any of its source nodes is in the configuration. The bag of relevant hyperedges, $relevant(C)$, is defined as follows:

$$\begin{aligned} relevant(C) \stackrel{\text{df}}{=} \{\, h \mapsto n \in \mathit{HyperEdges} \times \mathbb{N}_1 \mid \\ \mathit{settobag}(\mathit{source}(h)) \sqsubseteq C \\ \wedge\ n = \mathit{min}(\{\, C \sharp s \mid s \in \mathit{source}(h) \,\}) \\ \} \end{aligned}$$

where $\sqsubseteq$ denotes bag containment, function *settobag* maps a set into an equivalent bag:

$$\mathit{settobag}(S) \stackrel{\text{df}}{=} \{s \mapsto 1 \mid s \in S\}$$

and, given a set $X$ of natural numbers, $min(X)$ returns the minimum number of the numbers in $X$, and $B \sharp el$ counts the number of times element $el$ occurs in bag $B$.

A hyperedge is *enabled* iff it is relevant, it is triggered by an event in the input $E$, and its guard evaluates to true. A guard expression can contain variables. To evaluate a guard expression, each variable must have a value. A guard is evaluated in a valuation $\sigma$ by substituting for every variable $v$ its value $\sigma(v)$. If $g$ is true in valuation $\sigma$, this is written as $\sigma \models g$. For example, if $\sigma(x) = 10$ then $\sigma \models x \geq 5$ but $\sigma \not\models x = 8$. The bag $enabled_\sigma(C, E)$ of enabled hyperedges is defined formally as follows.

$$\begin{aligned} \mathit{enabled}_\sigma(C, E) \stackrel{\text{df}}{=} \{\, h \mapsto n \in \mathit{HyperEdges} \times \mathbb{N}_1 \mid \\ h \mapsto n \in \mathit{relevant}(C) \\ \wedge\ (\, \mathit{event}(h) \sqsubseteq E \vee \mathit{event}(h) = \bot \,) \\ \wedge\ \sigma \models \mathit{guard}(h) \\ \} \end{aligned}$$

where $\sqsubseteq$ is bag membership and as before $\bot$ denotes the absence of a trigger event (see page 34).

A special guard expression is the in predicate. Given a node $n$, in($n$) abbreviates $n \sqsubseteq C$.

Given a configuration $C$ and a bag $E$ of input events, a bag of hyperedges $H$ is defined to be consistent, written $consistent(C, E, H)$, iff all hyperedges can be taken at the same time. Some hyperedges cannot be taken at the same time, because either (i) they leave some sources node more often than possible in the

configuration, or (ii) some point-to-point event in $E$ triggers more than one hyperedge in $H$.

$$\begin{aligned} \mathit{consistent}(C,E,H) \quad \stackrel{\text{df}}{\Leftrightarrow} \quad & (\uplus_{h \sqsubseteq H} \mathit{settobag}(\mathit{source}(h))) \sqsubseteq C \\ & \wedge\ \forall\, e \sqsubset E \bullet \mathit{sendtype}(e){=}\mathsf{p2p} \ \Rightarrow \\ & \qquad \mathit{length}(\{\, h{\mapsto}n \in H \mid \mathit{event}(h){=}e \,\}) \leq E \sharp e \end{aligned}$$

where $\uplus$ denotes bag union, p2p stands for point-to-point (see page 33) and where $\mathit{length}(B)$ counts the number of elements in bag $B : X \longrightarrow \mathbb{N}_1$:

$$\begin{aligned} \mathit{length}(\{x \mapsto n\}) \quad & \stackrel{\text{df}}{=} \quad n \\ \mathit{length}(\{x \mapsto n\} \uplus B) \quad & \stackrel{\text{df}}{=} \quad n + \mathit{length}(B) \end{aligned}$$

Configuration $C$ is *interfering* iff some of the activities enabled by the activity nodes in $C$ update the same variable $v$, so they conflict.

$$\begin{aligned} \mathit{interfering}(C) \quad \stackrel{\text{df}}{\Leftrightarrow} \quad \forall\, a, b \sqsubset C \bullet \quad & a \neq b \ \Rightarrow\ \neg\, (a \lightning b) \\ & \wedge\ a = b \ \Rightarrow\ C \sharp a = 1 \end{aligned}$$

A bag of hyperedges $H$ is defined to be maximal iff for every enabled hyperedge $h$, the bag $H \uplus [\, h \,]$ is inconsistent or the configuration reached next is interfering. Notation $[\, h \,]$ denotes a bag that only contains $h$. We will define function *nextconfig* below.

$$\begin{aligned} \mathit{maximal}_\sigma(C,E,H) \quad \stackrel{\text{df}}{\Leftrightarrow} \quad & \forall\, h \sqsubset \mathit{enabled}_\sigma(C,E) \bullet h \not\sqsubseteq H \ \Rightarrow \\ & \qquad (\ \neg\ \mathit{consistent}(C,E,H \uplus [\, h \,]) \\ & \qquad \vee\ \mathit{interfering}(\mathit{nextconfig}(C, H \uplus [\, h \,]))\ ) \end{aligned}$$

Finally, predicate *isStep* defines a bag of hyperedges $S$ to be a step iff every hyperedge in $S$ is enabled, $S$ is maximal and consistent, and the next configuration is noninterfering. The two semantics that we will define in the next sections both use the predicate *isStep*.

$$\begin{aligned} \mathit{isStep}_\sigma(C,E,S) \quad \stackrel{\text{df}}{\Leftrightarrow} \quad & S \sqsubseteq \mathit{enabled}_\sigma(C,E) \\ \wedge \quad & \mathit{consistent}(C,E,S) \\ \wedge \quad & \neg\ \mathit{interfering}(\ \mathit{nextconfig}(C,S)\ ) \\ \wedge \quad & \mathit{maximal}_\sigma(C,E,S) \end{aligned}$$

This definition of a step is declarative. Steps can be computed, given some bag $H$ of enabled hyperedges, by splitting $H$ into maximal, consistent bags of hyperedges that do not lead to interfering next configurations.

**Effect of step on configuration.** By taking a step, some nodes are left and others are entered. Given a step $H$, the function *left* returns the bag of nodes that

are left if all the hyperedges in $H$ are taken and the function *entered* returns the bag of nodes that are entered if all the hyperedges in $H$ are taken:

$$left(H) \stackrel{\text{df}}{=} \biguplus_{h \sqsubseteq H} settobag(source(h))$$
$$entered(H) \stackrel{\text{df}}{=} \biguplus_{h \sqsubseteq H} settobag(target(h))$$

By taking a step the configuration changes. The function *nextconfig* returns the next configuration, given a configuration $C$ and a consistent bag of hyperedges $H$:

$$nextconfig(C, H) \stackrel{\text{df}}{=} (\, C \uplus left(H) \,) \uplus entered(H)$$

**Example.** Consider the example activity diagram in Figure 1.1 and its underlying activity hypergraph in Figure 3.6. As explained on page 40, we let activity *Check stock* update the boolean variable insufficient stock, activity *Check customer* the boolean variable customer ok, and *Handle payment* the boolean variable payment ok. Suppose the current configuration is [Check stock, Check customer]. In order to give an impression of all the different execution possibilities in this configuration, we have listed in Table 5.1 for each of the relevant inputs that may occur, the configuration that is entered subsequently by taking a step. The computed step is implied by the reached configuration.

| insufficient stock | true | true | false | false |
|---|---|---|---|---|
| customer ok | true | false | true | false |
| Terminating activity node | | | | |
| Check stock | [Check customer, Make production plan] | [Check customer, Make production plan] | [Check customer, WAIT-1] | [Check customer, WAIT-1] |
| Check customer | [Check stock, WAIT-2, Send bill] | [Check stock, WAIT-2, final] | [Check stock, WAIT-2, Send bill] | [Check stock, WAIT-2, final] |
| Check stock & Check customer | [Make production plan, WAIT-2, Send bill] | [Make production plan, WAIT-2, final] | [WAIT-1, WAIT-2, Send bill] | [WAIT-1, WAIT-2, final] |

**Table 5.1** *Possible next configurations for the activity hypergraph in Figure 3.6.* final *denotes one final node*

Note that in this configuration, given a certain input, there is only one possible step. The only hyperedges that might be inconsistent represent different branches of a decision. Every decision in the example is deterministic. Hence, no two edges that might be inconsistent are enabled at the same time. Therefore, for each bag of input events in this configuration, the calculated step is unique.

An example of a configuration in which, given a certain input, more than one step is possible, is configuration [WAIT-3,WAIT-4]. If the input is [after(2 weeks), receive payment], there are two possible steps, one entering node Send reminder and the other one entering node Handle payment. The WFS chooses arbitrarily one of these steps and takes it.

## 5.3 Requirements-level semantics

In the requirements-level semantics, we use the variables $C$, $I$, and $LVar$ as discrete variables for the Clocked Transition System (see Section 5.1):

$$Disc = \{ C, I \} \cup LVar$$

where, as before, $C$ is the configuration, $I$ is the current bag of input events, and $LVar$ is the set of local variables of the activity diagram.

We now explain how we model temporal events in the semantics. In UML, there are two kinds of temporal events, after and when events. after events are dealt with by timers in *ClockReservoir*, whereas when events are dealt with by the master clock $MC$. Remember that in Chapter 3, we parametrised each after($texp$) expression with the hyperedge $h$ it belongs to: after($texp, h$). For each after($texp$,$h$) expression, a timer is started as soon as $h$ becomes relevant; $texp$ time units after the timer was started a timeout is generated.

So, given a timer $t$ belonging to expression after($texp$,$h$), the deadline of $t$ is $texp$. Thus, we can associate with each timer a deadline, denoted by function $deadline : ClockReservoir \rightarrow \mathbb{N}_1$. In order to make sure that there is no confusion between two hyperedges with a similar after($texp$) constraint, a timer always belongs to at most one hyperedge. Function $hedge : ClockReservoir \longrightarrow HyperEdges$ associates with every timer its unique hyperedge.

Due to unboundedness of a node, there does not have to exist an upper bound on the number of times a hyperedge is relevant at the same time. For example, in Figure 5.2 node WAIT is unbounded: there is no bound on the maximum number of

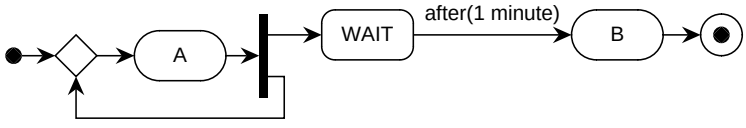



**Figure 5.2** *Example in which an unbounded number of timers is needed*

its active instances. So, the hyperedge leaving WAIT can be relevant unboundedly often. Therefore, infinitely many timers are needed to generate all the relevant timeouts. Hence, we assume that for every hyperedge enough timers, i.e. possibly unboundedly many, are available.

Temporal when events occur modulo a certain period. Thus, each when event $w$ specifies a set of points in time at which the event occurs. Denote this set by $\mathit{deadlines}(w)$. The set of all when events also specifies a set of points, namely the union of all the individual sets. Denote this set by $\mathit{deadlines}(\mathit{WhenEvents})$.

We now specify the transition relation $\rightarrow$ for the CTS in the requirements-level semantics. The transition relation consists of seven transition relations. We first give a brief explanation of each relation; then we will formalise each relation.

- relation $\rightarrow_{\mathit{time}}$ represents the passage of time: timers are increased;
- relation $\rightarrow_{\mathit{event}}$ represents the occurrence of some events;
- relation $\rightarrow_{\mathit{retrieve\_lvar}}$ represents the retrieval of the current values of the local variables. Retrieval is needed for evaluating guard conditions;
- relation $\rightarrow_{\mathit{unstable}}$ tests whether the current valuation is unstable. A valuation is unstable if there are some enabled hyperedges or the bag of input events is filled with some events;
- relation $\rightarrow_{\mathit{stable}}$ tests whether the current valuation is stable. If the valuation is stable, some new events can occur;
- relation $\rightarrow_{\mathit{step}}$ represents that a step is computed and taken according to the step semantics outlined in Section 5.2. The configuration and the bag of input events are updated;
- relation $\rightarrow_{\mathit{end}}$ represents the termination of the case.

In a requirements-level run, transitions must occur in a certain order. Figure 5.3 specifies the order of the transition relations in the requirements-level semantics. In the figure, a node represents a valuation. The initial valuation of an activity diagram is unstable by definition: a step is taken in order to leave this initial state and enter a stable state.

There is a loop of transitions $\rightarrow_{\mathit{step}}$ and $\rightarrow_{\mathit{unstable}}$ in Figure 5.3. So, more than one step can be taken in response to some event occurrences. To be precise, if some event occurs, the system state becomes unstable. The system then reacts by taking a step and entering a new state. If this state is unstable, then another step is taken. A maximal sequence of steps is called a *superstep*. The superstep begins and ends in a stable state and all the intermediary states of the superstep are unstable. As explained on page 50, the superstep can diverge; then the superstep never ends in a stable state but keeps on taking steps and entering unstable states. There may be more than one step possible in an unstable state; then one of these possible steps is chosen and taken.

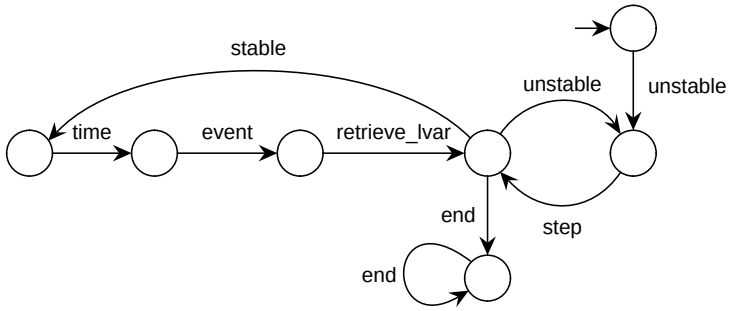



**Figure 5.3** *Execution cycle in requirements-level semantics*

Ideally, each workflow should terminate, i.e., reach the state in which the only possible transition is $\rightarrow_{end}$. But there are some workflows which fail to do so, either because they diverge, or because they deadlock. The verification approach presented in Chapter 10 allows, among others, to detect such failures.

We now formally define each of these seven relations.

Relation $\rightarrow_{time}$ defines the passage of time. Timers are increased by some real number $\Delta$. They cannot be increased beyond their deadline, because timeout events must be generated on time.

$$\begin{aligned}\sigma \rightarrow_{time} \sigma' \quad \stackrel{\text{df}}{\Leftrightarrow} \quad & \exists\, \Delta \in \mathbb{R} \bullet \Delta > 0 \\ & \wedge\ \sigma' = \sigma[\,\&_{t \in RT}\, t/\sigma(t){+}\Delta, MC/\sigma(MC){+}\Delta] \\ & \wedge\ \forall\, l \in \mathit{deadlines}(\mathit{WhenEvents}) \bullet \\ & \qquad \sigma(MC) < l\ \Rightarrow\ \sigma(MC){+}\Delta \leq l \\ & \wedge\ \forall\, t \in \sigma(RT) \bullet \\ & \qquad \sigma(t) < \mathit{deadline}(\mathit{hedge}(t)) \Rightarrow \\ & \qquad\quad \sigma(t){+}\Delta \leq \mathit{deadline}(\mathit{hedge}(t))\end{aligned}$$

Valuation $\sigma[x/val]$ assigns to variable $x$ value $val$ and to every other variable $y$, $y \neq x$, the value $\sigma(y)$. Symbol & denotes a bulk update:

$$\sigma[\&_{x \in X}\, x/val_x] \quad \stackrel{\text{df}}{=} \quad \sigma[x_1/val_1, .., x_n/val_n]$$

where $n = \#X$.

The defined constraints upon clocks use a guard condition, modelled with the implication ($\Rightarrow$), for the following reason. If a clock reaches its deadline, time can no longer pass, because a timeout must be generated. But afterwards, in a later time transition, if the clock has reached its deadline and has not been switched off, it should be able to increase beyond any bound. The consequent of the guard condition does not allow such an increase. That is why an implication is used.

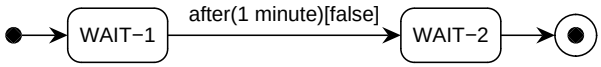



**Figure 5.4** *Example in which timer cannot be switched off after timeout*

Consider for example the activity diagram in Figure 5.4. Suppose the system enters configuration [WAIT-1]. After 1 minute, time can no longer progress for the corresponding timer and a timeout is generated (by relation $\rightarrow_{event}$ specified below). In the example, the edge leaving WAIT-1 can never be taken, because its guard condition is false. Thus, node WAIT-1 cannot be left. The defined constraint on clocks, including $MC$, in $\rightarrow_{time}$ allows the timer to keep on running even after it has reached its deadline. Thus, the constraint prevents that time stands still in case some timers have reached their deadlines but are not switched off.

Relation $\rightarrow_{event}$ defines that events occur. The only component that changes is $I$, the bag of input events. The non-occurrence of events is excluded: no change is not a change. The occurrence of events must satisfy some additional constraints, that we will discuss below. Line by line, the definition says that a bag $E$ of event occurrences is allowed if and only if:

- the bag is not empty;
- a broadcast event can only occur once;
- only activity nodes that are in the current configuration can terminate;
- if the master clock $MC$ has reached the deadline of a when event, the when event is generated;
- if $n$ after timers are the same, i.e. reach their deadline simultaneously, then $n$ after events should be raised. Two timers are the same if and only if they belong to the same hyperedge $h$ and have the same value, so they are started at the same time. Note that by definition, timers that belong to the same hyperedge have the same deadline.

$$
\begin{aligned}
\sigma \rightarrow_{event} \sigma' \quad \stackrel{\text{df}}{\Leftrightarrow} \quad & \exists\, E : \text{bag}\, \mathit{Events} \bullet \sigma' = \sigma[I/E] \\
& \wedge\ E \neq [\,] \\
& \wedge\ \forall\, e \in \mathit{Events} \bullet \mathit{sendtype}(e) = \mathsf{bc} \Rightarrow E \sharp e \leq 1 \\
& \wedge\ \forall\, t \in \mathit{TerminationEvents} \bullet \sigma'(I) \sharp t \leq \sigma(C) \sharp \mathit{term}(t) \\
& \wedge\ \forall\, we \in \mathit{WhenEvents} \bullet \sigma(MC) \in \mathit{deadlines}(we) \Leftrightarrow we \sqsubseteq E \\
& \wedge\ \forall\, t \in \sigma(RT) \bullet \sigma(t) = \mathit{deadline}(t) \Rightarrow \\
& \qquad\qquad E \sharp \mathit{event}(\mathit{hedge}(t)) = \#\mathit{same}(t, \sigma) \\
& \text{where } \mathit{same}(t, \sigma) = \{\, t' \in \sigma(RT) \mid \mathit{hedge}(t) = \mathit{hedge}(t') \\
& \qquad\qquad\qquad\qquad \wedge\ \sigma(t) = \sigma(t') \,\}
\end{aligned}
$$

Before the step can be computed, the valuation of the local variables in the database must be known. The valuation of these variables may have changed, because some activities have terminated (recorded in $I$), or because the environment has updated some variables. Relation $\rightarrow_{retrieve\_lvar}$ specifies that the new values of the local variables are retrieved. The valuation of variables that are observed or updated in some running activity does not change.

$$\begin{aligned}\sigma \rightarrow_{retrieve\_lvar} \sigma' \quad \stackrel{\text{df}}{\Leftrightarrow} \quad & \sigma(C) = \sigma'(C) \\ \wedge\ & \sigma(I) = \sigma'(I) \\ \wedge\ & \sigma(MC) = \sigma'(MC) \\ \wedge\ & \forall\, t \in RT \bullet \sigma(t) = \sigma'(t) \\ \wedge\ & \forall\, a{\in}AN \bullet a \sqsubseteq \sigma(C) \uplus terminated(\sigma(C), \sigma(I)) \Rightarrow \\ & \quad \forall\, v{\in}LVar \bullet v{\in}Obs(a) \cup Upd(a) \Rightarrow \sigma(v) = \sigma'(v)\end{aligned}$$

where, given a configuration $C$ and input $I$, the function *terminated* returns the bag of terminated activity nodes of $C$.

$$\begin{aligned}terminated(C, I) \quad = \quad & \{\, a \mapsto n \in AN \times \mathbb{N}_1 \mid \exists\, e \in TerminationEvents \bullet \\ & \qquad a \sqsubseteq C \wedge e \sqsubseteq I \wedge a = term(e) \wedge n = I \sharp e \,\}\end{aligned}$$

where as before $b \sqsubseteq B$ is true iff $b$ is member of bag $B$.

A valuation $\sigma$ is stable iff there are no enabled hyperedges and the bag of input events is empty:

$$\sigma \models stable \quad \stackrel{\text{df}}{\Leftrightarrow} \quad enabled_\sigma(\sigma(C), \sigma(I)) = \emptyset \ \wedge\ \sigma(I) = [\,]$$

Transitions $\rightarrow_{unstable}$ and $\rightarrow_{stable}$ test whether a valuation is unstable or stable. Both transitions have lower priority than transition $\rightarrow_{end}$, defined below.

$$\begin{aligned}\sigma \rightarrow_{unstable} \sigma' \quad & \stackrel{\text{df}}{\Leftrightarrow} \quad \sigma = \sigma' \wedge \sigma \not\models stable \wedge \sigma \not\rightarrow_{end} \sigma' \\ \sigma \rightarrow_{stable} \sigma' \quad & \stackrel{\text{df}}{\Leftrightarrow} \quad \sigma = \sigma' \wedge \sigma \models stable \wedge \sigma \not\rightarrow_{end} \sigma'\end{aligned}$$

Predicate $\rightarrow_{end}$ tests whether an activity diagram has ended.

$$\begin{aligned}\sigma \rightarrow_{end} \sigma' \quad \stackrel{\text{df}}{\Leftrightarrow} \quad & \sigma = \sigma' \\ \wedge\ & \sigma(I) = [\,] \\ \wedge\ & \forall\, n \in Nodes \bullet n \sqsubseteq \sigma(C) \Rightarrow n \in FN\end{aligned}$$

Note that if $\sigma \rightarrow_{end} \sigma'$ then $\sigma \models stable$, because final nodes have no outgoing hyperedges.

We next define the step transition relation $\rightarrow_{step}$. A step is computed as described in Section 5.2. Line by line, the $\rightarrow_{step}$ definition says that a step is done between $\sigma$ and $\sigma'$ iff:

- there is a step $S$ (using the predicate *isStep* defined in Section 5.2);

- the variables that are contained in the guards of the hyperedges in $S$ are not being updated in some non-terminated activity (otherwise an inconsistent value could be read);
- there is a set $T$ of timers that can be turned on;
- $\sigma$ is then updated into $\sigma'$ by computing the next configuration if step $S$ is taken (using the function *nextconfig* defined in Section 5.2), putting the generated events in $I$, initialising the new timers $T$, and updating $RT$ by removing the timers that have become irrelevant (set *OffTimers*) and adding the new timers $T$.

$$
\begin{aligned}
\sigma \rightarrow_{step} \sigma' \quad \stackrel{\text{df}}{\Leftrightarrow} \quad & \exists\, S : \text{bag}\, \mathit{HyperEdges} \bullet \mathit{isStep}_\sigma(\sigma(C), \sigma(I), S) \\
& \wedge\ \forall\, a \in AN \bullet a \sqsubseteq C \uplus \mathit{terminated}(\sigma(C), \sigma(I)) \Rightarrow \\
& \qquad \mathit{Upd}(a) \cap (\bigcup_{h \sqsubseteq S} \mathit{var}(\mathit{guard}(h)) ) = \varnothing \\
& \wedge\ \exists\, T \subseteq \mathit{ClockReservoir} \bullet \mathit{NewTimers}(\sigma(C), S, \sigma(RT), T) \\
& \wedge\ \sigma' = \sigma[C/\mathit{nextconfig}(\sigma(C), S), \\
& \qquad I/\mathit{settobag}(\mathit{generated}(S)), \\
& \qquad \&_{t \in T}\, t/0, \\
& \qquad RT/(\sigma(RT) \setminus \mathit{OffTimers}(\sigma(C), S, \sigma(RT)) \cup T)]
\end{aligned}
$$

where $\mathit{var}(g)$ denotes the variables guard $g$ tests, given a bag of hyperedges, function *generated* returns the set of generated events. The function is defined recursively.

$$
\begin{aligned}
\mathit{generated}([\,]) \quad & \stackrel{\text{df}}{=} \quad \varnothing \\
\mathit{generated}([\,h\,] \uplus H) \quad & \stackrel{\text{df}}{=} \quad \mathit{sendactions}(h) \cup \mathit{generated}(H)
\end{aligned}
$$

We next define function *OffTimers* and predicate *NewTimers*. Function *OffTimers* returns all timers in $RT$ that can be switched off because their corresponding hyperedges are irrelevant in the next configuration, even though these were relevant in the current configuration. Predicate *NewTimers* is true iff all timers in $T$ are off but can be turned on, i.e. they are not in $RT$, and moreover for every instance of an hyperedge that becomes relevant there is a timer in $T$. The definition of *NewTimers* requires that for every hyperedge there are always enough new timers, i.e. non-running so not in $RT$, available. That is why we require that for every hyperedge $h$ there are unboundedly many timers belonging to $h$ available in *ClockReservoir*.

$$
\begin{aligned}
\mathit{OffTimers}(C, S, RT) \ \stackrel{\text{df}}{=} \ & \{\, t \in RT \mid \mathit{hedge}(t) \sqsubseteq \mathit{oldrel} \,\} \\
& \text{where } \mathit{oldrel} = \mathit{relevant}(C) \uplus \mathit{relevant}(\mathit{nextconfig}(C, S))
\end{aligned}
$$

$$\begin{aligned} NewTimers(C, S, RT, T) \;\stackrel{\text{df}}{\Leftrightarrow}\; & \\ & T \subseteq \{\, t \in ClockReservoir \mid t \notin RT \wedge hedge(t) \sqsubseteq newrel \,\} \\ & \wedge\, \forall\, h \sqsubseteq newrel \bullet newrel \sharp h = \#\{\, t \in T \mid hedge(t) = h \,\} \\ & \text{where } newrel = relevant(nextconfig(C, S)) \uplus relevant(C) \end{aligned}$$

**Initial valuation.** In the initial valuation $\sigma_0$, the configuration only contains one copy of *initial* and the input is empty. There are no timers running, so set $RT$ is empty. The other variables, including master clock $MC$, must be initialised with an appropriate value.

$$\begin{aligned} \sigma_0 \;\models\; & C = [\, initial \,] \\ \wedge\; & I = [\,] \\ \wedge\; & RT = \varnothing \end{aligned}$$

**Execution algorithm.** Figure 5.5 shows an informal execution algorithm for UML activity diagrams. The execution algorithm models the same behaviour as the formal requirements-level semantics defined above, but in a less formal way. The algorithm may however be more intuitive and easier to understand than the formal definitions above.

## 5.4 Implementation-level semantics

In the implementation-level semantics, the system reacts by taking an event from the queue and processing it. To take a hyperedge with no trigger event, a special event is needed in this semantics. A *completion event* is defined in the implementation-level semantics of UML [150] as the event that is generated when a wait state is entered[1]. Completion events should not be confused with termination events: the latter are defined by us and refer to activity nodes. There is an ambiguity in the UML standard [150] concerning the triggering of completion hyperedges, i.e. hyperedges that do not have an explicit trigger event. Can a completion event of node n only trigger hyperedges leaving n, or can a completion event also trigger say a completion hyperedge that does not have n in its sources? We assume here that a completion event can trigger arbitrarily many other completion hyperedges. This assumption can easily be adapted to the more strict case where a completion event only triggers outgoing hyperedges of some specific node.

Let set *CompletionEvents* represent the set of completion events. The bijective function *comp* gives for each completion event the wait state upon whose entry the completion event is generated.

$$comp : CompletionEvents \rightarrowtail\!\!\!\!\!\rightarrow WN$$

[1] According to UML 1.4, a completion event occurs when all entry actions and do-activities in the current state have completed. In this thesis, however, wait states have no entry actions. On page 26 we explain why we do not use do-activities.

- Initialise;
- While ($C \neq$ final configuration) do
  - Repeat until $I \neq [\,]$; `// wait for input events`
  - Retrieve the valuation of the local variables;
  - Take a superstep:
    - Repeat
      1. Compute a step;
      2. Compute the internal events generated in the step;
      3. Compute the next configuration;
      4. Update $C$ with the next configuration;
      5. Empty the input $I$ and fill it with the generated internal events;
      6. Switch new relevant timers on and irrelevant timers off;
    - Until $I$ is empty and there are no enabled hyperedges;
- od ;

**Figure 5.5** *Execution algorithm for activity diagrams in requirements-level semantics*

We now turn to the variables used in the implementation-level semantics. There are two differences with the variables used in the requirements-level semantics. First, in the implementation-level semantics, we no longer have the bag $I$ of input events. Instead of $I$ we have two new variables: a queue $Q$ in which events that occur are stored, and a variable $re$ in which the router event, i.e. the single event that the Router is currently processing, is stored.

The queue $Q$ is filled by the environment. As in the requirements-level semantics, at each moment in time a bag of events can occur (and as in the requirements-level semantics one broadcast event cannot occur more than once at the same time). According to the UML [150], events in $Q$ are ordered, but the precise ordering used is left open. The only ordering rule that is given is that that completion events have priority over non-completion events. We will assume a FIFO order here but any other order compatible with the UML ordering rule can be chosen. We therefore model a queue as a sequence of a bag of events, $Q : \mathrm{seq}\,\mathrm{bag}\,\mathit{Events}$. Events occurring simultaneously are unordered: they belong to the same bag.

In order to give a simple formalisation of priority of completion events, we assume that the completion events are kept in a separate queue $Q_{comp} : \mathrm{seq}\,\mathit{CompletionEvents}$. This does not imply that an implementation of this semantics should

store completion events in a separate queue!

Router event $re$ either contains an event or is empty, $re \in \mathit{Events} \cup \{\bot\}$. Upon the start of a reaction, the Router picks an event from $Q$ and puts it in $re$. When the Router finishes its reaction, the event is removed from $re$. So, if $re = \bot$, the router is not busy processing, otherwise it is.

A second difference is that the Router uses a variable $S$ to store the current step that is being taken. This variable is not used in the requirements-level semantics.

Summarising, we use the following variables for the Clocked Transition System in the implementation-level semantics.

$$\mathit{Disc} = \{\ C, Q, Q_{comp}, re, S\ \} \cup \mathit{LVar}$$

As before, $C$ is the configuration and $\mathit{LVar}$ is the set of local variables.

In the implementation-level semantics, the transition relation consists of twelve transition relations that we describe next:

- relation $\longrightarrow_{time}$ represents the passage of time: timers are increased;
- relation $\longrightarrow_{event}$ represents the occurrence of some events;
- relation $\longrightarrow_{pick\_event}$ represents the Router picking an event from the queue and putting it in $re$;
- relation $\longrightarrow_{retrieve\_lvar}$ represents the retrieval of the current values of the local variables. Their current values are needed to evaluate guard conditions;
- relation $\longrightarrow_{compute\_step}$ represents the computation of a step. The computed step is put in $S$;
- next, a step is taken:
  - relation $\longrightarrow_{leave\_config}$ represents the first part of taking a step by leaving some state nodes;
  - relation $\longrightarrow_{remove\_offtimers}$ represents that some timers are turned off because they have become irrelevant;
  - relation $\longrightarrow_{generate\_comp}$ represents the generation of completion events;
  - relation $\longrightarrow_{generate\_internal}$ represents the generation of named internal events in hyperedges;
  - relation $\longrightarrow_{add\_newtimers}$ represents that some timers are turned on because they have become relevant;
  - relation $\longrightarrow_{enter\_config}$ represents the entering of some state nodes;
- relation $\longrightarrow_{end}$ represents the termination of the case.

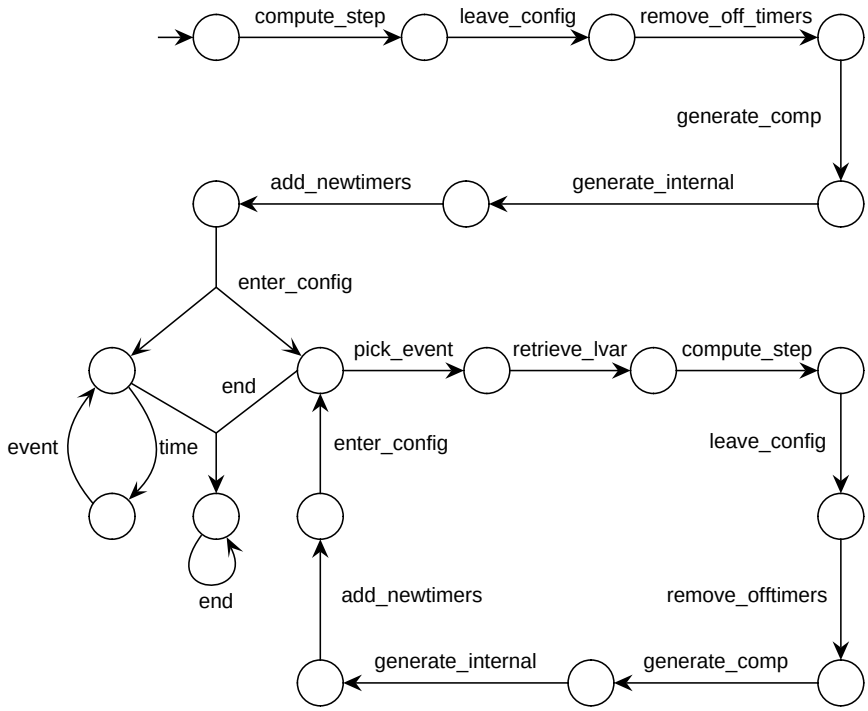



**Figure 5.6** *Execution cycle in implementation-level semantics*

The order of these transition relations in the implementation-level semantics is shown in Figure 5.6. As in the requirements-level semantics, the initial state of the activity diagram is unstable by definition. It is left by computing and taking a step.

The relation of the cycle in the implementation-level semantics (Figure 5.6) with the cycle in the requirements-level semantics (Figure 5.3) is as follows. Transitions $\rightarrow_{compute\_step}$, $\rightarrow_{leave\_config}$ through $\rightarrow_{enter\_config}$ correspond to transition $\rightarrow_{step}$ in the requirements-level semantics. Transition $\rightarrow_{pick\_event}$ is not present in the requirements-level semantics, as there is no Router in that latter semantics. Requirements-level transitions $\rightarrow_{stable}$ and $\rightarrow_{unstable}$ are not present at the implementation-level, as the queue is polled for events in that semantics.

Note there are two hyperedges in the figure (one with label enter_config and one with label end). The hyperedge labelled enter_config creates two parallel branches. The hyperedge labelled end joins the two parallel branches. In terms of the abstract WFS architecture depicted in Figure 2.3 on page 17, the left branch is behaviour of the environment and the Clock Manager, whereas the right branch is behaviour of the Router. The queue is the buffer that connects these parallel components.

Transition $\rightarrow_{time}$ can never happen concurrently with a parallel transition. Transition $\rightarrow_{event}$ can happen concurrently with parallel transitions; for example, $\rightarrow_{event}$ and $\rightarrow_{pick\_event}$ can happen simultaneously. This parallelism is not reflected in the formal definition below, but can be easily added by introducing some new relations that define these concurrent transitions, using the definitions for the atomic transitions that we will specify below. For example, the concurrent transition for the example above could be defined as:

$$\rightarrow_{event/pick\_event} \stackrel{\text{df}}{=} \rightarrow_{event} \fatsemi \rightarrow_{pick\_event}$$

which is equivalent to the following definition:

$$\rightarrow_{event/pick\_event} \stackrel{\text{df}}{=} \rightarrow_{pick\_event} \fatsemi \rightarrow_{event}$$

where $\fatsemi$ denotes relational composition, i.e. the intermediary state does not exist. We do not define these concurrent transitions, in order to avoid a blow up of definitions and notation. Nevertheless, the proofs in the next chapter, in which we show the correspondence between both semantics, carry easily over to this concurrent setting (due to the definition of relational composition $\fatsemi$).

We now proceed to formally define the transition relations. Relation $\rightarrow_{time}$ defines the passage of time. This definition is the same as in the previous section and therefore not shown here.

The $\rightarrow_{event}$ is almost the same as the definition in the previous section. The only differences with the previous definition are the first line, which specifies that events are put at the end of the queue ($\frown$ denotes concatenation on sequences), and the fourth line, which deals with termination events. The behaviour specified in the last line, that temporal events are input to the WFS, is behaviour of the clock manager (cf. Figure 2.3).

$$\begin{aligned}
\sigma \rightarrow_{event} \sigma' \quad \Leftrightarrow \quad & \exists\, E : \text{bag}\, Events \bullet \sigma' = \sigma[Q/\sigma(Q) \frown E] \\
& \wedge\ E \neq [\,] \\
& \wedge\ \forall\, e \in Events \bullet sendtype(e) = bc \Rightarrow E \sharp e \leq 1 \\
& \wedge\ \forall\, t \in TerminationEvents \bullet \\
& \qquad\qquad (\sigma'(Q) \uplus [\,\sigma'(re)\,]) \sharp t \ \leq\ \sigma(C) \sharp term(t) \\
& \wedge\ \forall\, we \in WhenEvents \bullet \sigma(MC) \in deadlines(we) \Rightarrow we \sqsubseteq E \\
& \wedge\ \forall\, t \in \sigma(RT) \bullet \sigma(t) = deadline(hedge(t)) \Rightarrow \\
& \qquad\qquad E \sharp event(hedge(t)) = \#same(t, \sigma) \\
& \text{where } same(t, \sigma) = \{\, t' \in \sigma(RT) \mid hedge(t) = hedge(t') \\
& \qquad\qquad\qquad\qquad \wedge\ \sigma(t) = \sigma(t') \,\}
\end{aligned}$$

Next, we model the behaviour of the Router. The Router first picks an event from the queue and puts it in *re*. Completion events have priority over non-completion events. As explained on page 68, completion events are stored in a

separate queue $Q_{comp}$.

$$\begin{aligned}\sigma \rightarrow_{pick\_event} \sigma' \quad \overset{\text{df}}{\Leftrightarrow} \quad & \exists\, e_c \in Q_{comp} \bullet \sigma' = \sigma[Q_{comp}/\sigma(Q_{comp}) \uplus [\, e_c \,], re/e_c] \\ & \vee\ Q_{comp} = \langle\rangle \wedge \exists\, e \in Q \bullet \sigma' = \sigma[Q/\sigma(Q) \uplus [\, e \,], re/e]\end{aligned}$$

Next, the Router retrieves the current valuation of the local variables from the database system. As terminated activities do not update variables anymore, new valuations of local variables can be retrieved when the corresponding activity termination events are in the queue or processed by the router.

$$\begin{aligned}\sigma \rightarrow_{retrieve\_lvar} \sigma' \quad \overset{\text{df}}{\Leftrightarrow} \quad & \sigma(C) = \sigma'(C) \\ & \wedge\ \sigma(Q) = \sigma'(Q) \\ & \wedge\ \sigma(Q_{comp}) = \sigma'(Q_{comp}) \\ & \wedge\ \sigma(re) = \sigma'(re) \\ & \wedge\ \sigma(S) = \sigma'(S) \\ & \wedge\ \sigma(MC) = \sigma'(MC) \\ & \wedge\ \forall\, t \in RT \bullet \sigma(t) = \sigma'(t) \\ & \wedge\ \forall\, a \in AN \bullet \\ & \quad a \sqsubseteq C \uplus terminated(\, \sigma(C), \sigma(Q) \uplus [\, \sigma(re) \,]\,) \Rightarrow \\ & \quad \forall\, v \in LVar \bullet v \in Upd(a) \cup Obs(a) \Rightarrow \sigma(v) = \sigma'(v)\end{aligned}$$

Next, the Router computes a step and stores it in $S$. As in the requirements-level semantics, the variables that are contained in the guards of the hyperedges in $S$ should not be updated in some non-terminated activity.

$$\begin{aligned}\sigma \rightarrow_{compute\_step} \sigma' \quad \overset{\text{df}}{\Leftrightarrow} \quad & \exists\, S' : \text{bag}\, HyperEdges \bullet isStep_\sigma(\sigma(C), [\, \sigma(re) \,], S') \\ & \wedge\ \sigma' = \sigma[S/S'] \\ & \wedge\ \forall\, a \in AN \bullet \\ & \quad a \sqsubseteq C \uplus terminated(\sigma(C), \sigma(Q) \uplus [\, \sigma(re) \,]) \Rightarrow \\ & \quad Upd(a) \cap (\bigcup_{h \sqsubseteq S'} var(guard(h))\,) = \varnothing\end{aligned}$$

Then, the Router starts taking $S$ by first leaving the current configuration.

$$\sigma \rightarrow_{leave\_config} \sigma' \quad \overset{\text{df}}{\Leftrightarrow} \quad \sigma' = \sigma[C/\sigma(C) \uplus left(\sigma(S))]$$

Next, some timers are switched off, because their hyperedges have become irrelevant.

$$\sigma \rightarrow_{remove\_offtimers} \sigma' \overset{\text{df}}{\Leftrightarrow} \sigma' = \sigma[RT/\sigma(RT) \setminus OffTimers(\sigma(C), \sigma(S), \sigma(RT))]$$

The queue is updated with completion events.

$$\sigma \rightarrow_{generate\_comp} \sigma' \quad \overset{\text{df}}{\Leftrightarrow} \quad \sigma' = \sigma[Q_{comp}/\sigma(Q_{comp}) \frown comp\_event(\sigma(S))]$$

where $\frown$ denotes concatenation of sequences and function *comp_event* returns the bag of completion events that are generated if step $H$ is taken:

$$\begin{aligned} comp\_event(H) \quad \stackrel{\text{df}}{=} \quad & \{\, c \mapsto n \in CompletionEvents \times \mathbb{N}_1 \mid \exists\, w \in WN \bullet \\ & w \sqsubseteq entered(H) \wedge comp(c) = w \wedge n = entered(H) \sharp w \,\} \end{aligned}$$

Then the queue is updated with generated events.

$$\sigma \rightarrow_{generate\_internal} \sigma' \quad \stackrel{\text{df}}{\Leftrightarrow} \quad \sigma' = \sigma[Q / \sigma(Q) \frown generated(\sigma(S))]$$

Then, some new timers are switched on, because their hyperedges have become relevant.

$$\begin{aligned} \sigma \rightarrow_{add\_newtimers} \sigma' \quad \stackrel{\text{df}}{\Leftrightarrow} \quad & \exists\, T \subseteq ClockReservoir \bullet \\ & NewTimers(\sigma(C), \sigma(S), \sigma(RT), T) \\ & \wedge\ \sigma' = \sigma[RT / \sigma(RT) \cup T, \\ & \qquad \&_{t \in T}\, t / 0] \end{aligned}$$

Finally, the Router enters the new configuration. Processing of the event in *re* has completed, so the event is removed from *re*.

$$\sigma \rightarrow_{enter\_config} \sigma' \quad \stackrel{\text{df}}{\Leftrightarrow} \quad \sigma' = \sigma[C / \sigma(C) \uplus entered(\sigma(S)), re / \bot]$$

The systems ends iff the configuration only contains final nodes and the queue is empty.

$$\begin{aligned} \sigma \rightarrow_{end} \sigma' \quad \stackrel{\text{df}}{\Leftrightarrow} \quad & \sigma = \sigma' \\ & \wedge\ \sigma(Q) = [\,] \\ & \wedge\ \sigma(Q_{comp}) = [\,] \\ & \wedge\ \sigma(re) = \bot \\ & \wedge\ \forall\, n \in Nodes \bullet n \sqsubseteq \sigma(C) \ \Rightarrow\ n \in FN \end{aligned}$$

**Initial valuation.** In the initial valuation $\sigma_0$, the configuration only contains one copy of *initial* and the input is empty. There are no timers running, so set $RT$ is empty. The other variables, including master clock $MC$, must be initialised with an appropriate value.

$$\begin{aligned} \sigma_0 \quad \models \quad & C = [\, initial \,] \\ & \wedge\ Q = [\,] \\ & \wedge\ Q_{comp} = [\,] \\ & \wedge\ re = \bot \\ & \wedge\ S = [\,] \\ & \wedge\ RT = \varnothing \end{aligned}$$

**Execution algorithm.** Figure 5.7 shows an informal execution algorithm for activity diagrams under the formal implementation-level semantics. As the previous algorithm, this algorithm is less precise than the formal semantics, but may be more intuitive and easier to understand.

**Run to completion.** The OMG semantics [150] of UML statecharts (and thus the OMG semantics of activity diagrams; see Chapter 9) satisfies run-to-completion, meaning that an event can only be processed if processing of the previous event

---

- Initialise;
- While (true) do
  `// wait for input events`
  receive input events;
- od;

- While ($C \neq$ final configuration) do
  - If ($Q \neq [\,]$) then
    1. Pick an event from $Q$ and put it in $re$;
    2. Retrieve the valuation of the relevant variables;
    3. Compute a step and store it in $S$;
       `// start taking` $S$
    4. Update $C$ by removing the source nodes of the hyperedges in $S$;
    5. Remove the timers that have become irrelevant;
    6. Generate completion events and update $Q$ with them;
    7. Generate send actions and put them in $Q$;
    8. Add some new timers because their sources become relevant in a moment;
    9. Update $C$ by adding the target nodes of the hyperedges in $S$;
       `// finish taking` $S$
- od; `// workflow terminated`
- abort;

---

**Figure 5.7** *Execution algorithm for activity diagrams in implementation-level semantics*

has fully completed. Since we do not have call events, this property is trivially satisfied by our implementation-level semantics (and even by our requirements-level semantics).

## Appendix: Token-game semantics

In order to give an impression what a Petri net-like token-game semantics for activity hypergraphs would look like, and how it differs from our two semantics, we define a token-game semantics for activity hypergraphs in this section.

Under the token-game semantics, the only variable that is used is the configuration.

$$Disc = \{C\}$$

The transition relation $\longrightarrow$ is defined as follows.

$$\sigma \longrightarrow \sigma' \quad \stackrel{\text{df}}{\Leftrightarrow} \quad \exists\, h \in \mathit{relevant}(\sigma(C)) \bullet \sigma' = \sigma[C/\mathit{nextconfig}(\sigma(C), [\,h\,])]$$

Under the token game semantics, every relevant hyperedge $h$ can be taken. It is left undetermined when exactly $h$ is taken: under this semantics, the system itself, i.e., the WFS, decides when $h$ is taken and does not look at its environment. We think this does not accurately reflect WFS behaviour: as a WFS is reactive, $h$ should be taken when it is triggered by some change in the environment (see Section 4.2).

In the initial valuation, the configuration only contains the initial node.

$$\sigma_0 \quad \models \quad C = [\,\mathit{initial}\,]$$

Every configuration reachable under the requirements-level or implementation-level semantics, is also reachable under the token-game semantics. A step under either the requirements-level or implementation-level semantics can be simulated under the token-game semantics by taking the hyperedges in the step one by one (remember the hyperedges in a step are consistent, i.e., taking one of the hyperedges in the step does not disable any of the other hyperedges in the step). The reverse, however, does not hold. For example, in Figure 1.1 (Figure 3.6) configuration [final, Send bill] is reachable under the token-game semantics, because guard customer ok is not interpreted in this semantics. But configuration [final, Send bill] is unreachable under both the requirements-level and implementation-level semantics, since in both these semantics guard labels are interpreted.

In Chapter 8 we focus more in general on the differences between Petri nets and our two semantics.

# Chapter 6

# Relation between the two formal semantics

In the previous chapter we have defined two execution semantics for activity diagrams. In this chapter we look at differences and similarities between these two semantics. The goal is to identify a class of activity diagrams that behave similarly under both semantics. Based on this study, we will identify in Chapter 10 a class of functional requirements that are *insensitive* to the particular semantics that is used, requirements-level or implementation-level. An insensitive requirement holds under the requirements-level semantics iff the requirement holds under the implementation-level semantics. To verify an insensitive requirement, we will use the requirements-level semantics. The obtained verification result carries over to the implementation-level semantics, even though the requirements-level semantics is considerably more easy to verify than the implementation-level semantics.

We begin by observing that, at a certain level of abstraction, the requirements-level semantics (RLS) and implementation-level semantics (ILS) behave similarly. Consider for example the workflow of the production company (see Figure 1.1). If the system is in configuration [Receive order] and activity node Receive order terminates, under both the RLS and ILS the next configuration will be [Check stock, Check customer]. But the moment in time at which this next configuration is reached differs in both semantics: under the RLS the next configuration is entered immediately whereas under the ILS the next configuration is entered after a finite, non-zero amount of time. In other words, the RLS and ILS have similar behaviour to the extent that the result of a reaction to some input events is the same. But each of the semantics may reach this result in its own way.

Unfortunately, there are activity diagrams[1] that have different observable behaviour under the RLS and ILS. In the first part of this chapter, we will list several

[1] We use the term 'activity diagram' even though we actually mean an activity hypergraph. This is not harmful, however, as each activity diagram maps into a unique activity hypergraph.

issues in which an activity diagram under the RLS has different behaviour than the same activity diagram under the ILS. Since our aim is to find common behaviour for both semantics, we must resolve these issues. There are three possible solutions.

- Put some extra constraints on the environment, so that activity diagrams behave the same according to both semantics. We label such a solution 'env'.
- Put some extra constraints on activity diagrams, so that activity diagrams not satisfying the constraints, are ruled out. We prefer constraints on the syntax of activity diagrams, but the constraints can also refer to the semantics of activity diagrams. We label such a solution 'act'.
- Put some extra constraints on the defined semantics, so that activity diagrams behave the same according to both semantics. So the semantics is changed. We label such a solution 'sem'.

We prefer the second solution. The first and third solution would require additional constraints on environment and semantics, whereas the focus is just to look at similar requirements in both semantics, not to redefine the semantics or put extra constraints upon the environment! Nevertheless, in some cases it is impossible to define extra constraints on activity diagrams; then we have to resort to defining an extra constraint on the environment or semantics.

To clarify the presentation, we introduce an intermediate level between the requirements-level and the implementation-level semantics, namely a restricted version of the implementation-level semantics. In this restricted version of the ILS, external events only occur when the current state is stable: the queue is empty and the Router is not busy processing an event. We call this restricted version the *stable implementation-level semantics* (stable ILS for short).

The structure of this chapter is as follows. Section 6.1 discusses the differences between the RLS and stable ILS on the one hand, and between the stable ILS and ILS on the other hand. We list several issues in which the behaviour of activity diagrams under the RLS semantics is different from the behaviour under the ILS. For each issue, we will number the solution we choose for resolving the difference in behaviour. The preferred solution is to formulate a constraint to rule out activity diagrams in which these differences occur, but sometimes this is impossible and we have to put a constraint on the environment or semantics. In Section 6.2 we prove that activity diagrams satisfying these constraints indeed behave similarly in both the RLS and ILS. Section 6.3 winds up with conclusions.

## 6.1 Differences between the two semantics

### 6.1.1 RLS and stable ILS

**Updating of variables by environment.** Because in the RLS events are processed immediately, the valuation of a guard at the time the event occurs and

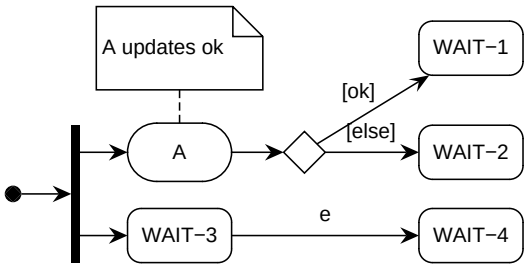



**Figure 6.1** *Example for issue "Updating of variables by environment"*

at the time the event is processed is the same. In the ILS and stable ILS, after an event has occurred and before the event is processed, the environment may change the valuation of a guard. The same activity diagram can therefore behave differently in the requirements-level and implementation-level semantics.

For example, suppose that in Figure 6.1 the system is in configuration [A,WAIT-3] and activity node A terminates and e occurs at the same time. Moreover, $A$ has set ok to true. Then under the RLS, the only possible next configuration is [WAIT-1,WAIT-4]. In the ILS, one event at a time is processed. Suppose e is processed before the activity termination event. While e is being processed, the environment may set ok to false. Since $A$ has terminated, the variable ok is no longer locked in the database system. So in that case, under the ILS configuration [WAIT-2,WAIT-4] is entered. This configuration cannot be reached under the RLS in this particular example.

We solve this difference in behaviour between RLS and stable ILS by forbidding the environment to update case attributes (control data). Note that this difference occurs in every activity diagram in which activities update variables that occur in some guard conditions of the activity diagram. It is therefore not feasible to define constraints on activity diagrams: those would rule out almost every activity diagram.

C1 The environment does not update case attributes (control data). (env)

**Locking of variables.** Before a relevant hyperedge can be taken, its guard needs to be evaluated. A guard cannot be evaluated if it refers to some variable $v$ that is currently being updated in some activity. In that case, we say $v$ is *locked*. If the guard refers to a variable that is locked, the activity diagram may behave differently under RLS and ILS.

For example, in Figure 6.2, variable ok is tested in edge e1 that is in a thread parallel with activities $A$ and $B$ that both update ok. Since $A$ and $B$ both update ok, they are interfering. Edge e1 can only be taken if variable ok is not locked, in other words, if $A$ and $B$ are not active.

Suppose the current configuration is [A,WAIT-2,WAIT-4] and node A terminates

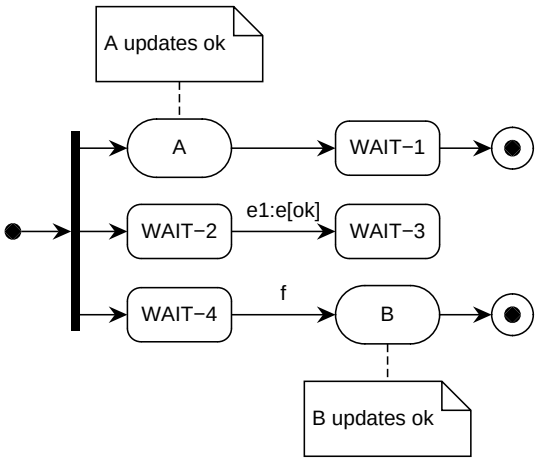



**Figure 6.2** *Example for issue "Locking of variables"*

and e and f occur. Under the RLS, the only possible next configuration is [WAIT-1,WAIT-3,B]. Since activity $A$ is not running anymore, edge e1 is always taken in this particular case. Under the ILS, suppose f is processed before e. Then node B is entered and activity $B$ becomes active before e1 is taken. If $B$ is active, however, variable ok becomes locked. Thus, edge e1 cannot be taken anymore. So, then the next configuration under the ILS is [WAIT-1,WAIT-2,B]; this configuration is not reachable under the RLS for this particular input.

The difference between RLS and ILS in this particular example is due to the parallel event processing in the RLS and the single-event processing in the ILS. Due to single-event processing, in the ILS a variable can become locked after one (external) event has been responded to, whereas under the RLS the variable becomes locked after all (external) events have been responded to.

We avoid this difference by putting the following constraint on activity diagrams.

C2 If the guard of some hyperedge $h$ refers to a variable that is updated in some activity $A$ and node A is not source of $h$, then hyperedge $h$ is not in parallel with node A, i.e., there does not exist a configuration $C$ such that $settobag(source(h) \cup \{\mathsf{A}\}) \sqsubseteq \mathsf{C}$. (act)

This constraint cannot be checked on the syntax of activity diagrams. Instead, it presupposes a semantics of activity diagrams. So we would have to check the constraint in both the RLS and ILS. But the purpose of this whole exercise is to avoid using both semantics, in particular the ILS semantics!

Fortunately, we can use the Petri net token-game semantics on hypergraphs, defined on page 75, to compute all the configurations that are possible in theory.

(Then the labels of the hyperedges are not interpreted.) Some of these configurations may not exist under the RLS or ILS. But every configuration existing in both RLS and ILS can be computed using the Petri net token-game semantics. If the check using the token-game semantics fails, the constraint may still hold under both RLS and ILS, because the violating configuration perhaps does not exist under these semantics. Thus, verification of the constraint using the token-game semantics is sufficient but not necessary.

Sometimes, by visual inspection of the syntax one can analyse whether or not the activity diagram satisfies the constraint. For example, in Figure 1.1 (Figure 3.6) it is immediately clear that the hyperedges leaving node WAIT-1 and WAIT-2 are only enabled once both activities *Check stock* and *Check customer* have completed. So, this activity diagram satisfies above constraint.

Finally, note that if the activity diagram satisfies this constraint, the guard of a relevant hyperedge can always be evaluated, because the variables it refers to are never locked.

**Internal events.** Since events are processed in parallel in the RLS, events generated by the system (internal events) may have a different effect under the RLS, compared to the ILS.

For example, consider the activity diagram in Figure 6.3. Suppose the current configuration is [WAIT-1,WAIT-3,WAIT-5] and events e and g happen simultaneously. Under the RLS, the reaction ends in configuration [WAIT-2,WAIT-4,WAIT-6]. Edge e1 is taken, because f is generated in the previous step, whereas edge e2 is not taken, because its trigger event h is already removed from the input when the system reaches [WAIT-6]. Under the ILS, however, it is possible that the reaction ends in configuration [WAIT-2,WAIT-4,WAIT-7], namely if e is processed before g and g before f. Then both edge e1 and edge e2 are taken.

We circumvent this difference in behaviour by putting the following constraint on the syntax, which says that events generated in some thread are expected in some other parallel thread. First some terminology: a hyperedge $h$ makes another hyperedge $h'$ (directly) relevant if $h$'s targets and $h'$'s sources overlap. A hyperedge $h$ makes another hyperedge $h'$ indirectly relevant, if $h$ makes another hyperedge

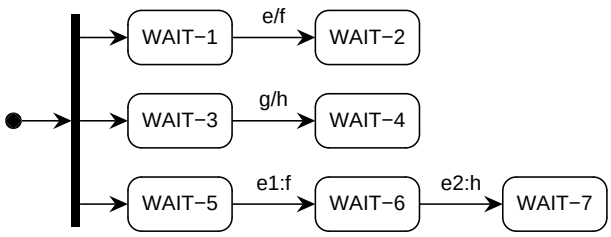



**Figure 6.3** *Example for issue "Internal events"*

$h''$ directly relevant, and $h''$ either makes $h'$ directly relevant or indirectly relevant.

C3 A hyperedge triggered by an internal event i is only made (directly) relevant by another hyperedge that also makes the hyperedge that generates i (in)directly relevant. (act)

The constraint is sufficient to rule out activity diagrams with different behaviour w.r.t. internal events. The constraint rules out the activity diagram in Figure 6.3: edge e2 violates the rule. It is made relevant by e1, but e1 does not make (in)directly relevant an edge that generates h.

**The in predicate.** If the in predicate is used, an activity diagram may behave differently under the RLS and stable ILS. For example, consider the activity diagram in Figure 6.4. Suppose that the current configuration is [A,WAIT-2] and $A$ terminates and e occurs simultaneously. Then under the RLS the next configuration is always [WAIT-1,WAIT-3]. When e occurs, the current configuration still contains A, so predicate in(A) is true. But under the stable ILS, it is possible that node A's termination event is processed before e. Then predicate in(A) is false when e is processed. Consequently, configuration [WAIT-1,WAIT-4] is reached. Thus, the RLS and stable ILS behave differently in this example.

We can rule out such activity diagrams by forbidding to use the in constraint.

C4(a) The in predicate is not used. (act)

We also formulate a weaker constraint, however, that will turn out to be useful for showing that it is possible in the stable ILS to mimic the behaviour of the RLS. The observation we make is that by choosing the right order of event processing, the stable ILS can behave similarly to the RLS even when the in predicate is used. For example in Figure 6.4 as described above, if e is processed before A's termination event, the RLS and ILS behave similarly. For simplicity, we assume that the in predicate is only used in guards of hyperedges triggered by an external event, i.e., not by a completion or internal event. If either $h$ or $h'$'s trigger event is

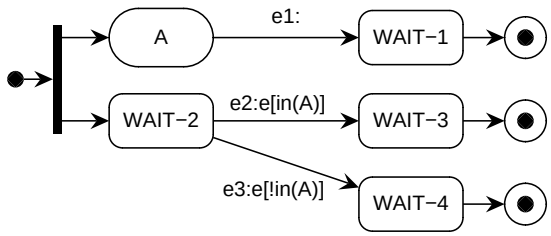



**Figure 6.4** *Example for issue "The* in *predicate"*

a completion or internal event, the ILS can no longer influence the order of event processing, because these events have priority of external events (see page 88).

We now formulate a constraint that guarantees that the right order can be chosen. A hyperedge $h$ must be taken before another hyperedge $h'$, written $h \prec h'$, iff there is a hyperedge triggered by $event(h)$ and with guard containing $\mathsf{in}(\mathsf{n})$, and there is a hyperedge triggered by $event(h')$ that either leaves $\mathsf{n}$ or enters $\mathsf{n}$. If $h \prec h'$, taking $h'$ may change the truth value of the guard of $h$. For example in Figure 6.4, taking $\mathsf{e1}$ the truth values of the guards of $\mathsf{e2}$ and $\mathsf{e3}$ change. So $\mathsf{e}_2 \prec \mathsf{e}_1$ and $\mathsf{e}_3 \prec \mathsf{e}_1$.

Now, if there is no cycle in the relation $\prec$, it is always possible to choose an appropriate order of event-processing in the stable ILS, namely by taking into account the relation $\prec$: if $h \prec h'$ then $event(h)$ should be processed before $event(h')$.

C4(b) The $\mathsf{in}$ predicate is only used in guards of hyperedges triggered by an external event. There is no cycle in the relation $\prec$. (act)

**Conflicting hyperedges.** We say that a hyperedge $h$ *conflicts* with another hyperedge $h'$ iff $h$ and $h'$ have some sources in common, i.e., $source(h) \cap source(h') \neq \emptyset$. Activity diagrams with conflicting hyperedges may have different behaviour under the RLS and stable ILS. We rule out such activity diagrams by defining constraints on conflicting hyperedges that every activity diagram must satisfy.

First, we define some terminology for hyperedges (cf. Table 3.1 on page 33).

- a *completion hyperedge* is a hyperedge that has no trigger event. A completion hyperedge is implicitly triggered by completion event under the ILS.

$$completion(h) \stackrel{\text{df}}{\Leftrightarrow} event(h) = \bot$$

- an *internal hyperedge* is a hyperedge that is triggered by an internally generated named event.

$$internal(h) \stackrel{\text{df}}{\Leftrightarrow} event(h) \in NamedInternalEvents$$

- an *external hyperedge* is a hyperedge that is triggered by an external event in the environment of the WFS. Every hyperedge that is not internal and not completion is external (see Table 3.1).

$$external(h) \stackrel{\text{df}}{\Leftrightarrow} \neg\, completion(h) \wedge \neg\, internal(h)$$

Table 6.1 defines the constraints on conflicting hyperedges and Table 6.2 shows what constraints apply to what hyperedges. We motivate each constraint by a small activity diagram that does not satisfy the constraint, shown in Figure 6.5.

Constraint C5 rules out the activity diagram in Figure 6.5(a). In the corresponding activity hypergraph, from node WAIT-2 two conflicting hyperedges leave

| | | |
|---|---|---|
| C5 | If two completion hyperedges are conflicting, they have the same sources. | (act) |
| C6 | A completion hyperedge is not conflicting with an internal hyperedge. | (act) |
| C7 | A completion hyperedge is not conflicting with an external hyperedge. | (act) |
| C8 | An internal hyperedge is not conflicting with another internal hyperedge. | (act) |
| C9 | An internal hyperedge is not conflicting with an external hyperedge. | (act) |

**Table 6.1** *Definition of constraints on conflicting hyperedges*

| | completion | internal | external |
|---|---|---|---|
| completion | C5 | C6 | C7 |
| internal | C6 | C8 | C9 |
| external | C7 | C9 | |

**Table 6.2** *Relation between constraints and hyperedges*

and also from node WAIT-4 two conflicting hyperedges leave. In Figure 6.5(a), if in configuration [WAIT-1,WAIT-3] events e and f occur simultaneously, then under the RLS it is possible that activity $B$ is started. Whereas under the stable ILS activity $B$ is never started and activity $A$ is always started. Under the stable ILS, if e is processed before f, then first node WAIT-1 is left and node WAIT-2 is entered. A completion event for WAIT-2 is generated. Since completion events have priority over non-completion events, the enabled edge leaving WAIT-2 and entering WAIT-5 is taken and node WAIT-5 is entered. After event f has been processed and all completion events, finally configuration [A] is reached. By similar reasoning, it can be shown that if event f is processed before e, also configuration [A] is reached.

Constraint C6 rules out the activity diagram in Figure 6.5(b). In Figure 6.5(b), suppose the current configuration is [A,WAIT-2] and that activity node A terminates. Under both the RLS and ILS, a step is taken and the next configuration is [WAIT-1,WAIT-2], and internal event done has been generated. Then under the RLS two steps are possible, one in which the next configuration is [WAIT-4], and another one in which the next configuration is [WAIT-1,WAIT-3]. Whereas under the ILS, only one step is possible, namely the one in which configuration [WAIT-4] is reached, because completion events have priority over non-completion events.

Constraint C7 rules out the activity diagram in Figure 6.5(c). In Figure 6.5(c), suppose the current configuration is [WAIT-1,WAIT-4] and event e and f occur at the same time. Then under the RLS, the next configuration is [WAIT-2,WAIT-5]. The corresponding state is unstable, and next configuration [WAIT-3] is reached.

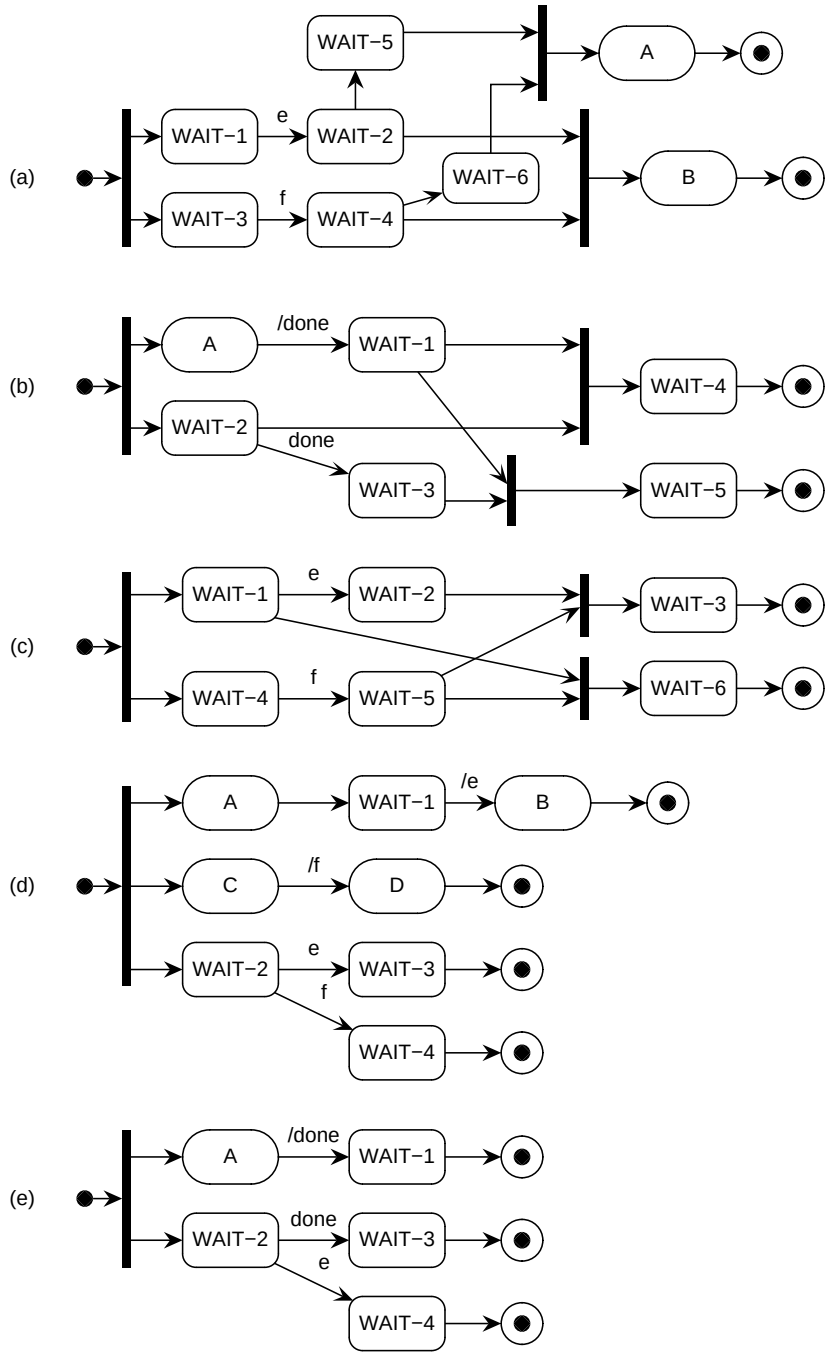



**Figure 6.5** *Motivating examples for the constraints in Table 6.1*

Under the stable ILS, either e or f is processed first. If f is processed before e, configuration [WAIT-1,WAIT-5] is reached. Then a completion event is generated and configuration [WAIT-6] is reached. So, the behaviour of the activity diagram under the ILS is different from its behaviour under the RLS.

Constraint C8 rules out the activity diagram in Figure 6.5(d). In Figure 6.5(d), suppose the current configuration is [A,C,WAIT-2] and activity nodes A and C terminate at the same time. Then under the RLS, internal event f is generated before e is generated. So node WAIT-4 is entered under the RLS. Under the stable ILS, if the termination event of A is processed before the termination event of C, then internal event e is processed before f, since completion events (in this case the completion event of WAIT-1) have priority over non-completion events. So under the stable ILS, it is possible that node WAIT-3 is entered, rather than WAIT-4.

Constraint C9 rules out the activity diagram in Figure 6.5(e). In Figure 6.5(e), suppose the current configuration is [A,WAIT-2] and activity node A terminates at the same time as e occurs. Under the RLS, the next configuration is always [WAIT-1,WAIT-4]. Under the stable ILS, suppose activity node A's termination event is processed before e. On page 88 we explain that internal events must have priority over external events in the (stable) ILS. So, under the stable ILS the next configuration can be [WAIT-1,WAIT-3]. This configuration is unreachable under the RLS in this case.

**Duplicate conflicts.** In an activity diagram, the same conflict between some hyperedges may have be to be resolved twice at the same moment. Then there are two groups of inconsistent, i.e. conflicting, hyperedges and the hyperedges in each group have similar trigger events. Figure 6.6 shows an example. Formulated more precisely, an activity diagram has a *duplicate conflict*, if there are two different groups $G_1, G_2$ of hyperedges such that:

- all the hyperedges in the two groups can be enabled at the same time;
- hyperedges within a group are inconsistent (conflicting);
- hyperedges between the groups are consistent (non-conflicting);
- the enabling trigger events and guard conditions of the hyperedges in $G_1$ also enable the hyperedges in $G_2$.

For example, the activity diagram in Figure 6.6 has duplicate conflicts, since nodes WAIT-1 and WAIT-4 are in parallel and the two hyperedges with trigger e (f) belong to different groups (group {e1,e2} and group {e3,e4}).

The behaviour of activity diagrams that have duplicate conflicts is different under the RLS and ILS due to difference between parallel (RLS) and single-event processing (ILS). For example, in Figure 6.6, suppose in configuration [WAIT-1,WAIT-4] events e and f occur at the same time. Under the RLS, a possible step

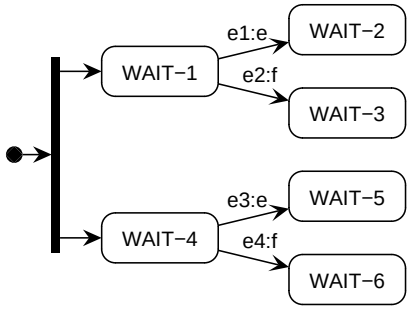



**Figure 6.6** *An example of a duplicate conflict*

is [e1,e4]. This step is impossible under the ILS, due to the single-event processing. If e is processed first, the only possible step is [e1,e3]. If f is processed first, the only possible step is [e2,e4]. Note however that both these steps are also possible under the RLS.

Assuming absence of duplicate conflicts in an activity diagram, we prove that single-event processing does not impact the computation of steps: the union of the steps that are computed for each event separately (ILS), equals a step that is computed when all events are considered at the same time in parallel (RLS).

**Theorem 6.1** *Assume an activity diagram does not have duplicate conflicts. Take an arbitrary configuration $C$ and input $I = [\, e_1, .., e_n \,]$.*

*Let $S$ be a step such that $isStep(C, I, S)$. For each input event $e \sqsubseteq I$, step $S_e$ is defined as $S_e = S \cap Enabled(C, [\, e \,])$.*

*Then $S = \bigcup_{e \sqsubseteq I} S_e$, and for all $e \sqsubseteq I$, $isStep(C, [\, e \,], S_e)$ holds.*

*Proof.* Straightforward checking of definition of step. □

If an activity diagram did have duplicate conflicts, we would not be able to prove such a theorem. We gave a counterexample above that we explain now in more detail. In Figure 6.6 if $C = [\mathsf{WAIT\text{-}1}, \mathsf{WAIT\text{-}4}]$ and $I = [\, \mathsf{e}, \mathsf{f} \,]$ then a possible step $S = [\mathsf{e1}, \mathsf{e4}]$. Denote by $S_{\mathsf{e}} = S \cap Enabled(C, [\, \mathsf{e} \,]) = [\mathsf{e1}]$. But step $S_{\mathsf{e}}$ does not satisfy $isStep([h_1], C, S_{\mathsf{e}})$, because e3 is not part of $S_{\mathsf{e}}$.

C10 The activity diagram does not have duplicate conflicts. (act)

**Events have extra effects.** Due to single-event processing, in the (stable) ILS events can have unexpected extra effects. Consider for example the activity diagram in Figure 6.7. Suppose in configuration [A] that activity node A terminates and event e occurs simultaneously. Then under the ILS, it is possible that configuration [WAIT-2] is reached, namely if A's termination event is processed before e.

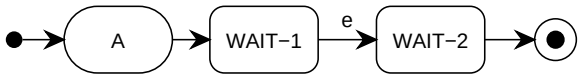



**Figure 6.7** *Event* e *has extra effect*

But under the RLS, node WAIT-2 is never reached in this particular case. So under the ILS event e has an extra effect. We conclude that although under the ILS the same hyperedges as under the RLS are taken in response to some event occurrences, under the ILS sometimes some *extra* hyperedges are taken.

To take these extra hyperedges under the RLS, some events must be regenerated. This puts a constraint upon the environment: it should be cooperative, i.e., regenerate some external events if needed. Note that an activity termination event does not have this problem, as such an event always triggers an edge that is already relevant when the event occurs.

C11 The environment is cooperative. (env)

Extra effects of an event cannot be avoided by imposing constraints upon activity diagrams. Such constraints would rule out almost every activity diagram that has a named event in it; for example the one in Figure 6.7.

Constraint C11 shows the inequivalence of RLS and ILS. We will only use it to prove one theorem (Theorem 6.2(ii)), and then drop it. Then we will prove a weaker theorem (Theorem 6.7) for which we do not need to use this constraint.

### 6.1.2 Stable ILS and ILS

**Atomic reaction.** In the RLS, a reaction is atomic: the complete reaction to an event, i.e., the chain of steps, is immediately computed and executed. In particular, next event occurrences cannot interfere with the current reaction. This is simulated under the stable ILS by the constraint that the next events occur only when the current reaction has completed. By contrast, in the ILS, a reaction does not have to be atomic: next event occurrences can interfere with the current reaction.

Consider for example the activity diagram in Figure 6.8. Suppose the system is in configuration [A,WAIT-4,WAIT-7] and activity node A terminates and subsequently event g occurs. Under the RLS and stable ILS, the first reaction ends in stable configuration [WAIT-2,WAIT-5,WAIT-7]. (Under the stable ILS, we know that g can only occur once the previous reaction is finished.) If subsequently g occurs, configuration [WAIT-3,WAIT-6,WAIT-8] is entered. Under the ILS, however, g may occur *while* the system is busy reacting to A's termination event. For example, g may occur before f has been generated. Then, assuming a FIFO order on events, g is processed before f, and consequently after all events, including f,

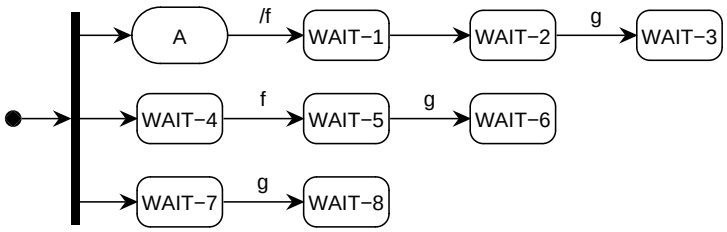



**Figure 6.8** *Example for issue "Atomic reaction"*

have been processed, configuration [WAIT-2,WAIT-5,WAIT-8] is entered. So then nodes WAIT-3 and WAIT-6 are not entered under the ILS, whereas they are entered under the stable ILS. Note that even though the completion event of WAIT-4 may be generated *after* event g occurs, the completion event is always processed *before* g, due to the priority of completion events over non-completion events.

We conclude that the example activity diagram behaves differently under the RLS and stable ILS. The difference occurs because under the ILS g is processed before the reaction to A's termination event has been completed.

The easiest way to solve the problem sketched above is to assume that internal events have priority over external events, just like completion events have priority over non-completion events. Then the reaction to an external event, which includes generating internal events and reacting to these events, is completed before the next external event is processed. A constraint on activity diagrams would rule out almost every activity diagram. A constraint on the environment would also solve the problem, but would unnecessarily restrict the environment in its behaviour.

C12 In the ILS, internal events have priority over external events. (sem)

Our implementation-level semantics, given in Chapter 5, can be easily changed to satisfy this property, but we will not do this here.

**Locking of variables.** On page 79 we showed that locking of variables may lead to different behaviour under RLS and stable ILS. Now we show that it also may lead to a different behaviour under stable ILS and ILS.

Under the RLS and stable ILS, if the trigger event of a hyperedge $h$ occurs, $h$ is not taken if one of the variables that $h$'s guard refers to is locked. Then $h$'s guard cannot be evaluated. In the ILS, however, $h$'s trigger event may be processed at a later time, when the activity that prevented $h$'s guard from being evaluated has already terminated. Then $h$'s guard can be evaluated and $h$ can be taken, even though at the time the trigger event occurs, $h$'s guard could not be evaluated. This is impossible under the stable ILS, because in that semantics the assumption

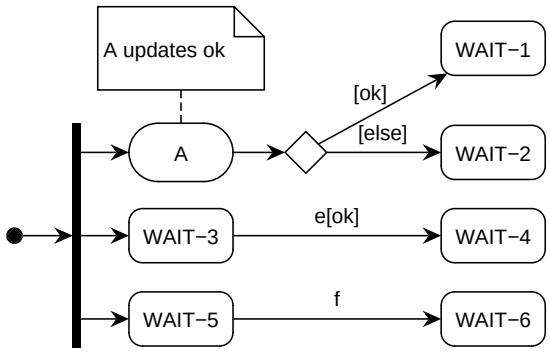



**Figure 6.9** *Example for issue "Locking of variables"*

is made that events, including termination events, only occur when the system has completed its previous reaction.

For example, in Figure 6.9, suppose e and f happen at the same time while the system is in configuration [A,WAIT-3,WAIT-5]. Then under the stable ILS, the system reaches configuration [A,WAIT-3,WAIT-6]. Node WAIT-4 is not entered because variable ok is updated in activity $A$. Under the ILS, suppose f is processed before e. While f is processed, activity $A$ can terminate and ok can become true. If next e is processed, node WAIT-4 is entered. So for this activity diagram its behaviour under the RLS is different from its behaviour under the stable ILS and RLS.

On page 80 we already defined a constraint on activity diagrams that rules out activity diagrams exhibiting this difference in behaviour in RLS and stable ILS.

**Overflowing of ILS queue.** So far, we have not put any constraints upon the queue. In theory, it is possible that in the ILS the queue gets overflowed, either because of its limited capacity or because the Router is too slow to keep in pace with changes in the environment. We here assume that the queue does not get overflowed.

C13 The queue in the ILS does not get overflowed. (env)

This is on the one hand a constraint on the environment: no infinitely many events occur in a finite time interval. On the other hand, it is also a constraint upon the WFS itself: the Router must not be too slow with respect to the pace of change of the environment. Thus, together the environment and WFS must be able to cooperate such that the environment is not too fast for the WFS and the WFS is not too slow for the environment. This resembles the issue of flow control in computer networks [149].

**FIFO queue.** To ensure that an activity diagram behaves the same under the ILS and stable ILS (and RLS), the queue in the ILS needs to have a FIFO ordering. If no FIFO ordering is used, under the ILS an event that happens after some other event $e$, can be processed before $e$. Such behaviour is impossible under the RLS and stable ILS.

C14 The queue in the ILS uses a FIFO (First-In-First-Out) ordering. (env)

**Summary of constraints.** Table 6.3 summarises the constraints defined in this section. Constraints C15, C16 and C17 are treated in Sections 6.2.3 and 6.2.4.

| | | |
|---|---|---|
| C1 | The environment does not update case attributes (control data). | (env) |
| C2 | If the guard of some hyperedge $\mathsf{h}$ refers to a variable that is updated in some activity $A$ and node $\mathsf{A}$ is not source of $h$, then hyperedge $\mathsf{h}$ is not in parallel with node $\mathsf{A}$, i.e., there does not exist a configuration $C$ such that $\mathit{settobag}(\mathit{source}(h) \cup \{\mathsf{A}\}) \sqsubseteq \mathsf{C}$. | (act) |
| C3 | A hyperedge triggered by an internal event $\mathsf{i}$ is only made (directly) relevant by another hyperedge that also makes the hyperedge that generates $\mathsf{i}$ (in)directly relevant. | (act) |
| C4(a) | The $\mathsf{in}$ predicate is not used. | (act) |
| C4(b) | The $\mathsf{in}$ predicate is only used in guards of hyperedges triggered by an external event. There is no cycle in the relation $\prec$. | (act) |
| C5 | If two completion hyperedges are conflicting, they have the same sources. | (act) |
| C6 | A completion hyperedge is not conflicting with an internal hyperedge. | (act) |
| C7 | A completion hyperedge is not conflicting with an external hyperedge. | (act) |
| C8 | An internal hyperedge is not conflicting with another internal hyperedge. | (act) |
| C9 | An internal hyperedge is not conflicting with an external hyperedge. | (act) |
| C10 | The activity diagram does not have duplicate conflicts. | (act) |
| C11 | The environment is cooperative. | (env) |
| C12 | In the ILS, internal events have priority over external events. | (sem) |
| C13 | The queue in the ILS does not get overflowed. | (env) |
| C14 | The queue in the ILS uses a FIFO (First-In-First-Out) ordering. | (sem) |
| C15 | An external broadcast event triggers at most one hyperedge. | (act) |
| C16 | A wait hyperedge does not have overlapping sources and targets. | (act) |
| C17 | An external event does not imply another external event. | (act) |

**Table 6.3** *Summary of constraints*

## 6.2 Similarities between the two semantics

### 6.2.1 RLS and stable ILS

The following theorem relates the requirements-level and stable implementation-level semantics. The theorem states that the outcome of the system reaction, i.e. the hyperedges taken in response to some events, is the same under both semantics. The used constraints have been identified and motivated in Section 6.1.1.

**Theorem 6.2** *Given Constraints C1–C3 and C5–C10 (Table 6.3).*

*Given a certain stable state with configuration $C$. Suppose a bag $E$ of input events occurs.*

*(i) Under Constraint C4(b), if under the RLS the system takes a superstep in response to $E$, then under the stable ILS, the system can take every hyperedge in the superstep in response to $E$, but not necessarily in the same order.*

*(ii) Under Constraint C4(a) and Constraint C11, if under the stable ILS the system takes a sequence of steps in response to $E$, then under the RLS the system can take every hyperedge that is part of one these steps in response to $E$, but not necessarily in the same order.*

*Proof.* We prove both claims by induction on the sequence of steps taken under the RLS and stable ILS. Let the sequence of steps in the superstep of the RLS be $S_1, S_2, .., S_n, ..$. The stable ILS takes a sequence of steps in response to $E$. Each step is taken when processing some event $e$ in the queue, which can be external, internal, or completion. Denote this sequence by $S_{e_1}, S_{e_2}, .., S_{e_m}, ..$, where $S_{e_m}$ is a step that is taken in response to event $e_m$.

We first make some general observations. Bag $E$ only contains external events. When taking steps, internal events (RLS and ILS) and completion events (only in ILS) are generated. (Remember the RLS does not have completion events.)

(i) We prove the claim by induction on the sequence of steps taken under the RLS.

**Basic case:** We prove that every hyperedge $h$ in $S_1$ can also be taken in some step $S_{e_j}$ under the stable ILS.

By definition of a superstep in the RLS, $S_1$ only contains external hyperedges. So $h$ must be external.

Let $e_1$ be the first event in $E$ that is processed under the ILS. Denote by $S_{e_1}$ the step that is taken under the ILS in response to $e_1$. By Constraint C10 and Theorem 6.1, we have that step $S_{e_1} = S_1 \cap \mathit{Enabled}(C, [\, e_1 \,])$ is a step. If $h$ is in $S_{e_1}$ we are done. If $h$ is not in $S_{e_1}$, $h$ can be taken in a later step under the ILS. We show this in two parts: $h$ can stay relevant, and $h$ can become enabled.

- $h$ can stay relevant.

  $h$ can only be made irrelevant by a conflicting hyperedge $h_c$ that is enabled and taken. By C7 and C9, $h_c$ cannot be a completion or internal hyperedge. So $h_c$ is external. Then $h_c$ cannot be part of $S_1$, because $S_1$ consists of non-conflicting hyperedges (by definition of step). If $h_c$ is enabled in the first state and not in $S_1$, the triggers of $h$ and $h_c$ are both in $E$ and thus have equal priority. Thus $h$ can stay relevant.

- $h$ can become enabled.

  Following the definition of enabled, we show that $h$'s guard can become true, and that $h$ can become triggered.

  - $h$'s guard can be true.

    Since $h$ is taken in $S_1$, $h$'s guard must be true under the RLS just before $h$ is taken. $h$'s guard was either true in the begin state or became true because some variables changed value. Suppose that the guard does not contain an in predicate (we deal with in predicates below). By Constraint C1, such a guard only changes valuation when an activity terminates. So, termination events in $E$ can cause a change in valuation of $h$'s guard condition but the environment cannot. The current configuration $C$ is non-interfering, so every variable is updated by at most one activity at a time. Under both RLS and ILS, the new valuation of the local variables is immediately retrieved once the activities terminate (see definition of transition $\rightarrow_{retrieve\_lvar}$ in both semantics). Thus, variables contained in $h$'s guard have the same value under RLS and (stable) ILS. The only difficulty is if some variables in $h$'s guard become locked under the stable ILS by some newly started activity $a$, i.e., $a$ is started in response to $E$. Then $h$ can no longer be taken under the stable ILS. By Constraint C2 we know that such an activity $a$ cannot be in parallel with $h$'s source nodes. So $a$ is not active. So variables in $h$'s guard cannot become locked under the stable ILS.

    If $h$'s guard contains an in predicate, the valuation of that predicate does not change for the following reason. By Constraint C4(b), there is no cycle in relation $\prec$. If there is no cycle, events can be processed under the stable ILS in an order that is consistent with $\prec$. Therefore, the valuation of the in predicate in the guard of hyperedge $h$ does not change to false or true before $h$ is taken.

  - $h$ can become triggered.

    The trigger event of $h$ is processed at a later time than $e_1$. By definition, under the stable ILS new external events only occur after the current reaction to $E$ has completed. Thus a hyperedge that conflicts with $h$ and whose trigger event is not in $E$, cannot be triggered before $h$.

**induction case:** Step $S_n$.

We prove that every hyperedge $h$ in step $S_n$, taken under the RLS, can also be taken in some step $S_{e_j}$ under the stable ILS. By the induction hypothesis all hyperedges in previous steps $S_1, .., S_{n-1}$ can be taken under the stable ILS as well (but not necessarily in the same order). Since the current state is unstable (by definition of superstep), step $S_n$ can only contain completion or internal hyperedges (or both).

- $h$ is a completion hyperedge. We show that every completion hyperedge $h$ that is taken in $S_n$ can be taken in some step $S_{e_j}$ of the stable ILS.
  - $h$ can become relevant.

    By the induction hypothesis all hyperedges in previous steps $S_1, .., S_{n-1}$ can be taken (but not necessarily in the same order). So by the induction hypothesis $h$'s source nodes become active, but not necessarily all at the same time. Constraints C5, C6, and C7 rule out that one of $h$'s source nodes is left while $h$ is irrelevant. By Constraints C6 and C7, we know that if a hyperedge $h_c$ conflicts with $h$, then $h_c$ must be a completion hyperedge. So $h$'s source nodes can be left by taking $h$ or $h_c$. By Constraint C5, we know that $h$ and $h_c$ have the same sources. So $h$'s sources can only be left when $h$ has become relevant. Thus, $h$ can become relevant under the stable ILS.
  - $h$ can become enabled.
    - $h$'s guard can be true.

      Since $h$ is taken under the RLS, $h$'s guard is true under the RLS just before $h$ is taken. By definition, in the RLS $h$'s guard only changes value in the beginning of the superstep. Some terminated activities might have changed some variables in $h$'s guard. In both RLS and ILS, the new valuation of the variables is immediately retrieved once these activities terminate (see definition of transition $\rightarrow_{retrieve\_lvar}$ in both semantics). So $h$'s guard can also be true in the begin state of the stable ILS, once the new valuations of local variables have been retrieved. We now show that if $h$'s guard is true, it stay true. Since no activity is updating variables contained in $h$'s guard condition (because the current configuration $C$ is non-interfering), $h$'s guard cannot change value by some activity. By Constraint C1 the environment cannot change $h$'s guard. Moreover, taking hyperedges under the stable ILS does not impact the truth value of $h$'s guard, since by Constraint C4(b) $h$'s guard does not contain an in predicate. So $h$'s guard stays true under the stable ILS.

      The only difficulty is if some variables in $h$'s guard become locked in one of the steps of the stable ILS by some newly started activity

$a$, i.e., $a$ is started in response to $E$. If variables contained in $h$'s guard are locked, $h$ cannot be taken. By Constraint C2 we know that such an activity $a$ cannot be in parallel with $h$'s source nodes. So variables in $h$'s guard cannot become locked under the stable ILS.

  - ∗ $h$ can become triggered.
    Every time a source node of $h$ is entered, a completion event is generated under the stable ILS. When the last source node is entered, the generated completion event can trigger $h$.

- internal hyperedge. Every internal hyperedge $h$ in $S_n$, taken under the RLS, can be taken in some step $S_{e_j}$ of the stable ILS.

  - $h$ can become relevant.
    By similar reasoning as in the previous item [completion hyperedge]. By the induction hypothesis, all hyperedges in previous steps are taken. By Constraints C6, C8, and C9, $h$ does not have any conflicting hyperedges.
  - $h$ can become enabled.
    - ∗ $h$'s guard can be true.
      By similar reasoning as in the previous item [completion hyperedge], it can be shown that $h$'s guard is true in both RLS and stable ILS.
    - ∗ $h$ can become triggered.
      By definition of the RLS, some hyperedge in step $S_{n-1}$ triggers $h$. By the induction hypothesis, under the stable ILS this hyperedge can be taken as well and therefore $h$'s trigger event is generated and put in the queue. By Constraint C3 the hyperedge is already relevant when the trigger event is generated. So the trigger event cannot be generated too early.

(ii) We prove the claim by induction on the sequence of steps taken under the stable ILS.

**Basic case:** Step $S_{e_1}$. Using similar reasoning as in (i), but Constraint C10 is not needed now.

**induction case:** Step $S_{e_m}$. We prove that every hyperedge $h$ in step $S_{e_m}$, taken under the stable ILS, can also be taken in some step $S_i$ of the RLS. By the induction hypothesis all hyperedges in previous steps $S_{e_1}, .., S_{e_{m-1}}$ are taken in some step $S_x$ of the RLS (but possibly $x > i$).

- $h$ is a completion hyperedge. Using similar reasoning as in (i)(induction case).
- $h$ is an internal hyperedge.

- $h$ can become relevant.
  By similar reasoning as in (i)(induction case).
- $h$ can become enabled.
  - $h$'s guard can be true.
    By similar reasoning as in (i)(induction case).
  - $h$ can become triggered.
    $h$'s trigger event is generated by some hyperedge in a previous step $S_{e_l}$, where $0 < l < m$. By the induction hypothesis, that other hyperedge can be taken under the RLS as well. Moreover, by Constraint C3 the trigger event is only generated when $h$ is already relevant. So the trigger event cannot be generated too early.

• external hyperedge. Every external hyperedge that is taken in $S_{e_m}$ can be taken in some step of the RLS.

  There are two cases. (1) $h$ is relevant in the first state, or (2) $h$ is made relevant later on because some hyperedges have been taken in previous steps (see page 87).

  - $h$ can become relevant.
    Case (1): since RLS and stable ILS have the same initial state, $h$ is also relevant in the first state of the stable ILS. Case (2) follows from the induction hypothesis: the hyperedges making $h$ relevant can also be taken under the RLS, and Constraints C7 and C9.
  - $h$ can become enabled.
    - $h$'s guard can be true.
      Similar reasoning as in (i)(basic case), using Constraints C1, C2 and C4(b) (so $h$'s guard does not contain an in predicate).
    - $h$ can be triggered.
      Case (1): $h$'s trigger is already in the start queue of the RLS. If $h$ is relevant in $C$, the event can be picked from the queue.
      Case (2): If $h$ becomes relevant later on, then by Constraint C11, the environment can regenerate the trigger event of $h$, and $h$ can be taken under the RLS.

□

The following corollary states that an activity diagram can diverge under the RLS iff it can diverge under the ILS. The corollary follows immediately from above theorem.

**Corollary 6.3** *The system can diverge under the RLS iff the system can diverge under the ILS.*

### 6.2.2 Stable ILS and ILS

The following theorem relates the behaviour of an activity diagram under the stable implementation-level semantics with its behaviour under the implementation-level semantics. The used constraints have been identified and motivated in Section 6.1.2.

**Theorem 6.4** *Given Constraints C2 and C12–C14 in Table 6.3.*

*Given a stable ILS state in which some bag of events $E_1$ occur. By definition, a stable ILS state is also an ILS state.*

*No matter whether a subsequent bag of events $E_2$ happens after or during the system reaction to the events in $E_1$ under the ILS, under both semantics, ILS and stable ILS, the same steps can be taken and thus the same end state can be reached.*

*Proof.* By Constraint C14, events in $E_1$ are processed before events in $E_2$. Moreover, in both semantics completion events have priority over other events (by definition) and internal events have priority over external events (by Constraint C12). Thus, events in $E_2$ cannot interfere with the processing of events in $E_1$ even if they occur while the system is busy responding to events $E_1$. So, the stable ILS and ILS can process all events in $E_1$ and $E_2$, including internal and completion events, in the same order, regardless of whether in the ILS external events happen while the system is busy reacting to events in $E_1$ or after the system has completed its reaction to $E_1$. By Constraint C13 every event in $E_2$ is eventually responded to.

By Constraint C2, while $h$ is relevant, there are no activities running in parallel that update some variables $h$'s guard refers to. In other words, $h$'s guard does not become locked. So newly started activities cannot prevent $h$ from being taken. □

From this theorem, it follows that for analysis purposes we can restrict ourselves to the stable ILS rather than to the ILS.

### 6.2.3 A stronger result

Above, we have shown that every hyperedge taken under the RLS in a reaction to some bag $E$ of event occurrences, can also be taken under the ILS in a reaction to $E$. Unfortunately, because external events can have extra effects as discussed on page 87, this does not mean that the same end configuration is reached in RLS and stable ILS. Nevertheless, we now proceed to strengthen Theorem 6.2(i) by stating that, in addition to taking the same hyperedges, the stable ILS should reach the same end configuration. We enforce this by defining an extra constraint on activity diagrams.

First, we look again at the example in Figure 6.7 on page 88 that we discussed earlier. In this example, it is possible under the ILS to reach a similar configuration

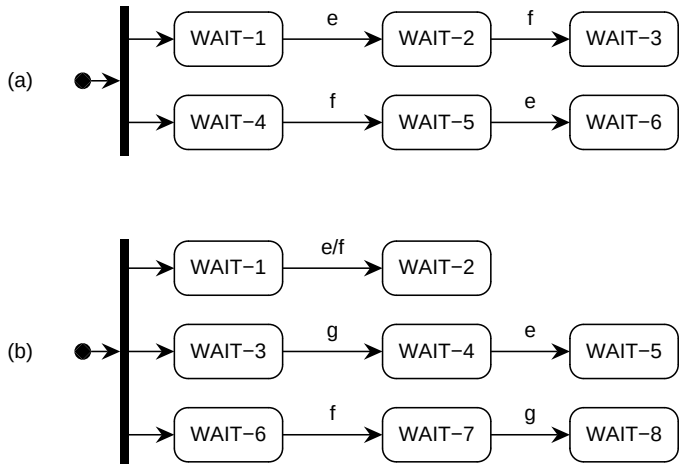



**Figure 6.10** *Two examples in which events have unavoidable extra effects*

as under the RLS: If e is processed before the termination event of $A$ under the ILS, then the same configuration as under the RLS, [WAIT-1], is reached.

Unfortunately, there exist activity diagrams whose behaviour under the RLS cannot be simulated under the ILS, because no matter what order of event processing is chosen, events will have an extra effect under the ILS that they do not have under the RLS. Figure 6.10 shows two examples. In Figure 6.10(a), suppose the system is in configuration [WAIT-1,WAIT-4] and events e and f occur simultaneously. Then under the RLS configuration [WAIT-2,WAIT-5] is reached. This configuration is unreachable under the ILS, because under the ILS one event is processed at a time. For example, if e is processed before f, first configuration [WAIT-2,WAIT-4] is reached and then configuration [WAIT-3,WAIT-5].

Figure 6.10(b) shows a more complex example. Suppose the system is in configuration [WAIT-1,WAIT-3,WAIT-6] and events e and g occur simultaneously. Under the RLS, the system reaction stops in configuration [WAIT-2,WAIT-4,WAIT-7]. Under the ILS, the system reaction stops either in configuration [WAIT-2,WAIT-5,WAIT-7] or in configuration [WAIT-2,WAIT-4,WAIT-8]. Under the ILS, configuration [WAIT-2,WAIT-4,WAIT-7] is unreachable as end configuration of the system reaction.

We observe from the examples in Figure 6.10 that the problem of unavoidable extra effects is only due to external broadcast events that trigger more than one hyperedge. Therefore, a sufficient (but not necessary) constraint to rule out such activity diagrams is by forbidding that two or more different hyperedges are triggered by the same external broadcast event.

C15 An external broadcast event triggers at most one hyperedge. (act)

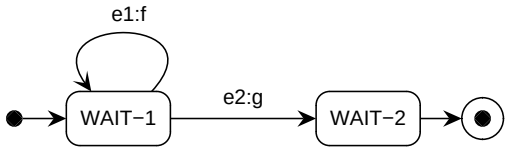



**Figure 6.11** *Another example in which an event has an unavoidable extra effect*

This constraint rules out the two activity diagrams in Figure 6.10. It may, however, also rule out activity diagrams in which external broadcast events do not have extra effects. Thus, the constraint is sufficient but not necessary. Above constraint was not too restrictive for our case studies. A less restrictive constraint would have been considerably more complex to formulate.

Another constraint is needed to rule out the activity diagram shown in Figure 6.11. In the example, if the current configuration is [WAIT-1] and events f and g occur at the same time, then under the RLS there are two possible next configurations: [WAIT-1] (by taking e1) and [WAIT-2] (by taking e2). Under the stable ILS, configuration [WAIT-1] cannot be reached; instead, always configuration [WAIT-2] is reached. Thus, under the stable ILS event g has an unavoidable extra effect. To rule out such activity diagrams, we require that every wait hyperedge that is triggered by an external broadcast event does not make itself relevant.

C16 A wait hyperedge does not have overlapping sources and targets. (act)

Using these two constraints, Theorem 6.2(i) can be strengthened as follows: given a configuration $C$ and $I$, it is possible under the stable ILS to take the same hyperedges *and* reach a similar end configuration as under the RLS.

**Theorem 6.5** *Given Constraints C1–C3, C4(b), C5–C10, C15 and C16 (Table 6.3).*

*If under the RLS the system in configuration $C$ takes a superstep in response to input events $E$ and reaches configuration $C'$, then under the stable ILS the system can reach $C'$ from $C$ as well in response to $E$ by taking every hyperedge in the superstep.*

*Proof.* By Theorem 6.2(i) we have that every hyperedge that is taken in the superstep under the RLS, can also be taken under the stable ILS. We now show that by choosing an appropriate order of event processing under the stable ILS, it can be avoided that external broadcast events have extra effects. Thus, if under the stable ILS events are processed under this order, then every hyperedge that is taken under the stable ILS is also taken under the RLS. We define the order as follows. Denote the bag of events that trigger relevant hyperedges in $C$ by $E'$. An appropriate processing order is to first process events that are in $E$ but not

in $E'$, and then process events that are in $E'$. Events in $E$ but not in $E'$ do not trigger any hyperedges. By processing an event in $E'$, a hyperedge is taken and some hyperedge $h$ may become relevant. By Constraint C16, $h$ was not relevant in $C$. By Constraint C15, $h$ is not triggered by any of the events in $E'$. Hence, $h$ is not taken under the stable ILS in response to $E$. □

The following corollary, which states that every configuration in a stable state under the RLS is also a reachable configuration in a stable state under the stable ILS, now follows immediately.

**Corollary 6.6** *Given Constraints C1–C3, C4(b), C5–C10, C15 and C16 (Table 6.3).*

*Each configuration that is reachable in some stable state under the RLS is reachable in some stable state under the stable ILS.*

### 6.2.4 Stable simulation relations

We now define a simulation relation between the CTSs induced by the RLS and ILS. We will use this relation in Chapter 10 to identify the functional requirements that are shared by both semantics.

We will require that both RLS and ILS reach the same stable configurations. Thus we will use Theorem 6.5, which is a strengthened version of Theorem 6.2(i). Theorem 6.5 says that the stable ILS can simulate the complete reaction of the RLS to some events $E$, i.e., under the stable ILS the same end configuration can be reached as under the RLS. Theorem 6.2(ii) does not show that the RLS can simulate the stable ILS, because under the stable ILS events can have extra effects, requiring a constraint (C11) upon the environment to regenerate some events.

However, below we will prove a weakened version of Theorem 6.2(ii) by not referring to some specific bag $E$ of input events anymore. We will prove that if the ILS can reach from the current stable state another stable state, the RLS can reach that other stable state too. This implies that the RLS can reach the same end configuration as the ILS. In the weakened theorem, we do not use Constraint C11 on the environment anymore.

In our proof of the weakened theorem, each external event processed by the Router in the ILS corresponds to an external event occurrence in the RLS. This correspondence only holds if no two external events occur by definition at the same time. So an external event should not imply another external event. One event $e$ implies another event $e'$ iff the occurrence of $e$ implies the occurrence of $e'$. For example, event when(April 1) implies when(0:00h), because when the current date becomes April 1, it is also 0:00 hours. So if when(April 1) occurs, when(0:00h) also occurs.

We rule out activity diagrams that have events that occur by definition simultaneously, by defining the following constraint on the syntax of activity diagrams.

C17 An external event does not imply another external event. (act)

Note that by definition named events cannot imply other events.

Before we give our theorem, let us define some terminology and auxiliary definitions. Given some activity hypergraph $AH$, let $CTS^{\mathsf{RLS}}$ denote the CTS induced by $AH$ under the RLS, and let $CTS^{\mathsf{ILS}}$ denote the CTS induced by $AH$ under the ILS.

We define a relation $R \subseteq \Sigma(Var^{\mathsf{RLS}}) \times \Sigma(Var^{\mathsf{ILS}})$ on the (stable) valuations of these two CTSs as follows.

$$\begin{aligned} R \stackrel{\text{df}}{=} \{\, (\sigma^{\mathsf{RLS}}, \sigma^{\mathsf{ILS}}) \mid\ & \sigma^{\mathsf{RLS}} \models \mathit{stable} \\ & \wedge\ \sigma^{\mathsf{ILS}} \models \mathit{stable} \\ & \wedge\ \sigma^{\mathsf{RLS}}(C) = \sigma^{\mathsf{ILS}}(C) \\ & \wedge\ \forall\, v \in LVar \bullet \sigma^{\mathsf{RLS}}(v) = \sigma^{\mathsf{ILS}}(v) \\ & \wedge\ \sigma^{\mathsf{RLS}}(MC) = \sigma^{\mathsf{ILS}}(MC) \\ & \wedge\ \forall\, t \in RT \bullet \sigma^{\mathsf{RLS}}(t) = \sigma^{\mathsf{ILS}}(t) \\ \} & \end{aligned}$$

Predicate *stable* is defined for a valuation $\sigma$ of a ILS system as follows:[2]

$$\sigma \models \mathit{stable} \quad \stackrel{\text{df}}{\Leftrightarrow} \quad \sigma(Q) = \langle\rangle \ \wedge\ \sigma(Q_{comp}) = \langle\rangle \ \wedge\ \sigma(Q_{int}) = \langle\rangle \ \wedge\ \sigma(re) = \bot$$

Note the similarity with the definition of *stable* in the RLS semantics (page 65).

We call $R$ an *stable simulation relation*. With simulation we mean that each CTS can mimic (sequences of) transitions of the other. We now proceed to define this more formally.

First we define an abbreviation for a sequence of transitions between stable valuations. All intermediary valuations in the sequence are unstable. Given two stable valuations $\sigma$, $\sigma'$, that is $\sigma \models \mathit{stable}$ and $\sigma' \models \mathit{stable}$. If $\sigma \rightarrow \sigma_1 \rightarrow \ldots \rightarrow \sigma_n = \sigma'$ where every intermediary state is unstable: $\sigma_i \not\models \mathit{stable}$ for $0 < i < n$, we write $\sigma \twoheadrightarrow \sigma'$.

We are now able to formulate the following theorem, which states that if a stable valuation $\sigma$ is followed by a stable valuation $\sigma'$, then any stable valuation related to $\sigma$ by $R$ can be followed by a stable valuation that is related to $\sigma'$ by $R$. Thus, the theorem is a weakened version of Theorem 6.2.

**Theorem 6.7** *Let $R$ be a stable simulation relation as defined above. Given Constraints C1–C3, C4(b), C5–C10, C15, C16, and C17 (Table 6.3).*

*If $\sigma^{\mathsf{RLS}}\ \underline{R}\ \sigma^{\mathsf{ILS}}$, the following holds (as illustrated in Figure 6.12):*

*(i) for any $\sigma'^{\mathsf{RLS}}$, if $\sigma^{\mathsf{RLS}} \twoheadrightarrow \sigma'^{\mathsf{RLS}}$ then there exists a $\sigma'^{\mathsf{ILS}}$ such that $\sigma^{\mathsf{ILS}} \twoheadrightarrow \sigma'^{\mathsf{ILS}}$ and $\sigma'^{\mathsf{RLS}}\ \underline{R}\ \sigma'^{\mathsf{ILS}}$; and*

[2] We assume internal events are put in a separate queue $Q_{int}$ in order to implement Constraint C12.

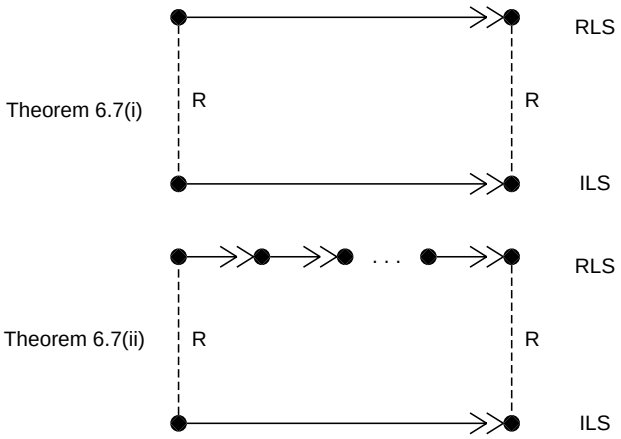



**Figure 6.12** *Illustration of Theorem 6.7. $R$ is a stable simulation relation. Black dots denote stable states. $\twoheadrightarrow$ abbreviates a sequence of transitions $\rightarrow \ldots \rightarrow$ in which all intermediary states are unstable*

*(ii) for any $\sigma'^{\mathsf{ILS}}$, if $\sigma^{\mathsf{ILS}} \twoheadrightarrow \sigma'^{\mathsf{ILS}}$, then there exists a sequence $\sigma_1^{\mathsf{RLS}}, \ldots, \sigma_n^{\mathsf{RLS}}$ of stable valuations under the RLS such that $\sigma^{\mathsf{RLS}} \twoheadrightarrow \sigma_1^{\mathsf{RLS}} \twoheadrightarrow \ldots \twoheadrightarrow \sigma_n^{\mathsf{RLS}}$ and $\sigma_n'^{\mathsf{RLS}} \underline{R}\, \sigma^{\mathsf{ILS}}$.*

*Proof.* The first claim follows immediately from Theorem 6.5: the reaction under the RLS to an arbitrary bag $E$ of event occurrences can be completely simulated in the stable ILS.

The second claim we prove as follows. Suppose a bag $E$ of events occur under the ILS. These events are processed one by one in a certain order under the ILS. We can relate this to behaviour under the RLS by letting the event processed under the ILS correspond to an occurrence of the same event under the RLS. So, if an event $e$ in $E$ is processed by the Router, i.e., $e$ is contained in $re$, then $e$ occurs under the RLS, i.e., $I$ is filled with $e$. After under the RLS $e$ has been processed, a stable state is reached. Both RLS and ILS can take the same step in response to $e$ and reach the same next configuration. For example, if $E = [e_1, e_2]$, and the order of event processing under the ILS is $e_2$ before $e_1$, we can simulate this under the RLS by a stable state in which $e_2$ occurs followed by a stable state in which $e_1$ occurs.

By taking a step some internal (RLS and ILS) and completion events (ILS only) can be generated. Under the RLS, completion hyperedges are not triggered by completion events, as they are taken immediately by definition of the superstep. Generated events are all responded to at the same time in parallel. Under the ILS, events are processed one by one. Completion events are processed before

other events. By definition, one completion event can trigger arbitrarily many hyperedges.

We now show that this difference in event processing, parallel event versus single event, does not have any impact upon which hyperedges are taken: under the RLS the same internal and completion hyperedges can be taken as under the ILS. By Constraint C6, completion and internal hyperedges do not conflict. Thus, taking an internal hyperedge under the RLS does not disable an enabled completion hyperedge. By Constraint C8, taking an internal hyperedge does not disable another enabled internal hyperedge. By Constraint C5, conflicting completion hyperedges have the same sources. So under the RLS the same completion and internal hyperedges can be taken as under the ILS and the same configuration can be reached as under the ILS.

□

Note that this theorem does not require Constraint C11, so under the RLS the environment does not need to regenerate events.

## 6.3 Conclusion

We have shown that for some activity diagrams the requirements-level and implementation-level semantics behave similarly. The major difference between the two semantics is that under the implementation-level semantics an external (non-termination) event can have unexpected extra effects. Therefore, the two semantics only induce similar behaviour if events are not observed. Even though this may seem weak, the theorems are sufficiently powerful to guarantee that the CTSs of both semantics satisfy the same functional requirements, as we will show in Chapter 10. Of course, such common functional requirements do neither refer to events. Since the requirements-level semantics induces a much smaller state space than the implementation-level semantics, such functional requirements can be more efficiently verified under the requirements-level semantics than under the implementation-level semantics. Yet the outcome of verification under the requirements-level semantics is equivalent to the outcome under the implementation-level semantics.

In this chapter, we have justified a semantics that assumes perfect synchrony by relating it to a semantics that does not make that assumption. In every other approach we know from literature, a semantics that assumes perfect synchrony is justified by putting an assumption upon both the environment and the system: The system should be fast enough in its reaction to current events to be ready before the next events occur. Our justification is more realistic and also more general (allowing more implementations): we allow the environment to be faster than the system. Drawback of our approach is that it is more involved and that it requires some additional constraints on activity diagrams, the implementation-

level semantics, and the environment. It is possible to relax some of the constraints, but this will complicate the proofs.

The identified constraints on activity diagrams may serve as guidelines for workflow modellers. If an activity diagram violates the constraints, then likely some construct in the activity diagram might lead to misinterpretation, at least by different WFSs and perhaps also by different persons.

# Chapter 7

# Advanced activity diagram constructs

In this chapter we discuss several advanced constructs for activity diagrams that we did not consider in the previous chapters. We will sketch how these constructs can be formalised in our two semantics, but do not provide the formalisations themselves. In the case of object flows and object nodes, we do not sketch a formalisation as this is premature. Instead we provide a list of issues that need to be resolved in formalising object flows.

Section 7.1 sketches how dynamic concurrency can be formalised. Section 7.2 discusses several issues regarding object nodes and object flows. Section 7.3 discusses deferred events. Section 7.4 sketches how an interrupt construct can be formalised.

## 7.1 Dynamic concurrency

In the UML, an atomic activity node or a compound activity node can have "dynamic concurrency". The activity (or activities) of a node with dynamic concurrency is instantiated multiple times in parallel, denoted by marking the activity node (or compound activity node) with '*'. The word 'dynamic' indicates that the number of instantiations is determined at run-time. The word 'concurrency' indicates that the instantiations execute in parallel. In Figure 7.1 an example of dynamic concurrency within an activity node is given (adapted from Fowler [71]). First, an order is received. Next, the order is filled with each line item on the order. Finally, the order is delivered. The number of instantiations of Fill line item is dynamic (i.e. determined at run-time), since this number depends on the order that is being processed.

We now sketch how dynamic concurrency can be formalised for activity nodes and compound activity nodes.

**Activity nodes.** We assume a set $DN \subseteq AN$ of atomic activity nodes whose activities are dynamically instantiated. Every activity node $a$ in $DN$ is annotated with a dynamic concurrency expression $*[expr]$. We assume $*[expr]$ evaluates to a list of elements, so $*[expr]$ is of type list. If $a$ is entered, for every element $e$ in list $[expr]$ an activity $act(a)_e$ is started. If all started activities are completed, $a$ terminates. So activity node $a$ now represents the execution of a dynamically determined number of activities, rather than the execution of one single activity.

We do not have to change our definition of configuration and steps, since for both an activity node with dynamic concurrency and an ordinary activity node, the node is part of the configuration and the node terminates. Only the function $act$ needs to be redefined: it is not static anymore, but dynamic: the activities assigned to an activity node that has dynamic concurrency are defined when the dynamic concurrency expression is evaluated, i.e. when the node is entered.

**Compound activity nodes.** In Section 3.3 we explained how compound activity nodes can be eliminated from an activity diagram by substituting the activity diagram specification of the compound activity node. If the compound activity node contains a dynamic concurrency expression $*[expr]$, we treat the activity diagram specification as a parametrised specification. Assuming $*[expr]$ evaluates to a list of elements, for each element in the list an instance of the activity diagram specification is executed.

We can model this by adapting the elimination procedure of a compound activity node as follows. We create a copy of the activity diagram specification for each possible element of the list. For every copy, each node of the copy is subscripted with the name of the corresponding element; so, different copies of activity diagrams have different nodes. We create a new activity diagram specification $n$, in which all these copies are put in parallel (so they are started at the same time). Only copies present in the list need to be executed: therefore before each copy is started it is decided in $n$ to either start the copy or skip the copy. The decision depends upon the evaluation of $*[expr]$. The decision is a simple test whether an element belongs to the list $*[expr]$ or not: if so, the copy is executed, otherwise it is skipped.

We illustrate the adapted elimination procedure by a simple example. Let compound activity node B in Figure 7.2 have a dynamic concurrency expression $dce$ that evaluates into a list of integers. Let this dynamic concurrency expression depend on some local variable $x$, where $x$ has type integer, set by activity $A$. If B is specified as in Figure 7.3, then the activity diagram after elimination of compound node B becomes as depicted in Figure 7.4. Note that there are infinitely many copies made of the activity diagram specifying B, since there are infinitely many integers.

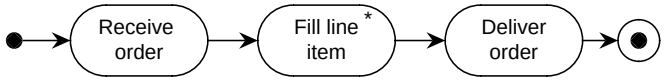



**Figure 7.1** *Example: multitask*

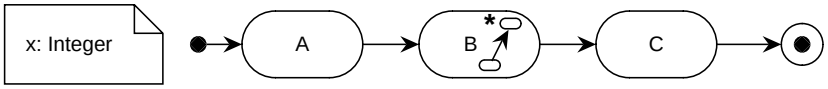



**Figure 7.2** *Dynamic concurrency example*

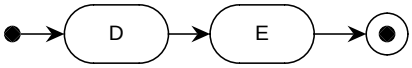



**Figure 7.3** *Specification of node* B *in Figure 7.2*

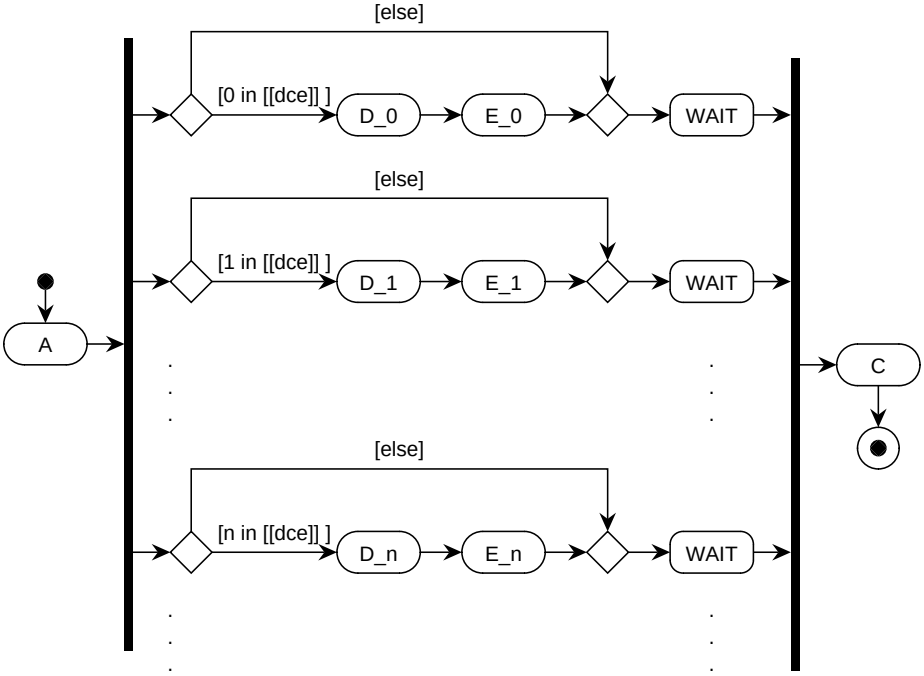



**Figure 7.4** *The activity diagram underlying the activity diagrams in Figures 7.2 and 7.3*

## 7.2 Object nodes and object flows

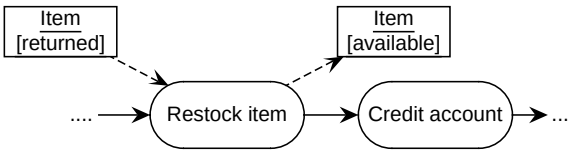
UML activity diagrams offer a notation for modelling data flows, namely object flows and object nodes. An object node denotes an object that is in a certain state. The name of the object is underlined. (UML uses underlining to distinguish objects from classes, whose names are not underlined.) The name of the state is put in brackets. So O[s] denotes object O in state s. The name of the state is optional. An object node can be connected to an activity state by a directed edge, called an object flow, meaning that the activity either produces the object as output (edge from activity to object node) or that the activity needs the object as input (edge from object node to activity). Like UML 1.4, we represent object flows with dashed lines in order to distinguish them from control flows.

In UML 1.4 [150] object flows are interpreted as control flows. Each object flow also specifies a control flow. This definition is however counterintuitive, which was pointed out by Bock [21]. Consider for example the partial activity diagram in Figure 7.5(a), which was taken from Booch [24]. As Bock [21] pointed out, this diagram is not in accordance with its intended meaning, which is presumably that after Restock item terminates, *both* an Item object in state [available] is output *and* the activity node Credit account is entered. But if we apply the UML rule that every object flow implies a control flow, the diagram in Figure 7.5(a) says that after Receive item terminates, *either* an Item object in state [available] is output *or* the activity node Credit account is entered, but not both. Presumably, what Booch intended is shown in Figure 7.5(b). If we drop the UML 1.4 rule that an object flow specifies a control flow, then Figure 7.5(a) makes sense.

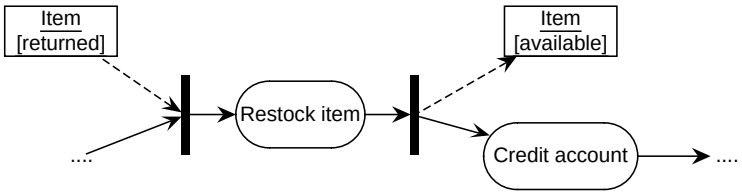
Most authors that use object flows (e.g. [59, 138]) seem to assume the meaning used by Booch. It is therefore not surprising that in one of the key proposals [11] for the UML 2.0 definition (of which the part on activity diagrams is edited by Bock), the control flow and data flow of activity diagrams are indeed separated from each other, like depicted above. We also assume that meaning in the remainder of this Section. To make things confusing, the UML 2.0 proposal not only allows Figure 7.5(a) but also 7.5(b); these two activity diagrams presumably have the same meaning.

Since the definition of object flows is still under development, formalisation is currently premature. The current proposal for UML 2.0 [11] contains some gaps and inconsistencies that we discuss here. So we merely identify some issues and discuss possible resolutions. We take the afore mentioned UML 2.0 proposal as starting point, even though it may change in the future.

Before we discuss the issues, we explain the basic execution semantics of object flow nodes as defined in the UML 2.0 proposal [11]. The questions to be answered by any execution semantics are the following. If an activity has more than one input node, do all inputs need to be present in order for the activity to start, or can some inputs arrive later? Likewise, can an activity output an object before the activity actually terminates, or are all objects output upon termination of the




(a) Incorrect according to UML 1.4, but correct according to UML 2.0




(b) Correct according to UML 1.4 and UML 2.0

**Figure 7.5** *Two example activity diagrams with object flows*

activity? The UML 2.0 proposal allows all four possibilities. It divides the set of input and output object nodes of an activity in two, synchronous and asynchronous ones. The following rules apply:

- All synchronous input objects must be present for the activity to start. If there are only asynchronous input object, at least one must be present.
- All inputs must be present for the activity to terminate.
- When the activity terminates, all synchronous output objects are posted, i.e., the synchronous output object nodes are filled.

Asynchronous output object nodes can be filled before the activity terminates.

Next, we discuss several assumptions we make in interpreting activity diagrams with object flows. All assumptions are motivated by the domain of workflow modelling.

- Objects modelled in activity diagram are software objects, not hardware objects, since a WFS can only coordinate software objects, not hardware objects.

  So a text processor file can be an object modelled in an activity diagram, but a paper file cannot be.

- Objects are subject to transactional constraints like atomicity and isolation (cf. Sections 2.2 and 3.4).

- The dependency between an activity and its input objects is enforced by a WFS.

  For example, in Figure 7.5(a), a WFS will ensure that activity *Restock item* is only enabled once object Item is present and moreover is in state returned.

- The dependency between an activity and its output objects is the responsibility of the actor doing the activity. The actor must ensure that the right objects are output.

Using these assumptions, we interpret input and output object flows in the following way (see Figure 7.6). If an activity outputs an object, but does not use it as input, that activity *creates* the object. If an activity observes an input object node and fills an output object node of the same object, it *updates* that object. This usually means that the state of the object is changed. If an activity only observes an input object node, but does not fill an output object node, then the activity *reads* the object, but does not change it. Note that we have not fixed a notation for deleting an object. We consider deletion to be a special activity that does not occur in an activity diagram, i.e. in a workflow.

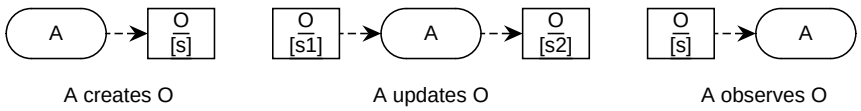



**Figure 7.6** *Specifying creation, updating and observation of an object*

Both the current UML definition [150] and the the UML 2.0 proposal [11] do not address what the intended meaning of connecting an object O with an activity $A$ is, apart from being input or output. Above interpretation seems reasonable for workflow modelling.

We now list several issues in the semantics of object flows and object flow nodes.

**Multiple occurrences of an object name.** Do multiple occurrences of an object name O denote the same object or different objects? The most straightforward interpretation of O is to see it as an object identifier. Object identifiers are unique, i.e., each identifier denotes the same, unique object throughout the system life-cycle (this principle is called referential transparency). So multiple object nodes with the same label O denote the same object.

This interpretation may give rise to problems, however, if an object flow node is active more than once at the same time. For example, Figure 7.7 shows an activity diagram in which two objects Partial order are created, even though there is only one identifier.

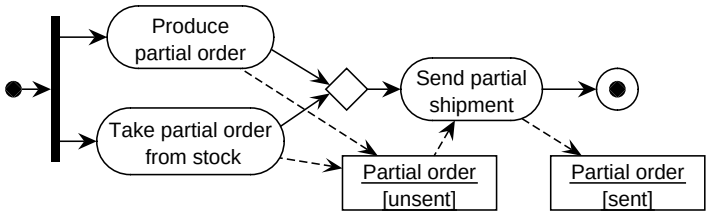



**Figure 7.7** *Activity diagram with two objects* Partial order

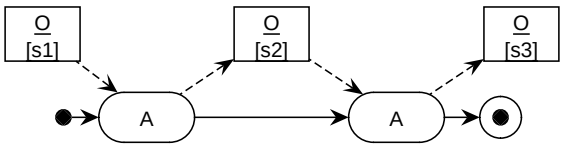



**Figure 7.8** *Multiple activity instances*

**Multiple occurrences of an activity.** Can an activity $A$ change the state of an object O twice at different times? In that case, the input/output behaviour of $A$ depends upon the context of $A$. Figure 7.8 shows a simple example, in which the input/output behaviour of $A$ depends upon whether or not $A$ has been executed before. Drawback of such activities is that they are harder to reuse, as their behaviour depends upon the context in which they are used.

**Multiple occurrences of O[s]: partial and complete states.** If an object flow node is labelled with a state [s], there are several possible interpretations. Either [s] is a *complete* specification of the state of the object, or a *partial* one. In the latter case, O's lifecycle will be modelled using a statechart with AND nodes. This distinction is very important, since partial object nodes can be active simultaneously, whereas complete object nodes cannot! So, if [s] is a complete specification and O[s] is active, all other object nodes of O should be inactive.

Which interpretation should be used, complete or partial, depends upon the specification under consideration. For example, in Figure 7.9 each of the three states [s1], [s2] and [s3] must be complete specifications of the state of O. If they would be partial, O would not be in a complete state. The corresponding statechart will have three sequential states.

On the other hand, in Figure 7.10 the states [s2] and [s4] cannot both be complete state specifications. If they would be complete, then [s2] = [s4] whereas they do not have the same name. Hence, [s2] and [s4] must be partial states. In the corresponding statecharts, nodes s2 and s4 are in parallel. And (s2, s4) is a complete state specification. Similar remarks apply for s3 and s5.

Note that modelling partial object nodes can be dangerous. If multiple activi-

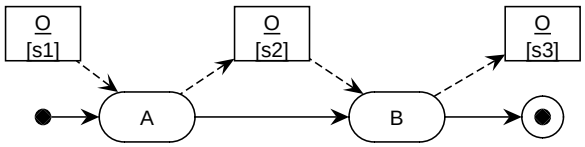



**Figure 7.9** *Complete state specification*

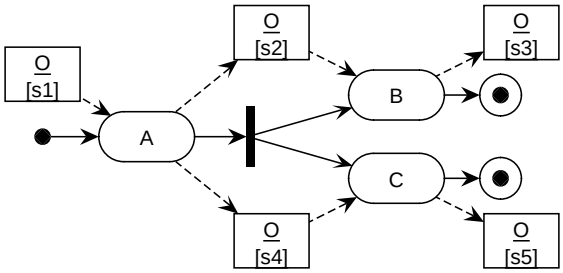



**Figure 7.10** *Partial state specification*

ties act upon the same object at the same time, the data integrity of the object can be violated, as well as the isolation property of activities (cf. Section 2.2 and 3.4).

Finally, there is the problem of matching states in the object flow specification with states in the statechart modelling the life cycle of the object. If an object node O[s] occurs more than once in the same object flow specification, then in the corresponding statechart there can either be a single node labelled s or multiple nodes labelled s. Thus, more than one statechart can match an activity diagram with object flows.

**Multiple activities updating the same object.** Objects are subject to integrity rules, in order to prevent them from being inconsistent. For example, the activity diagram in Figure 7.11 shows an object that is updated by two activi-

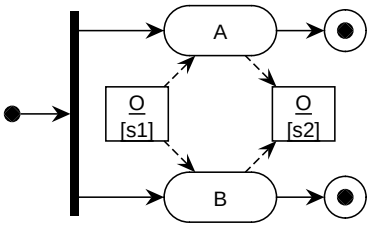



**Figure 7.11** *Object* O *is updated by two activities at the same time*

ties at the same time and therefore might become inconsistent, even though both activities have the same output. This should be prevented from happening, for example by ruling out such an activity diagram or putting some extra constraint on the semantics, in this case for instance interleaving activities $A$ and $B$.

The UML 2.0 proposal only partially addresses this issue. The proposal for the UML action semantics [12] requires that a data flow state is filled only once during one execution (called the *single assignment rule*). Applying this rule to activity diagrams, the requirement would be that an object node is filled at most once during a run. Unfortunately, this rule is both too weak and too strong for our purposes.

It is too strong in the sense that it rules out object nodes that are part of an iteration. Figure 7.12 gives an example (adapted from an example in Grefen et al. [82]). The activity diagram in Figure 7.12 would be incorrect according to this rule, since object node Trip[proposed] can be filled more than once during a run. We would allow it, however, because the activities filling Trip[proposed] are never active at the same time. So the isolation property is not violated.

The rule is too weak in the sense that it still allows multiple updates to the same object at the same time. This happens if object nodes are a partial specification of the state of the object, like in Figure 7.10. Then two object nodes referring to the same object can be active at the same time in parallel. With a complete state specification, object nodes cannot be active in parallel, so then the rule is not too weak.

A more useful rule in this respect is the *isolation rule*: an object cannot be updated and either read or updated at the same time (isolation property from database theory). We have adopted a similar rule in Section 3.4. The rule is defined for runs; it cannot be translated straightforwardly into a rule on the syntax of activity diagrams.

**Merging.** One can interpret the activity diagram in Figure 7.11 as specifying a merge. However, we think it is better to represent merging by an explicit *Merge* activity, as shown in Figure 7.13. This reflects that merging is done explicitly by an actor (person or application), not implicitly by the WFS. Note that in Figure 7.13, we divided state s2 in two parallel states s2a and s2b. Allowing merging might allow for a relaxation of the isolation constraint, because the merging activity can solve the inconsistency.

**Multiple activities observing the same object.** If multiple activities observe the same object at the same time, does this mean that only one activity at a time can read the object (i.e. the object is read-locked), or that more than one activity at a time can observe the object? For example, in Figure 7.14 two activities $A$ and $B$ observe the same object O at the same time. If only one activity at a time can observe O, the two activities must execute in interleaving order, otherwise not.

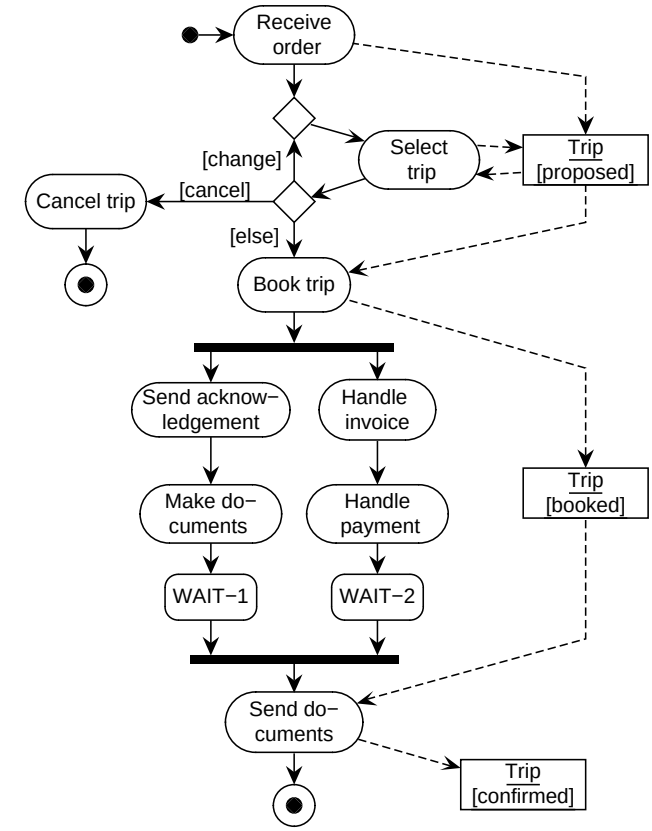



**Figure 7.12** *Activity diagram with iteration*

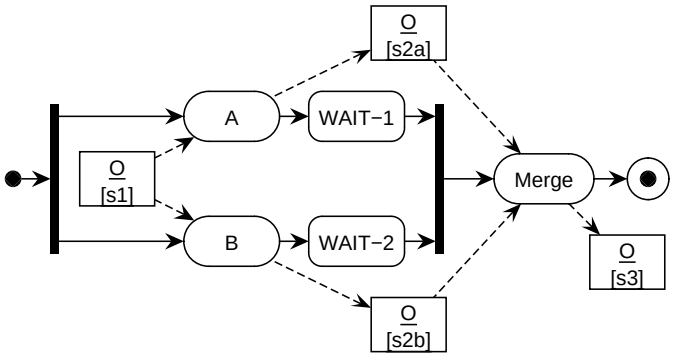



**Figure 7.13** *Merging object flows*

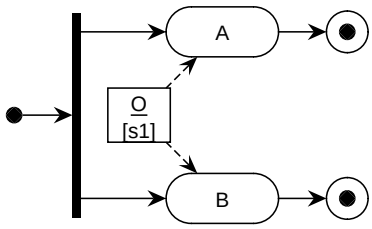



**Figure 7.14** *Object* O *is observed by two activities at the same time*

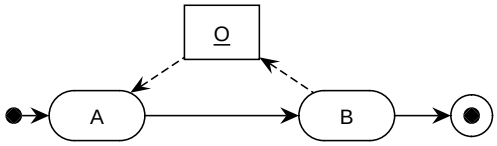



**Figure 7.15** *Example inconsistency between control flow and object flow*

**Inconsistency between control flow and data flow.** The control flow and object flow in an activity diagram might be inconsistent, which is undesirable. For example, in Figure 7.15 activity $A$ requires that object O is present. Object O is output by activity $B$. Activity $B$ can only start once $A$ has completed. Clearly, the control flow and object flow are inconsistent in this example: Executing this diagram leads to a deadlock. A possible way to check whether activity diagrams are inconsistent or not is to use the verification approach using model checking that we develop in Chapter 10.

**Object nodes and pseudo nodes.** Can pseudo nodes connect object nodes by object flows? The UML proposal answers this question positively; it allows for instance the two activity diagrams in Figure 7.16.

We make the following observation. In the two diagrams pseudo nodes cause

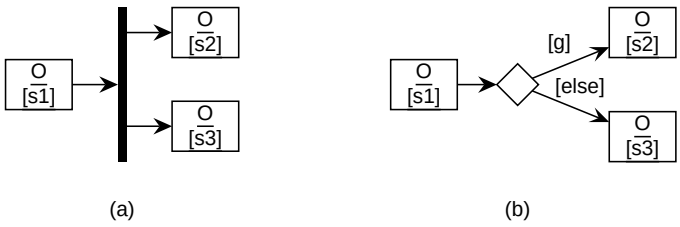



**Figure 7.16** *Object nodes connected with pseudo nodes*

a state change with object O. Since compound edges are executed by the WFS, this means that the WFS changes the state of O. This does not seem to make much sense, as activities change objects, not the WFS. So pseudo nodes should not connect object nodes.

## 7.3 Deferred events

It is possible to specify for each node $n$ a list of deferred events. If an event $e$ is deferred in $n$, it is not processed if the current configuration contains $n$. Processing of $e$ is postponed until a configuration is reached in which no node defers $e$. It is unclear from the UML standard whether or not an event that was deferred in the previous state but becomes undeferred in the current state, has priority over events in the queue that were not deferred in the previous state.

Incorporating deferred events in both our semantics will require an additional event queue for storing deferred events.

Deferring events may easily lead to deadlock, because a configuration in which the event is no longer deferred may not be reachable. For example, if in Figure 7.17 the current configuration is [WAIT-1,WAIT-3] and event e occurs, this event is deferred. If next event f occurs, configuration [WAIT-1,WAIT-6] is reached. Event e is still deferred in this configuration. Now the system deadlocks: event e can never be processed, even though e can trigger a relevant edge. Such deadlocks can be detected using the verification approach described in Chapter 10.

## 7.4 Interrupt regions

The afore mentioned UML 2.0 proposal contains the novel concept of an interrupt region and an interrupt edge. An interrupt region is a set of nodes that are left if one of the interrupt edges of the region is triggered. Figure 7.18 shows an example, a simplified version of an example in the UML 2.0 proposal [11]. According to the informal semantics given in the UML 2.0 proposal, if event cancel order request occurs while nodes Receive order, Fill order, Ship order, or WAIT-1 are active, the nodes in the interrupt region are left and node Cancel order is entered.

Figure 7.19 shows an activity diagram that is obtained from Figure 7.18 by eliminating the interrupt region. This shows that an interrupt region is actually a shorthand to abbreviate many edges. Statecharts use hierarchy in a similar way [88]. Note that the interrupt edge cannot be eliminated.

From Figure 7.19, it becomes clear that there are some difficulties with interrupt regions. First, Figure 7.19 suggests that activities can be interrupted. In Section 2.1 we explained that activities are atomic, i.e., they cannot be interrupted. Thus, an interrupt occurring while an activity is busy executing can only be handled once the activity has finished. Incorporating this in our two semantics will lead to a considerably more complicated and intricated version of both semantics.

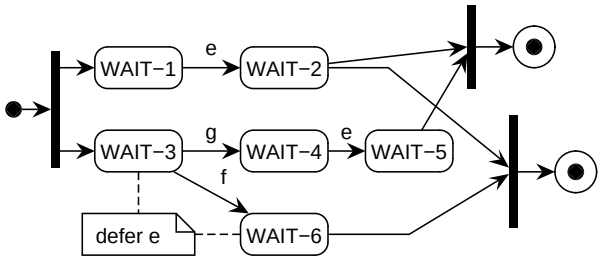



**Figure 7.17** *Activity diagram with a deferred event*

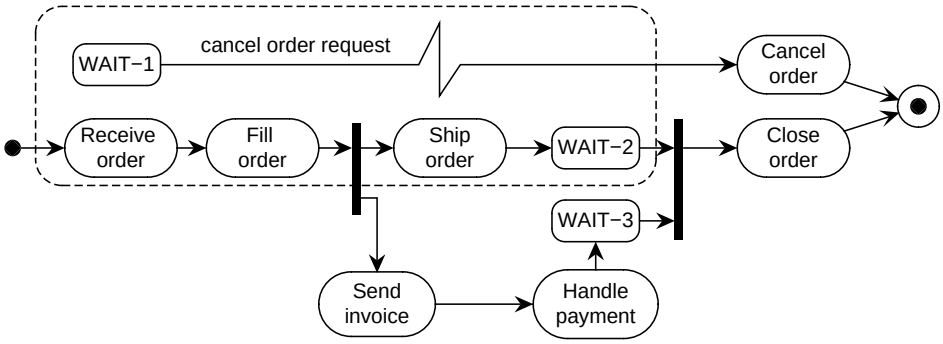



**Figure 7.18** *Activity diagram with interrupt region (adapted from [11])*

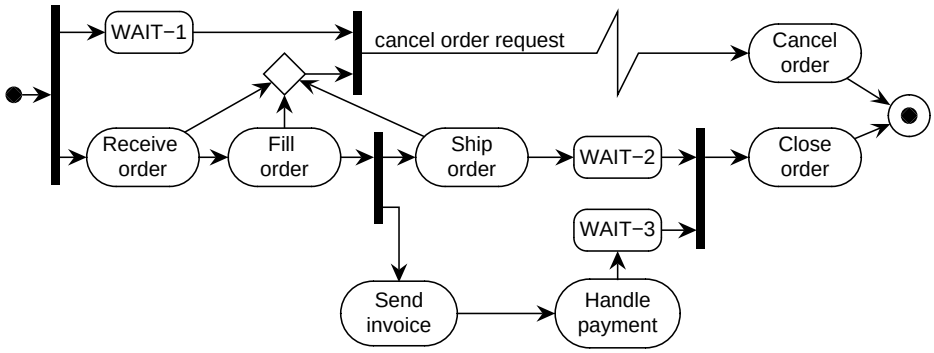



**Figure 7.19** *Activity diagram without interrupt region that is similar to the one in Figure 7.18*

Second, an interrupt may cause a deadlock, because some parallel threads are interrupted while others are not. For example, a deadlock occurs if event cancel order request occurs while node Send payment or Handle payment is active. Such an interrupt should not be allowed.

Third, the notion of interrupt suggests a sense of priority: an interrupt event has higher priority than a non-interrupt event. In the requirements-level semantics, an interrupt event and non-interrupt event can be processed simultaneously, and thus can enable conflicting hyperedges. In that case, the hyperedge enabled by the interrupt event should be taken, rather than the other hyperedge. This can be formalised by introducing a priority ordering on hyperedges, like is done in the STATEMATE semantics of statecharts [90]. In the implementation-level semantics, a priority ordering on hyperedge is not really useful, because interrupt events and non-interrupt events are not processed at the same time. Instead, the priority rule on events must be extended by stating that interrupt events should be processed before before other events. Then it is unclear whether interrupt events should have higher priority than completion events. Of course, extending the two semantics with interrupts means that the theorems in Chapter 6 have to be extended as well.

The UML 2.0 proposal [11] suggests a Petri net based token-game semantics for activity diagrams. The standard Petri net token-game semantics does not have the notion of interrupt. Stochastic Petri nets do have a notion of priority between transitions: a transition can only fire if no enabled transition with higher priority can fire. This notion can be used in the UML 2.0 proposal.

# Chapter 8

# Comparison with Petri nets

In this chapter we compare our two execution semantics for activity diagrams with existing Petri net semantics. We do this by focusing on how several – what we think are – important aspects of workflow models are modelled in Petri nets and in our semantics. In order to make a fair comparison we assume that a Petri net models a WFS too.

We use the design choices made in Chapter 4 as a yard stick for our comparison. This approach may seem subjective, since other persons might make other design choices, and consequently they might draw other conclusions about the suitability of Petri nets for workflow modelling. However, we think that the choices we have made in our semantics are reasonable, because they are motivated by the domain of workflows.

To recapitulate from Chapter 4, our most important design choice is that the semantics for activity diagrams must be reactive. The token-game semantics, which is characteristic for Petri nets, does not represent reactivity, which is characteristic of workflow systems (see Section 4.2). A Petri net transition (hyperedge) can fire if all its input places are in the current marking [129, 136]. But in a reactive system a transition (hyperedge) can be taken (fired) if all its source nodes (input places) are in the current configuration (marking) *and* its trigger event occurs [90, 150]. This trigger event is an event in the environment of the system, that the system will react to by taking the transition. Moreover, an enabled Petri net transition *may* fire, whereas an enabled transition in a reactive system *must* fire. Although the token-game semantics of Petri nets is not reactive, we will study different ways of *simulating* reactive behaviour in different Petri net variants.

In the sequel, we presuppose some basic knowledge of Petri nets and high-level Petri nets (see e.g. [105, 129, 136, 141, 142]). We have looked at Petri net variants that are traditionally used to specify and analyse workflows, namely Workflow Nets [2, 3], Information Control Nets [55], INCOME/WF [132], FunSoft nets [49, 58], Milano WFMS [10]. Next, we have looked at Petri net variants that are not

specifically tailored towards workflow modelling but nevertheless can be useful: Open Nets [16], Petri nets with synchronous communication [37], Signal-Event Nets [87, 70], Contextual Nets [128, 72], Zero-Safe nets [27], and several variants of Object-Oriented Petri Nets [9, 121]. More information about some of these references can be found in recent overviews and collections about the use of Petri nets for workflow modelling [1, 144]. A comparison of our semantics with other (formal) modelling techniques can be found in Chapter 9.

**Note on terminology.** In this chapter we will use the standard Petri net terminology of 'place' (corresponds to node), 'transition' (corresponds to hyperedge), and 'marking' (corresponds to configuration). By 'step' we mean a statechart step, unless stated otherwise.

The remainder of this chapter is structured as follows. The first four sections focus on the requirements-level semantics, since it most resembles the Petri net semantics. The implementation-level semantics is considered at the end of the chapter. Section 8.1 discusses how events can be modelled in Petri nets. Section 8.2 studies whether and how the statechart step semantics can be modelled in Petri nets. Section 8.3 studies how data can be modelled in Petri nets. Section 8.4 discusses how activities can be modelled in Petri nets. Section 8.5 discusses how the implementation-level semantics can be modelled in Petri nets. Section 8.6 looks at several Petri net variants for workflow modelling. Section 8.7 highlights some aspects of the question what actually a Petri net is. We end with discussion and conclusions.

## 8.1 Modelling events

Several researchers that use Petri nets for workflow modelling have recognised the importance of input events for workflow modelling ([2, 106]), even though they use a different name: 'trigger'. Figure 8.1, taken from Van der Aalst [2], presents a typical example of the use of input events in a Petri net. The activity diagram in Figure 8.2 models the same workflow. The envelope and the clock denote external and temporal trigger events respectively.

Unfortunately, although the importance of input events is recognised, hardly ever a semantics is given for them. Van der Aalst [3] gives an interesting motivation for abstracting from events for analysis purposes, that we will discuss below. But first we study two approaches to specify events in ordinary Petri nets and compare both approaches with our semantics of input events.

**Event as token.** For each input event a place is defined. The place represents a kind of interface with the environment. If the interface place is filled with a token, the input event occurs, otherwise it does not occur. The interface place is connected with all transitions that are triggered by the input event.

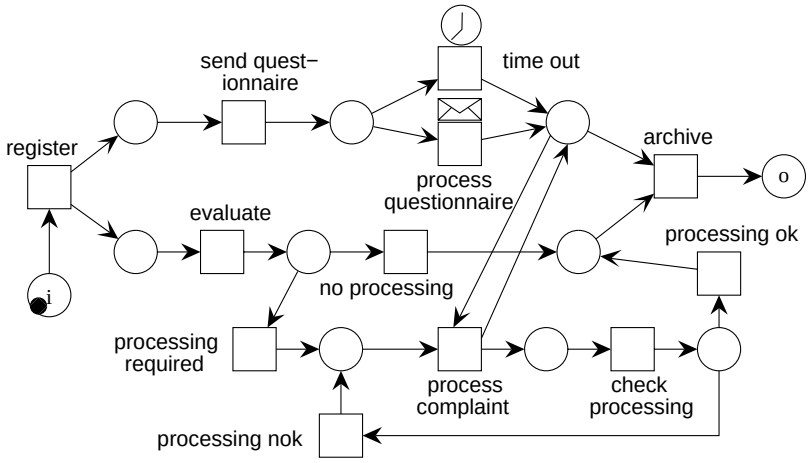



**Figure 8.1** *Petri net for "Processing complaints" workflow [2]*

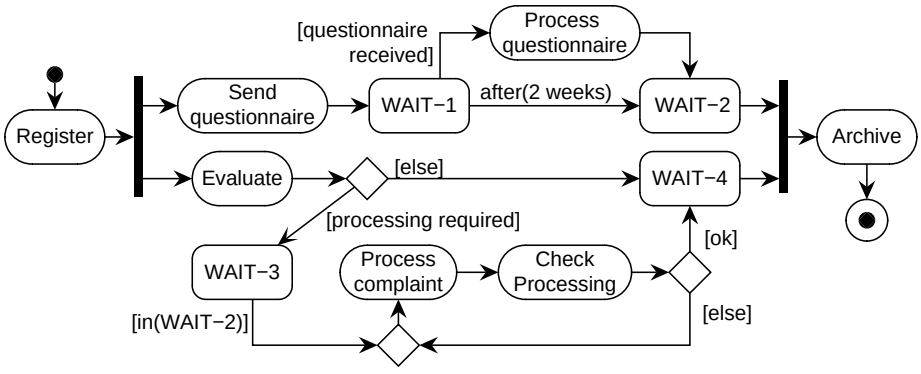



**Figure 8.2** *Activity diagram for "Processing complaints" workflow*

This is the approach taken in Trigger Modelling [106]; it is also suggested as an appropriate semantics for trigger events in Workflow Nets [2]. In these approaches, the environment is not specified. The suggestion is made that the environment fills the interface places spontaneously, but no formal semantics is presented. Open nets [16] gives a formal semantics for nets with interface places, which could be used for Trigger Modelling and Workflow Nets.

One important difference of the event-as-token approach with our semantics of events is that in the event-as-token approach one event occurrence triggers at most one transition whereas in our semantics one event can trigger more than one transition (edge). This is because we also allow event broadcasting in our semantics, in addition to point-to-point communication. For example, in Figure 4.2 on page 47 one occurrence of event $e$ can trigger the two edges `e1` and `e2` simultaneously. Since in the standard Petri net token-game semantics, firing a transition implies that its input tokens are consumed, in the token-game semantics only one transition can fire because of one event occurrence.

One might wonder whether event broadcasting is desirable. In other words, isn't the standard Petri net interpretation of consuming events, so having an event trigger at most one transition, better? We think event broadcasting is desirable for the following reasons. First, if an event would trigger a single transition only, the event would not have all the effect that is specified in the workflow model. For example, a cancel event that stops a workflow would be awkward to model. In the event-as-token approach, a cancel event could only stop one parallel branch, whereas in our semantics an event is global for the whole workflow and there a cancel event can stop the whole workflow. To cancel a workflow in the event-as-token approach, for every parallel branch a separate cancel event needs to be generated.

Second, the broadcast mechanism is used quite extensively in the field of workflow systems. Several non-Petri net based WFMS prototypes [30, 77, 85, 158]) also use a broadcast semantics in their workflow models. The industry standard for workflow interoperability [133], defined by OMG and WFMC, uses a publish-subscribe notification mechanism, which is similar to our broadcast semantics. An exception are XML and EDI based workflow specifications, which currently only use point-to-point communication between business partners. However, some of these approaches [153] will adopt publish-subscribe notification in the near future. Also, these approaches only consider inter-organisational communication. They do not specify what communication mechanisms are used within an organisation, since it falls outside the scope of these frameworks. So even in these approaches, a broadcast mechanism can be used for intra-organisational communication.

Third, we observe that our broadcast semantics is equivalent to a point-to-point semantics if all the used event names in the activity diagram are unique. But it is not possible to fully capture the broadcast semantics with point-to-point communication, since the exact addressee is not always known at design time and may depend upon the current state of the case. We explain this point in more

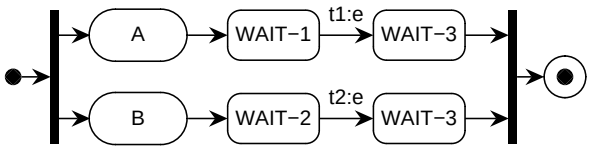



**Figure 8.3** *Event broadcasting*

detail by trying to simulate broadcasting in Petri nets.

There are several ways to simulate the effect of event broadcasting in Petri nets. The most obvious one is to use transition fusion and glue the (hyper)edges with the same event label together. Although this would work for the example in Figure 4.2, this is only a partial solution, for two reasons. The first one is that it depends upon the current configuration (marking) of the activity diagram whether or not two edges are taken simultaneously. For example, in Figure 8.3 the two edges are only taken simultaneously if the current configuration is [WAIT-1,WAIT-2]. Otherwise, if for example the configuration is [WAIT-1,B] and e occurs, then only t1 is taken and configuration [WAIT-3,B] is reached. So, only at run-time it is known which (hyper)edges need to be fused together, whereas transition fusing is applied at design time. Second, applying transition fusion at design time does not solve this problem, since the original (hyper)edges cannot be left out. For example, if in Figure 8.3 edges t1 and t2 are fused together into t12, then edges t1 and t2 must remain in the model, since it possible that either one of them is taken separately from the other. Consequently, if the current configuration is [WAIT-1,WAIT-2] and event e occurs, it still might be possible that only say t1 is taken, and not the fused edge t12.

Another possible way to simulate event broadcasting is to fill the interface place with as many tokens as needed to prevent that a transition cannot fire because of a lack of tokens. But the exact number of tokens that is needed is not known beforehand, since the number of transitions to be fired depends upon the current WFS state. Consequently, a lot of spare tokens would have to be introduced. This blurs the difference between two occurrences of the same event at different times and two copies of the same event occurrence. Although this could be resolved by time stamping tokens, the resulting semantics would be overly complex and more involved than the statechart step semantics.

A better alternative is to specify the control flow between an interface place and a transition that it triggers as a read arc [128], also known as context relation. A read arc from a place to a transition means that although a token must be present in the place to let the transition fire, this token is not consumed. (A read arc from a transition to a place is impossible.) Technically, a flow relation specifying the read arcs is added to the standard syntax of a Petri net [128].

A final drawback of the event-as-token approach is that the resulting Petri net

looks like ravioli, since the place where the input token $i$ resides must be connected to all transitions that are triggered by $i$.

**Event as transition.** In Petri nets, one can simulate an event by labelling a transition with the event name and interpreting the firing of the transition as the event occurrence. By specifying synchronisation constraints [37, 70] between the event transition and the system transitions, it can be specified that an event occurrence triggers a system transition. Note, however, that then the environment is being modelled explicitly, rather than implicitly as in the event-as-token approach. In other words, the whole Petri net is now a model (specification) of both the environment and the WFS, rather than of the WFS only.

One advantage of this semantics is that it is very easy to specify that one event occurrence can trigger more than one system transition, since the synchronisation constraint is specified as just a relation between transitions.

We now discuss the motivation Van der Aalst provides for abstracting from events. Although Van der Aalst recognises the need for modelling input events, he abstracts from them for analysis purposes for two reasons [3]. His first argument is that the environment cannot be modelled completely; from the point of view of the WFS it behaves nondeterministically. This is best modelled, he says, by leaving the events out. His second argument is that if an abstracted workflow is correct, the concrete one will also be. But ideally, however, it should also be the other way around. Moreover, as we showed above, both arguments fail to hold if one event can trigger more than one transition. In that case, abstracting from events will lead to different behaviour in the abstract model, when compared to the concrete model. Consequently, the verification results obtained for the abstract net might not be reliable anymore.

**Conclusion.** We conclude that events are not first-class citizens in Petri nets. Events can be simulated by using either tokens (open nets with read arcs) or transitions (nets with synchronisation constraints between transitions). In the next subsection, we will study how well the statechart step semantics that we use can be modelled using these two approaches.

## 8.2 Modelling steps

In the previous subsection we identified the two ways, open nets with read arcs, and nets with synchronisation constraints between transitions, that come closest to our event semantics. We now study whether and how well the statechart step semantics can be modelled in these approaches.

**Event as token.** If we take the Petri net step semantics, it is no problem in the event-as-token approach to model that events can occur simultaneously. But it is difficult to specify in open nets with read arcs that events live for the duration of one step only (see Section 4.3), without changing the semantics of open nets at this point. The removal of an event occurrence has to be modelled by a separate transition that removes the token from the interface place. But the sequence 'event occurrence-system reaction-event removal' which is key part of the basic statechart step semantics, is not part of the standard Petri net token-game semantics. It seems to us that it is impossible to model this sequence using a token-game semantics, since in this latter semantics any sequence of transitions, obeying the firing rules, is allowed. So, it could be possible that under the standard Petri semantics an event lives longer than a step, i.e., is not removed after it has been responded to.

Recently, a new Petri net variant, called zero safe nets [27], has been proposed that seems a good starting point for modelling the statechart step semantics. In zero safe nets, some places, called zero places, represent unobservable system states. A marking in which one or more zero places are filled is unstable, otherwise it is stable. During execution, the system moves from one stable marking (in which zero places are not filled) to another stable marking via a sequence of unstable markings. By modelling event places as zero places, the statechart step semantics can be simulated to some extent. Still there is a difference: zero safe nets have a constraint that all stable tokens present at the begin stable marking must be consumed during the sequence. For the statechart step semantics, this would mean that relevant transitions must fire, which is of course not true, since some may not be triggered at the moment.

Finally, the constraint that steps are maximal is not present in standard Petri net semantics. Rather, a step in the Petri net semantics can be an arbitrary consistent subset of the bag of enabled transitions. Of course, the maximality constraint could be added without a problem (like incidentally done by some authors, e.g. Foremniak and Starke [70]), but it does not seem very intuitive for the standard Petri net semantics. In fact, Foremniak and Starke [70] have considerably changed the standard Petri net token-game semantics. We discuss their approach in more detail below.

**Event as transition.** There are several Petri net variants that have incorporated synchronisation between transitions in their models [37, 9, 70]. The work of Christensen and Hansen [37] introduces the concept of synchronous transitions. They focus on symmetric synchronisation. Object-oriented Petri nets [9, 121] use both symmetric and asymmetric synchronisation between transitions, i.e., one transition has the initiative, the other one follows. All these references stick to the standard interleaving semantics, which differs considerably from the statechart step semantics, among others because the maximality constraint is not required.

Finally, in signal-event nets [87, 70] the standard Petri net step semantics is

abandonded in favour of a semantics in which also a maximality constraint is adopted. Signal-event nets are introduced by Hanisch and Lüder [87] in order to model discrete event systems. To model a discrete event system, both the behaviour of an uncontrolled plant and of a controller that guides the behaviour of the plant is modelled. Hanisch and Lüder argue that discrete event systems cannot be faithfully modelled using ordinary Petri nets. Discrete event systems are an excellent example of reactive systems: the controller must react to the behaviour of the plant and it does this in order to maintain the plant in a desired state. It is therefore interesting to note the similarities (and differences) between the execution semantics of signal-event nets and that of statecharts. Foremniak and Starke [70] introduce an execution semantics for signal-event nets. The key part of the execution of a signal-event net is a step. Before we discuss their definition of a step in more detail, we fix some terminology [70]. A transition is *forced* it is triggered by another transition; otherwise it is *spontaneous*. (So in standard Petri nets, every transition is spontaneous.) A transition $t$ can be forced by more than one transition. There are two options in that case: either all trigger transitions must occur simultaneously to trigger $t$ (AND), or only one trigger transition has to occur in order to trigger $t$ (XOR). We only consider the XOR interpretation here.

A set $s$ of transitions is *signal complete* iff

(i) If $s$ only contains spontaneous transitions, it is signal complete.

(ii) If $s$ is signal complete, $t \notin s$ is forced and $t$ is triggered by a transition $t' \in s$, then $s \cup \{t\}$ is signal complete.

A step $s$ must satisfy the following constraints:

1. $s$ contains transitions that are fired spontaneously, i.e., without being triggered by another transition,
2. the input places and input conditions (for read arcs) contain sufficient tokens for all transitions in the step to fire,
3. $s$ is signal-complete,
4. for every non-spontaneous transition $t'$ that is not in $s$, set $s \cup \{t'\}$ does not satisfy 1-3.

We can easily see the correspondence with our semantics: spontaneous transitions are transitions in the environment, representing events, whereas forced transitions are transitions done by the WFS. Constraint 1 states that every step must be triggered by at least one event. Constraint 2 states that all the transitions in the step must be enabled and consistent. Constraint 3 says that a transition that is triggered by an input event $e$ can only be part of the step if $e$ occurs. Constraint 4 states that the step be maximal. It is not difficult to see, that these constraints

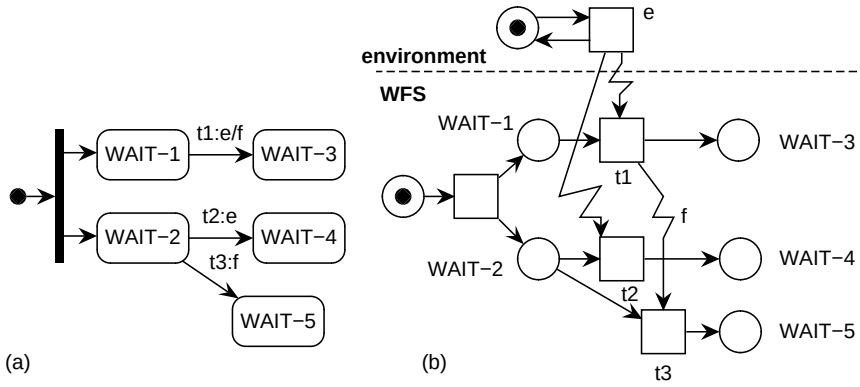



**Figure 8.4** *Event generation modelled in signal-event nets*

are indeed equivalent to the constraints we discussed in Sections 4.3 and 5.2 for the basic statechart step semantics without event generation.

But, this definition differs with our semantics w.r.t. the generation of events during a step, since it assumes that events generated by the system are sensed immediately in the same step (as in the statechart fixpoint semantics [140]), whereas we assume that they are sensed after the current step has been taken (as in STATEMATE [90] and UML [150]). To illustrate this, we translate the activity diagram in Figure 4.2, shown in Figure 8.4(a), into a signal-event net, shown in Figure 8.4(b). The interrupt arcs represent triggering; the triggered (forced) transition is pointed at. Suppose the current marking of the signal-event net is [WAIT-1,WAIT-2] and transition e fires. Then both [e,t1,t2] and [e,t1,t3] are valid steps, according to the constraints listed above. This is similar to the behaviour of the corresponding activity diagram (statechart) under the statechart fixpoint semantics of Pnueli and Shalev [140], which we explained in Section 4.3. But in our semantics, only [t1,t2] would be possible. As explained in Section 4.3, we regard the fixpoint semantics (and thus the signal-event step semantics) as counterintuitive here, since it seems that e is ignored in node WAIT-2 if step [t1,t3] is taken.

It is easy to show that the signal-event net execution semantics is a strict subset of the fixpoint statechart semantics (strict because in the statechart variant on which the fixpoint semantics is defined, a hyperedge can be labelled with a negative event ($\neg\ e$), which is true iff the event does not occur. Negative events cannot be defined in signal-event nets).

A more intricate example is presented in Figure 8.5. Figure 8.5(a) shows an activity diagram and Figure 8.5(b) a corresponding signal-event net. The predicate $\mathsf{in}(x)$ that is used in the activity diagram, evaluates to true iff node $x$ is contained in the current configuration. It can be translated into a Petri net construct using

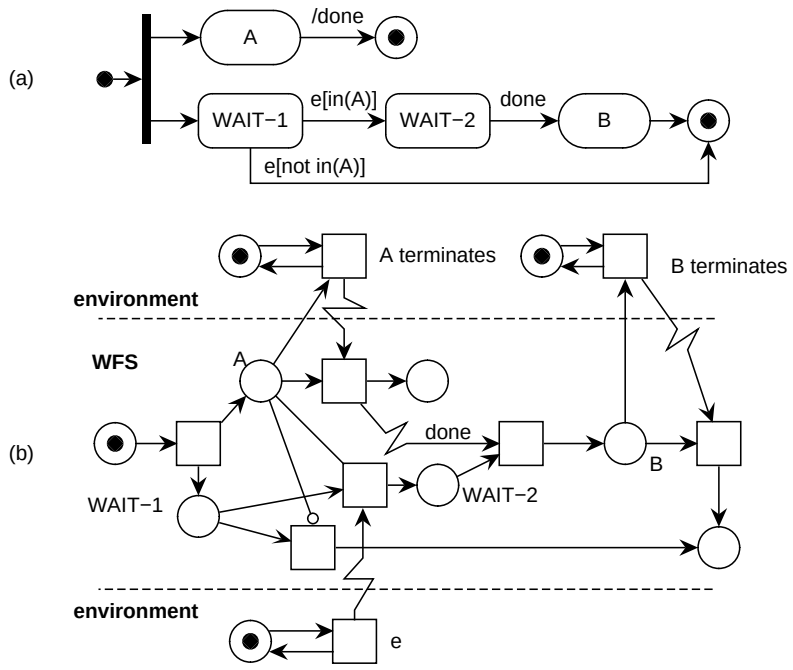



**Figure 8.5** *Activity diagram and a similar signal-event net*

read arcs and inhibitor arcs. Inhibitor arcs are necessary to model guard expression not in(x). In Figure 8.5(b), read arcs are lines, so do not have an arrow; inhibitor arcs are lines with a circle at the transition end. If the current configuration is [A,WAIT-1] and at the same time activity $A$ terminates and event e occurs, then in our semantics a sequence of steps is taken such that finally configuration [final,B] is reached. But in the corresponding signal-event net, configuration [final,WAIT-2] is reached and event done is not responded to and is lost! Consequently, the final configuration will never be reached in that case. This is clearly undesirable. We therefore prefer our semantics of event generation.

Of course, now the question arises whether our semantics of event generation can be simulated in signal-event nets. We think that this is impossible, since in both our semantics the bag of input events acts as a kind of registry in which the events that are generated during a step are stored. This can only be simulated by treating events as tokens; if events are treated as transitions, events get lost after the step in which the events are generated completes. But above, we discussed the inadequacy of the event-as-token approach to model our step semantics. We come back to this issue in the conclusion of this subsection below.

Other differences between activity diagrams and signal-event nets are that (1)

steps in signal-event nets are *sets* of transitions, rather than bags (but we could not find a compelling reason in Foremniak and Starke [70] why this is the case; probably the extension to bags is easily made), and (2) the environment must be modelled explicitly in signal-event nets, but not in activity diagrams.

**Conclusion.** Both in the event-as-token approach as in the event-as-transition approach, the statechart step semantics we have adopted cannot be modelled. However, in the event-as-transition approach, the signal-event net semantics resembles the statechart step semantics we use closely. The major difference is that the signal-event semantics has a fixpoint semantics of generated events, whereas we have not. Our semantics of event-generation can only be modelled using the event-as-token approach. It might therefore be worthwhile to try to incorporate the concepts used in signal-event nets into the event-as-token approach. Especially the concept of a forced transition seems promising. This concept seems to be present in zero safe nets as well.

## 8.3 Modelling data

The standard way to incorporate data in Petri nets is to use coloured tokens [105]. Coloured tokens are tokens that have attribute values. These attribute values are modified in/by transitions. Another way is to interpret places as predicates [74]. But then instances of the predicates can be seen as tokens that can change value when a transition consumes them. So, in both approaches tokens carry data.

Therefore, the straightforward way to model case attributes in Petri nets is to attach these attributes to tokens. But attaching case attributes to tokens suffers from the following problems.

**Who updates case attributes?** If case attributes are updated in some transition, then this transition cannot be part of the workflow model, because the WFS who executes the workflow specification does not update case attributes, it only routes the case (see Section 4.1). In other words, if case attributes are modelled in Petri nets, the environment (the actor) must be specified explicitly by a transition in order to let the case attributes change value.

**Data integrity.** Several tokens may represent the same case attributes. Ideally, this situation should be prohibited, since an attribute may then have several different values (i.e., the different tokens may assign different values to the same attribute): then the attribute is inconsistent. In terms of transaction theory, the isolation property fails to hold, since activities that update the tokens are not isolated from other executing activities.

One possible solution is to represent each case attribute by a single coloured token. Then each transition that reads or writes the attributes must have this

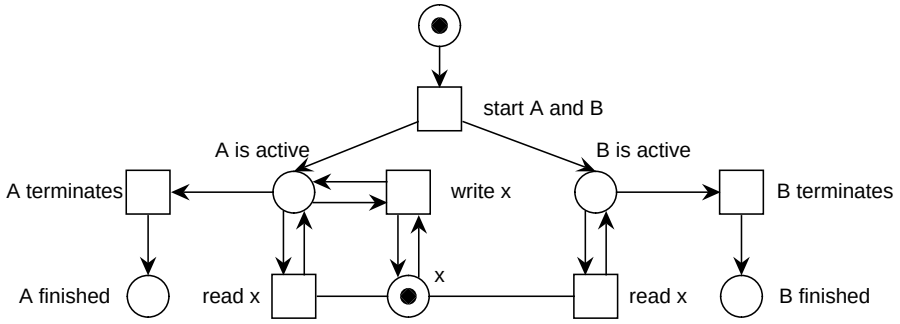



**Figure 8.6** *Example of concurrent access to shared data. Activities A and B both access data item* x

token as input and outputs the token when it finishes. Although the isolation property is then ensured, in standard Petri nets two read activities cannot be simultaneously active, since both consume the same token. In other words, a token is scarce. That is not what we want, because the concurrency of the WFS is then reduced. In both our semantics, read data is not scarce: many activities can read the same attribute at the same time. Thus, our semantics allows for more concurrency than the Petri net semantics. Another drawback of Petri nets is that the resulting net would look like ravioli if there are many case attributes.

To circumvent this, read arcs [128] can be used for read access. Interestingly, apparently read arcs have been proposed just to solve this problem of simultaneously access to shared data [72]. But unfortunately, read arcs do not solve the problem satisfactorily. To illustrate this, consider the Petri net with read arcs in Figure 8.6. Data item x is updated and read by activity $A$ and read only by $B$. In this net, although x cannot be updated and read simultaneously, it is possible that $B$ reads a value of x that is subsequently changed by $A$. So, activities $A$ and $B$ are not isolated from each other (viewing both activities $A$ and $B$ as separate transactions.) Therefore, this solution does not satisfy our needs. (De Francesco et al. [72] do not address this issue; they only consider the question when two Petri net executions are view equivalent.)

In fact, in our semantics we have ensured that if two activities are conflicting, that is, one of them writes a case attribute that the other one reads or writes, then they cannot be active simultaneously (see Sections 3.4 and 5.2). In the definition of a step, we have put the constraint that by taking the step a configuration is reached in which there are no conflicting activities. This conflict relation can of course be specified in the control flow as well, using for example a mutex place for each pair of conflicting activities. The mutex place acts as a kind of semaphore: the activity that can consume the token in the mutex place may be active and change the data item it likes and when it terminates it puts a token in the mutex place. A solution using mutex places would, however, clutter the workflow specification

with a lot of arrows, and again, we have a ravioli model, that is more unreadable and incomprehensible than the workflow specification presented in Figure 8.6.

**Conclusion.** We conclude that data can be modelled in Petri nets using read arcs, mutex places, and an explicit representation of the environment to model updates of case attributes. But, the resulting net is overly complex, unreadable and uncomprehensible. We think that a solution using local variables (used in Petri nets modelling flowcharts [75]) is more simple and elegant, and therefore preferable.

## 8.4 Modelling activities

In a Petri net, there are two options to model an activity: as a transition or as a place. Almost every Petri net workflow specification seems to take the first option, whereas if an activity diagram is viewed as a Petri net, the second option is taken. We discuss the advantages and disadvantages of each option.

**Activity is transition.** In almost every Petri net workflow specification that we know of, this interpretation is adopted, probably because of the intuition that an activity is something which changes the state of the case (the state is assumed to be modelled by the input tokens). There are, however, some mismatches between the properties of an activity and the properties of a transition. First, a transition takes no time to execute, whereas an activity does. There are two ways to solve this problem. The first solution is to decompose the transition into a "begin activity" transition and "end activity" transition that are connected by a place representing "activity busy executing". This solution results in a Petri net that is quite similar to an activity diagram. Then the execution of an activity is actually represented by a place. This approach is taken by for example Van der Aalst, Van Hee and Houben [6] and Desel and Erwin [50]. See the next item below for a discussion of this approach.

The second solution is to use timed or stochastic Petri nets, in which a transition can have a duration. In most timed and stochastic Petri net variants a transition still fires instantaneously, but it takes time before a transition is enabled. The transition in that case actually represents the starting or ending of an activity rather than the complete execution of the activity. This is not harmful for analysis purposes, but it gives a slightly awkward model of WFS reality. Analysis of timed and stochastic Petri nets is far more complex and involved than analysis of simple low-level nets.

However, our main objection against modelling an activity as a transition is the following. In Petri nets, a transition is executed by the system that the Petri net models. Hence, if a transition models an activity, this implies that the WFS does the activity. This approach violates the WFS characteristic that an activity

is performed by an actor in the environment of the WFS, not by the WFS itself. *And it is this characteristic that creates the need for reactivity in a WFS.* By contrast, in our semantics the WFS does not do activities; it merely routes cases. In Petri nets that model activities as transitions, the routing is not modelled at all. Therefore, such Petri nets do not model a WFS.

As an aside, note that in some variant of Workflow Nets [5], some transitions can be labelled with a silent action that is not observable for the environment. The semantics of these nets is defined in process algebra. Van der Aalst [4] suggests to use the silent step to model routing transitions [5]. Transitions labelled with an observable action then represent workflow tasks. However, in that process-algebraic semantics, the silent action can be abstracted from sometimes. For example, a sequential workflow specification with two tasks $a$ and $b$ and a routing transition from $a$ to $b$ is equal to a model in which $a$ is directly followed by $b$. It is unclear how this abstraction can be related to the execution of real workflow models: a WFS always routes a case after an activity terminates. In our view, routing cannot be abstracted from.

Of course, one could model the environment also in the Petri net workflow model, and let the activity be performed by the environment part of the Petri net model. But then the relationship with the corresponding part of the workflow is unclear, i.e., what should the WFS do while the environment is busy performing some activity?

**Activity is place.** To the best of our knowledge, this interpretation is never chosen in Petri nets. Most people modelling a workflow in Petri nets probably would find this interpretation counter-intuitive since (as they argue) during an activity the case is changed, whereas a place is static (the local part of the case is not changed). We disagree, however, with this argument, since for a WFS an activity state *does* represent something static, namely the WFS waits for an actor to complete the activity. The only dynamic behaviour of the WFS is when events occur, e.g. some activity terminates, and the case must be routed to a new state.

Nevertheless, the Petri net people who find this interpretation counter-intuitive are right to some degree. Whether we represent activities as places or as transitions, in Petri nets case attributes can only be changed in transitions, not in places. This corresponds to the fact that Petri nets model closed, active systems, in which the environment, i.e., that which is outside the Petri net, does not play any role. Any model of an open, reactive system, on the other hand, does allow for a change of case attributes during a state (place), namely if the change is initiated by the environment! (These changes are implicitly modelled and not explicitly represented by edges in the diagram.) And this is exactly what happens during an activity: the environment (i.e., an actor using an application) updates case attributes, whereas the WFS waits for the activity to terminate. Consequently, to model change of case attributes in a Petri net, we must model the environment explicitly in the Petri net as well.

**Conclusion.** Most Petri net workflow modelling approaches model activities as transitions. The only motivation that is given for this choice is that this is "straightforward" and "intuitive". We think that the real, underlying motivation is based upon the following properties of Petri nets: (1) a transition represents some change by the system, whereas a place represents a static condition on the system modelled by the Petri net, and (2) a Petri net can only change state by firing transitions. The two properties imply that all changes are caused by behaviour of the system itself. In other words, changes cannot occur due to the environment of the system. Consequently, any modelling language having these properties cannot faithfully model open, reactive systems; instead, such a language is more suitable for modelling closed, active systems. Thus, Petri nets are useful for example to represent (scarce) resource usage, e.g. the allocation of actors to activities, but not for modelling open, reactive systems.

## 8.5 Modelling the implementation-level semantics

In the implementation-level semantics, only a single event can be processed at at a time. Therefore, a queue is needed to store events that occur while the Router is busy processing some event. We now discuss whether this can be simulated using the event-as-token and event-as-transition approaches.

In the event-as-token approach, a queue can be modelled straightforwardly by switching to Petri nets with integers (counters) as is done in the FunSoft approach [58]. And a special place can be introduced to store the event that is currently being processed by the Router. But still, since the event-as-token approach cannot simulate very well the statechart step semantics that we use, it cannot simulate the implementation-level semantics very well either.

In the event-as-transition approach (signal-event nets), queues cannot be modelled, since the effect of an event is lost after the step in which the event occurs is completed. So, if in the environment an event occurs (a spontaneous transition fires), the WFS must react immediately, since otherwise the event will be lost. So, in the event-as-transition approach, the implementation-level semantics cannot be simulated at all.

We conclude that the implementation-level semantics cannot be modelled satisfactorily using Petri net semantics.

## 8.6 Petri nets for workflow modelling

We now discuss the Petri net models we found in literature that are used to specify and analyse workflows.

Van der Aalst, Van Hee and Houben [6] use high-level nets to model and analyse Petri net based workflow models that also model resources. Van der Aalst [3] uses

Workflow Nets, low-level Petri nets with a single start and a single end place, to verify proper termination of a workflow model.

FunSoft nets [49] are high-level nets for software process modelling, but they can also be used for workflow modelling. Their semantics is defined in terms of Predicate/Transition nets [74]. FunSoft nets focus on the flow of resources (objects), like business documents, through an organisation and do not focus on modelling events. Some shorthands are defined to model for example FIFO queues. Several analysis techniques, including verification, have been developed for FunSoft nets [49].

INCOME/WF [132] is a a workflow management system based on high-level Petri nets where the tokens are nested relations. Nested relations are introduced to increase the concurrency of the net: the actual transitions are defined on the basic elements of the relation, not on the relation itself. This implies, however, that for the basic elements, no concurrency exists, since the standard Petri net firing rule is employed, in which a transition consumes all tokens it reads.

Information/Control Nets [55] are a high-level Petri net variant for workflow modelling. The focus is on the modelling of resources, like documents, not on the modelling of events.

Milano [10] is a research prototype to investigate flexible workflow models. Only low-level Petri nets are considered. The Petri nets cannot contain loops and must be safe.

It is interesting to notice, from this brief overview, that most Petri net workflow models provide little support for modelling events. And if a notation for events is suggested, no formal semantics for them is given. Most approaches interpret tokens as resources that are being used by activities in transitions to deliver a requested service for a customer. However, the assumption seems to be implicitly made that resources are *scarce*, since no two transitions can consume the same token simultaneously. This assumption is questionable: certainly, some resources are scarce, but also a wide variety of resources, read-only information carriers like catalogues, are not. (None of these Petri net variants uses read arcs.)

## 8.7 What is a Petri net?

Even if one does not agree with the choices we made, our discussion gives – we hope – more insight in possible answers to the question: "What is a Petri net?" [51].

Most people will answer this question by saying that a Petri net is bipartite graph, whose nodes are places and transitions, that are connected by directed arcs. This is characteristic of a certain syntactic definition of a Petri net. However, an equivalent definition says that a Petri net is a hypergraph: a set of nodes (places) connected by directed hyperedges [134]. The relation between these two definitions is as follows: Each hyperedge corresponds to a transition; if a place is in the source (target) of a hyperedge, then there is an arc from (to) the place

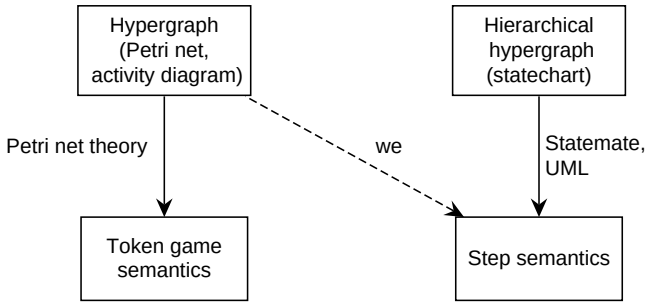



**Figure 8.7** *Possible semantics for (hierarchical) hypergraphs*

to (from) the corresponding transition. Since the underlying syntactic structure of an activity diagram is also a hypergraph (see Chapter 3), the question arises whether an activity diagram is a Petri net. In this chapter, we have shown that an activity diagram as we view it is *not* a Petri net, because the semantics attached to a Petri net (in fact, to any Petri net variant we found in literature), differs from the semantics we defined for activity diagrams in Chapter 5. The major difference is that our activity diagram semantics is reactive whereas the Petri net token-game semantics is not.

Moreover, we have shown that a statechart-like semantics can be given to a notation with a Petri net-like syntax (see Figure 8.7). For some obscure reason, this seems hard to swallow for some people. We quote one of the UML 2.0 proposals [11], which suggests a Petri net-like semantics for activity diagrams (in UML 1.4 activity diagrams are given a semantics by translating them into statecharts):

> Activities are redesigned to use a Petri-like semantics instead of state machines. Among other benefits, this widens the number of flows that can be modeled, especially those that have parallel flows.

This citation reveals that the reason why the UML 2.0 proposal has chosen a Petri net semantics, is that statecharts were considered to be too strict for modelling concurrency in workflows. This limitation of statecharts is due to the hierarchy rules of statecharts, which rules out some forms of concurrency (see Section 9.1). The hierarchy rules, however, are part of the statechart syntax, not of the statechart semantics.

The UML 2.0 proposal seems to reason as follows: remove the hierarchy constraints from statecharts (activity diagrams), and the result is something that looks like Petri nets. So the UML 2.0 proposal seems to make the following equation:

$$\text{statechart} - \text{hierarchy} = \text{Petri net}$$

But this equation is false. The equation only says something about the differences and similarities in syntax of statecharts and Petri nets: a statechart is a hierarchi-

cal hypergraph and a Petri net is a hypergraph. We actually have the following equation.

$$\text{statechart syntax} - \text{hierarchy} \approx \text{Petri net syntax}$$

Our semantics shows that Petri net-like diagrams (like activity diagrams) can be given a statechart-like step semantics (cf. Figure 8.7), which differs radically from a token-game semantics. The UML 2.0 proposal does not consider these differences in semantics. The real issue is whether a step-based or token-game semantics reflects WFS behaviour more accurately. We think a step-based semantics is more accurate than a token-game semantics.

## 8.8 Discussion and conclusion

From our comparison of our semantics with Petri net semantics, we draw the following conclusions. First, Petri nets model closed systems. All changes in Petri nets occur because of the firing of some transitions in the net that represent activity of some part of the system itself, rather than some activity in the system's environment. Our semantics models an open system.

Second, the standard Petri net token-game semantics models active systems, rather than reactive ones. A transition is enabled if its input places are filled. Also, an enabled transition does not have to fire immediately. Our semantics is reactive. An edge in an activity diagram is enabled if its source nodes are in the current configuration and its trigger event occurs in the environment. And an enabled edge *must* fire immediately. That is why we impose a maximality constraint on steps in our semantics. This constraint is lacking in the standard Petri net token-game semantics.

In Petri nets, reactivity can be simulated to some extent by modelling the environment in the Petri net as well. This is done, for example, in a recently proposed variant of Petri nets, called signal-event nets. Like activity diagrams, signal-event nets are motivated by the domain of reactive systems. Signal-event nets have a complex semantics that differs considerably from the standard Petri net token-game semantics (among others, a maximality constraint is imposed on steps). Since their semantics is so different from the token-game semantics, it is questionable whether these are Petri nets at all. We showed that signal-event nets have similar behaviour as statecharts under the fixpoint step semantics, defined ten years earlier by Pnueli and Shalev [140]. We are convinced that it is impossible to simulate in Petri nets our semantics of event generation, that is used in both UML and STATEMATE. However, it might be worthwhile to try to incorporate the concepts of signal-event nets into Petri net variants that model events as tokens, for example open nets. The resulting Petri net variants would likely be closer to the UML and STATEMATE interpretation of event generation than any of the currently existing Petri net variants, but we expect such variants will still be different.

Third, Petri nets in general model scarce resources, rather than unscarce ones. A transition can only fire if there are enough input tokens present, i.e., enough scarce resources are available. Using read arcs this can be circumvented, because with a read arc a token can be tested without being consumed. Thus, read arcs allow for elegant specification of concurrent access to shared data. However, if an activity is seen as a transaction, as is usually done in workflow modelling, read arcs must be combined with mutex places to enforce isolation between activities. We prefer our own approach using local variables, since it is more simple.

Fourth, the Petri nets that came closest to the requirements-level semantics we gave to activity diagrams contained inhibitor arcs, read arcs, synchronisation between transitions, and coloured tokens with timestamps. These nets had to contain both a description of the workflow and of the environment. Roughly speaking, the net had twice as many nodes compared to the corresponding activity diagram. Such Petri nets are truly gargantuan and difficult to analyse, both for a workflow modeller and a verification tool. (For example, a lot of the analysis results for standard Petri nets do not carry over to signal-event nets [70].) Moreover, these Petri nets still stay far away from our implementation-level semantics. They do not resemble it at all.

Also, one of the acclaimed advantages of Petri nets, that there is an abundance of analysis techniques available for them [2], is only true for low-level nets; its applies to a lesser extent to high-level nets. As we showed, not every desirable construct can be modelled in low-level and high-level nets; read arcs, inhibitor arcs and synchronisation constraints are needed as well. But for these latter net variants, there are only but few analysis techniques available. In Chapter 10 we show how activity diagrams with similar constructs can be efficiently verified using model checking.

Some WFMSs, for example Cosa [147], use Petri net variants as workflow modelling language. But even though these WFMSs use the Petri net syntax, this does not necessarily mean that they use the Petri net token-game semantics! For example, in Cosa activities are modelled as transitions. In the token-game semantics of Petri nets, a transition fires instantaneously whereas in real life an activity will not be performed instantaneously. So, although the Petri net syntax is used by this WFMS, it is doubtful whether the Petri net token-game semantics is used. Moreover, the semantics that WFMSs in general attach to input workflow specification is unknown, as vendors do not publish the semantics they implement. But for analysis purposes, the semantics a WFMS uses should be known, since analysing a Petri net-based workflow specification, using, for example, the Petri net token-game semantics, presupposes that this semantics gives an accurate description of the real workflow behaviour.

Of course, above conclusions are based upon our assumption that a workflow specification describes the behaviour of a WFS. One could argue whether this assumption is valid. In fact, does not a Petri net workflow specification describe an organisation, rather than a computerised system? But even then, the

interaction between the organisation and its environment must be modelled (customers, government, suppliers,...), since the organisation is a reactive system as well. Consequently, Petri net models of organisational behaviour suffer from the same problems as Petri net models of WFS behaviour: they cannot model reactivity.

From the above, it follows that if the standard Petri net token-game semantics is used in modelling a reactive system, the reactivity of that system is abstracted from. But we consider reactivity to be one of the most important aspects of workflow modelling. If reactivity is abstracted from, then at least some justification should be given that assures that the analysis results on the Petri net model will also carry over to a reactive setting. We do not think such justification has been given yet (at least we have not found one in literature). If no justification is given, it is unclear what the relationship is between execution of a workflow specification according to the Petri net token-game semantics and the actual execution of a similar workflow specification by a WFS.

# Chapter 9

# Related work

In the previous chapter we have compared our two semantics of activity diagrams with various Petri net semantics. In this chapter we will compare our two semantics with other related work, including statechart semantics, and other formal workflow modelling languages that are not based on Petri nets.

Section 9.1 compares our approach with statecharts, in particular the STATEMATE and UML variants. Section 9.2 looks at the informal OMG semantics of UML activity diagrams. Other formalisations of UML activity diagrams are discussed in Section 9.3. Section 9.4 looks at the state of the art WFMSs to see in what respect they can implement our semantics. Section 9.5 considers other formal workflow modelling languages. The last two sections focus on active databases and transactional workflows. We end with conclusions.

## 9.1 Statecharts

In order to keep the presentation simple, we only discuss the major differences between our two semantics of activity diagrams and the two existing statechart semantics, STATEMATE and UML, that inspired both semantics. More details can be found elsewhere [63].

**Statechart syntax.** The most striking difference between the syntax of activity diagrams and Petri nets on the one hand and the syntax of statecharts on the other hand is the different representation of parallelism (concurrency). In statecharts, parallelism is represented through a hierarchy of nodes, whereas in activity diagrams and Petri nets it is not.

The hierarchy relation in statecharts must be a tree: a node can have at most one parent (but a parent can have more than one child). Leaf nodes of the tree correspond to the nodes of an activity diagram (and to the places of a Petri net).

Leaf nodes are called BASIC nodes. There are two kinds of non-leaf nodes, AND nodes and OR nodes. OR nodes are used to group sequential states, whereas AND nodes are used to put different groups of sequential states in parallel.

The state of a statechart must satisfy the following constraints on AND nodes and OR nodes:

- If the system is in an AND node, then it is in every child of the AND node.
- If the system is in an OR node, then it is in exactly one child of the OR node.

These constraints ensure that every node, including BASIC nodes, is active at most once at the same time.

Due to these syntactic constraints, the concurrency that can be expressed in statechart syntax is limited. We show this by three examples. First, the activity diagram in Figure 9.1(a) would translate into the statechart in Figure 9.1(b). The AND node in Figure 9.1(b) is needed since nodes Produce partial order and Take partial order from stock are in parallel. But in the activity diagram in Figure 9.1(a), node Send partial shipment can be active more than once at the same time, since activities *Produce partial order* and *Take partial order from stock* both start their own separate instance of activity *Send partial shipment*. When the workflow stops, always two instances of activity *Send partial shipment* have been

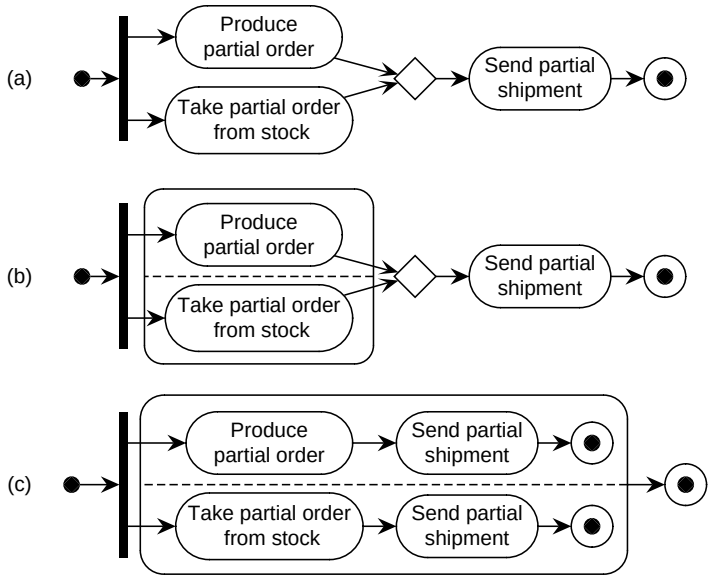



**Figure 9.1** *Activity diagram and two statecharts with different behaviour*

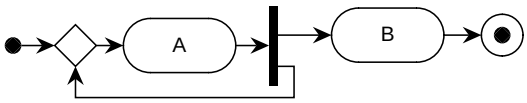



**Figure 9.2** *Activity diagram that cannot be translated into a statechart*

performed. Whereas in the statechart of Figure 9.1(b), if for example activity *Produce partial order* terminates and *Take partial order from stock* is still running, the AND node and all its subchildren, including Take partial order from stock, are left. So node Take partial order from stock is left, even though the corresponding activity is still running! And when the workflow stops, activity *Send partial shipment* has been performed only once, instead of twice.

The reason for this difference in behaviour between statecharts and activity diagrams is due to the syntactic constraints on AND and OR nodes in statecharts. These constraints enforce that a statechart node is not active more than once at the same time. This does not mean, however, that no statechart can express similar behaviour as the activity diagram in Figure 9.1(a). By replicating node Send partial shipment, the behaviour of the activity diagram in Figure 9.1(a) can be simulated in a statechart, as shown in Figure 9.1(c).

Our second example, the activity diagram in Figure 9.2, shows that not every activity diagram can be mimiced with a statechart by replicating some nodes. Each time node A terminates, a new instance of B is enabled, even though some instances of B are already active. In fact, node B can be active unboundedly often at the same time, i.e. there is no bound on the maximum number of simultaneous instantiations of B (the notion of unboundedness comes from Petri net theory). Since in a statechart a node cannot be active more than once at the same time, this activity diagram cannot be translated directly into a statechart.

The activity diagram cannot even be simulated by a statechart using node replication. To represent unboundedness of some node $n$ in a statechart using node replication, we need unboundedly many copies of node $n$. Since the copies can be in parallel, they should have some AND node $a$ as common ancestor. Every AND node has a fixed number of child nodes (for example, each of the AND nodes in the statecharts in Figure 9.1 has two children, OR nodes). All the children of an AND node are in parallel, not just some children. Thus, all copies of $n$ are active in parallel at the same time. But unboundedness of $n$ implies that the number of simultaneous instantiations of $n$, i.e., the number of copies in parallel, can change over time, so is not fixed. So it is impossible to model unboundedness in statecharts using node replication.

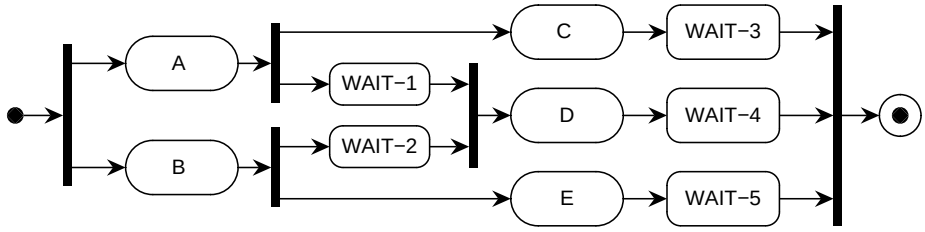



**Figure 9.3** *Another activity diagram that cannot be translated into a statechart*

This discussion might give the impression that every activity diagram in which the bounds are at most one can be translated directly into a statechart without having to resort to replicating nodes. But that is not true, as is shown by our third example in Figure 9.3. Let us try to translate this in a statechart. Since nodes A and B are in parallel, they must in a statechart be a subchild of two different parallel OR nodes, say $\mathsf{A}_{or}$ and $\mathsf{B}_{or}$. Because B and C can be active at the same time, C must be a subchild of $\mathsf{A}_{or}$. By similar reasoning, node E must be a subchild of $\mathsf{B}_{or}$. The AND/OR hierarchy constraints enforce that node D belongs to exactly one OR node. Since D is successor of A, node D must belong to $\mathsf{A}_{or}$. Since D is successor of B, node D must belong to $\mathsf{B}_{or}$. But $\mathsf{A}_{or}$ and $\mathsf{B}_{or}$ are in parallel, so not hierarchically related. So D can never be a subchild of both $\mathsf{A}_{or}$ and $\mathsf{B}_{or}$.

Note that not even node replication would help to translate the activity diagram in Figure 9.3 into a statechart. A possible solution is to remove the synchronisation between A and D by removing node WAIT-1 and its ingoing and outgoing edges. The precedence relation between A and D can be enforced by labelling the edge from WAIT-2 to D with guard $[\mathsf{in}(\mathsf{C}) \vee \mathsf{in}(\mathsf{WAIT\text{-}3})]$. Then D can be subchild of $\mathsf{B}_{or}$.

The OMG semantics of UML 1.4 activity diagrams translates an activity diagram into a statechart and uses the statechart semantics. To ensure that every activity diagram can be translated into a statechart, UML 1.4 only allows activity diagrams in which each fork is eventually followed by a join and in which multiple layers of forks and joins are well nested. This constraint rules out the activity diagrams in Figures 9.1, 9.2 and 9.3. The constraint is sufficient but not necessary to translate an activity diagram into a statechart. Figure 9.4(a) shows an activity diagram that violates the constraint, because the fork and joins are not well nested. Figure 9.4(b) shows the statechart that the activity diagram translates into.

We conclude that the hierarchy constraints in statecharts rule out certain forms of concurrency, that are allowed in activity diagrams and Petri nets. That is why we do not give a semantics to activity diagrams by defining a translation of an activity diagram into a statechart. Instead we have defined our semantics directly

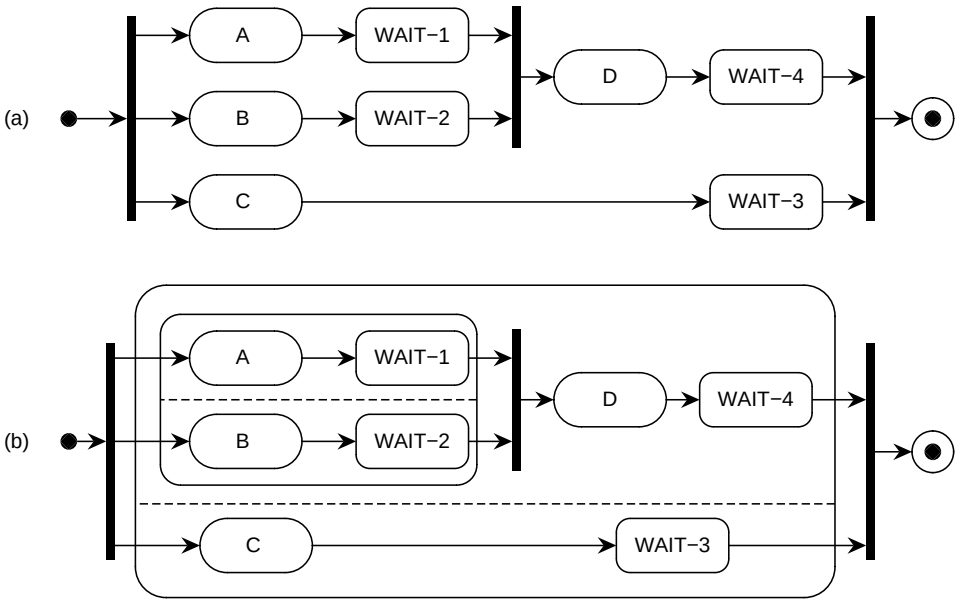



**Figure 9.4** *Activity diagram that can be translated into a statechart*

in terms of activity hypergraphs. We allow all the activity diagrams shown in this section. To allow multiple simultaneous instantiations of the same nodes, we have extended the statechart definition of configuration and step from sets of nodes and sets of hyperedges to bags of nodes and bags of hyperedges.

STATEMATE **and UML statechart semantics.** Harel [88] introduced statecharts as part of STATEMATE [92], a structured analysis approach for modelling reactive systems. The two most important specification techniques in STATEMATE are statecharts and activity charts. Activity charts describe the functional view of the system. An activity chart consists of activities and the dataflows between them. The behaviour of every activity is modelled with a statechart. A statechart can start and stop activities. Atomic (non-compound) activities themselves are specified imperatively, not declaratively.

The STATEMATE semantics of statecharts assumes that reactions do not take time (perfect technology). There are two variants of the semantics. One variant is event-driven (called asynchronous in STATEMATE), the other one is clock-driven (called synchronous in STATEMATE). The event-driven variant satisfies the perfect synchrony hypothesis.

The semantics of UML statecharts does not resemble the STATEMATE semantics at all, since the perfect technology assumption is dropped and events are processed

one by one. The UML semantics does not make the perfect synchrony hypothesis. Instead, the UML semantics is based on the ROOM statechart semantics [146]. We have given elsewhere a detailed comparison of UML statecharts with STATEMATE statecharts [61, 62]. A UML statechart can model the behaviour of any software item, but it is mostly used to model the behaviour of a software object.

Although our two semantics are based on elements from the STATEMATE and UML statechart semantics, there are some differences:

- In our semantics, activities fall outside the scope of the system that is being developed, whereas in both STATEMATE and UML activities fall inside that scope. Activities in STATEMATE and UML are always software activities, whereas in our semantics activities might be manual.
- Consequently, we specify activities declaratively and incompletely with pre and postconditions, whereas in STATEMATE and UML they are specified imperatively with a software procedure or another statechart.
- We have defined data integrity constraints between activities. Such constraints do not exist in STATEMATE or UML.

Thus, the major difference with both the STATEMATE and UML statechart semantics is that in our semantics activities are executed in and by the environment, whereas in STATEMATE and UML they are executed in and by the system being specified.

## 9.2 OMG semantics of UML activity diagrams

**UML 1.4.** In version 1.4 of UML [150], current at the time of writing (2002), the semantics of an activity diagram is specified in terms of a UML statechart, by translating a UML activity diagram into a UML statechart. (In UML 2.0, the semantics will be defined independently from statecharts.) Apart from the fact that not every activity diagram can be translated into a statechart (see the previous section), the UML 1.4 semantics of activity diagrams is not entirely suitable for workflow modelling. For example, in UML 1.4 an activity is defined as an entry action of a state. An entry action is executed to completion when its state is entered [150]. But in Figure 1.1 this means that the two activities *Check stock* and *Check customer* are executed simultaneously in the same run-to-completion step! This is not what we would like the activity diagram of Figure 1.1 to say. What we would like to express by Figure 1.1 is that *Check stock* and *Check customer* start simultaneously, not that they should stop at the same time.

The underlying problem is that in UML 1.4, and also in the UML 2.0 proposal [11], an activity diagram is viewed as model of a software system that executes the activities itself. We want to use activity diagrams for workflow modelling and therefore see an activity diagram as a model of a workflow system (WFS). In

workflows, the activities are performed by actors (people or applications) external to the WFS, not by the WFS itself. It is the task of the WFS to monitor these activities, to manage the flow of data between them, and to route work items through a collection of actors, but it is not the task of the WFS to *execute* the activities. So in Figure 1.1, the WFS executes the state transitions, i.e. the arrows in the diagram. The activities (nodes in the diagram) are executed by actors external to the WFS.

So, in our semantics, activity states are states in which the WFS waits for an actor to finish work. Transitions are steps in which the WFS records the completion of activities and the arrival of events, and computes what activities should be done next.

**UML 2.0.** Currently (2002) the UML 2.0 is under development. Several proposal have been sent in. The most comprehensive one, the one by the UML Revision Task Force (RTF) itself [11], presents a semantics to activity diagrams that is completely different from the UML 1.4 semantics.

The proposal talks about tokens flowing along edges. Clearly, this suggests a Petri net-like semantics. Unfortunately, some constructs are introduced that cannot be expressed in Petri net token-game semantics, even though this is suggested by the informal semantics of the UML RTF proposal [11].

For example, the notion of an interrupt region is introduced (see also Section 7.4). An interrupt region is a group of nodes that can be interrupted by one or more special edges, called interrupting edges. If one of these edges is taken, all other tokens flowing in the region are aborted. The semantics of of the interrupt region is given by the following two lines [11]:

> The region is interrupted when a token traverses an interrupting edge. At this point the interrupting token has left the region and is not terminated.

An interrupt has two important features that are missing from above description:

- All other tokens in the region should be removed (aborted).
- The interrupt edge must have higher priority than a non-interrupt edge, i.e., it must be taken first.

Both features cannot be modelled with the standard token-game semantics of Petri nets. The first feature can be modelled by interpreting an interrupt region as a shorthand to abbreviate many edges (see Section 7.4), whereas the second feature could in principle be modelled by using priority transitions from the theory of stochastic Petri nets.

More in general, the token-game semantics is not suitable to represent reactive behaviour, as we showed in Chapter 8. Thus, a token-game semantics for activity diagrams would make activity diagrams unsuitable for modelling reactive systems.

**Are activity diagrams OO?** Some members of the UML Revision Task Force (RTF) do not seem to be comfortable with activity diagrams, since, as they argue, these do not fit the paradigm of object orientation. As an illustration, we quote from a recent report by Kleppe and Warmer [111] (Warmer is member of the UML RTF):

> The way the activity diagram in the Unified Modeling Language is currently defined is not object-oriented. This is a bold statement, but it is backed up by experts in the field. In a recent presentation for the OMG, Conrad Bock mentioned: an "application is completely OO when all action states invoke operations, and all activity diagrams are methods for operations." [...] In other words, using an activity diagram one can model a system in a completely non object-oriented way. In such an activity diagram the object-oriented principle of responsibility is not applied.
>
> In fact, the current activity diagrams look in suspiciously many ways like the results of structured analysis and design, which uses functional decomposition to develop a software system. It is well known to object-oriented experts that structured analysis and design is a modeling paradigm that does not fit to the object-oriented paradigm. In the presentation for the OMG mentioned earlier [...] it is stated that activity diagrams specify data/object flow. At the same time the UML 1.3 standard [...] includes the following quote [...]: "Why does not UML support data-flow diagrams? Simply put, data-flow and other diagram types that were not included in the UML do not fit as cleanly into a consistent object-oriented paradigm."

Consequently, Warmer and other advocates of this viewpoint argue that activity diagrams should be thrown out of the UML, as they are not OO. Others take a more pragmatic viewpoint. We quote Bock in one of his papers [21] (Bock is also member of the UML RTF and leader of the group responsible for the syntax and semantics of activity diagrams):

> Another problem with UML [...] is that [...] the user is forced to assign the business function to an object [...] This may be gospel to object-orientation practitioners, but business modelers are aware that the responsibility for an activity may change over time, and consequently prefer to focus their models initially on the result that is expected from a function rather than who performs it [...]. This is related to the concept of interfaces adopted in the object orientation community, but more powerful. Business modelers are well in advance of object-oriented modelers in this respect.

The inevitable conclusion is that OO is not really suitable for business modelling. Inspection of some works on applying UML to business modelling conforms this conclusion [59]. Software objects are used passively in order to store information, not actively to process information. This explains why activity diagrams are useful for workflow modelling but also why they are not OO. Dropping activity diagrams from the UML implies dropping business modelling from the UML.

## 9.3 Other work on UML activity diagrams

Dumas and Hofstede [53] evaluate the suitability of UML activity diagrams for workflow modelling by trying to capture some workflow patterns [7] in activity diagrams. They only consider activity diagrams that are translated into statecharts according to OMG semantics of UML 1.4 [150].

There are several other formalisations of the UML 1.4 (and of earlier versions) of activity diagrams [13, 23, 25, 73, 137]. Gehrke et al. [73] propose to give a semantics to activity diagrams by translating them into Petri nets, but do not provide a formal semantics. They do not relate the proposed Petri net semantics to the OMG semantics of activity diagrams. See Chapter 8 for a comparison of both our semantics with Petri nets.

The other formalisations [13, 23, 25, 137] follow the OMG semantics closely. The discussion in Section 9.2 listing the difference of our semantics with the OMG semantics also applies to these other formalisations. Like the OMG semantics, these formalisations map activities into actions that are done in transitions of the system under development. Consequently, they too have problems modelling parallelism (see Section 9.2). The formalisations are not as complete as ours; for example none of them deals with events, none of them deals with the in predicate, none of them deals with wait nodes.

It is quite interesting to see the approach taken by these other formalisations. All of them give a semantics by defining a mapping from an activity diagram into the syntax of another formal technique, for example LOTOS [22] in the case of Apvrille et al. [13] and Pinheiro da Silva [137], or CSP [97] in the case of Bolton and Davies [23].

Implicitly, in these approaches the assumption is made that the semantic choices made in these formal techniques are valid for UML activity diagrams as well. Although this might be true, we think that choices made in the semantics should be made explicit, so that they can be validated for the intended domain of modelling, in this case workflow modelling. Unfortunately, none of the authors mentioned above seems to be aware of this problem. In none of these references the semantic choices are stated explicitly. Even if one thinks these semantic choices are valid for the domain being modelled, this should be brought out in the open and motivated. This is not done by the authors just mentioned.

To show the impact of these hidden assumptions, we give three examples of hidden assumptions that are implicitly made in the approaches above, and we validate these hidden assumption against the domain of workflow modelling. The first example concerns the semantics of activities. In the process algebraic formalisations of UML activity diagrams [13, 23, 137], activities are modelled as actions. In every process algebra, including CSP and LOTOS, an action does not take time. Also, actions are done in transitions of the system being modelled, not in states. Actions, including parallel ones, are interleaved: only one action is done at a time. This is justified in process algebras by the fact that an action is instantaneous.

Applying the process-algebraic formalisation to our running example (Figure 1.1), we thus have that activities like *Check stock* and *Check customer* do not take time. Moreover, activities *Check stock* and *Check customer* that are specified to be in parallel according to the specification, are done sequentially, i.e. one by one, in the semantics! Of course, a more fine grained mapping could be defined, in which an activity is mapped into a begin and an end action, but this is not done in these formalisations.

As a second example, one of the key features of process algebras like LOTOS and CSP is the definition of an equality relation on processes. Figure 9.5 shows two activity diagrams that are the same according to the formal semantics of Apvrille et al. [13], Pinheiro da Silva [137], and Bolton and Davies [23]. (We do not use wait nodes as these are not formalised by these references.) But the two activity diagrams are different according to our two semantics, because they have different runs. We think most workflow designers (and most UML designers) would consider them different as well.

The third example concerns the semantics of communication. In process algebra, communication between two parties is blocking in the sense that both parties cannot proceed unless they cooperate with each other. In other words, communication is synchronous. Translating this to workflow specifications, this would mean that the environment of a WFS must run in the same pace as the WFS itself. The environment might even get blocked by the WFS, if the WFS is too slow. It is very doubtful whether this is appropriate for WFSs. We do not want to put such a constraint upon the environment of the WFS. We therefore have adopted a non-blocking semantics (see Section 4.3 on page 47).

We conclude that in these other formalisations of UML activity diagrams, some of the semantic choices that are implicitly made do not match the domain of workflow modelling. Thus, these semantics are not fit to be used for workflow modelling.

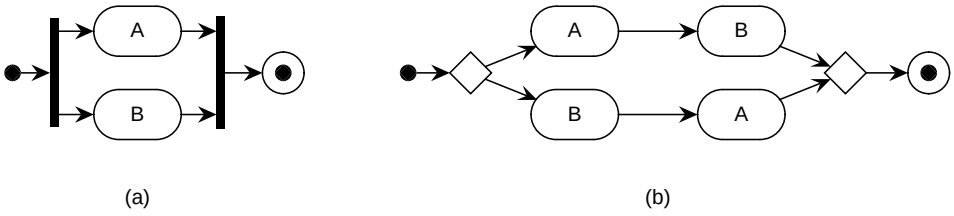



**Figure 9.5** *Two activity diagrams that are the same according to [13, 23, 137], but not according to our two semantics*

## 9.4 The state of the practice

By defining a semantics for activity diagrams and linking it with WFSs, we have implicitly defined how a WFS behaves. We now discuss the current state of the art w.r.t. the semantics outlined above. So, is our semantics realistic?

We emphasise that it is not our aim to define a semantics that is implementable by any state of the art WFMS (see page 8). As WFMS product vendors do not publish the semantics their WFMSs attach to a workflow specification, it is impossible to validate whether the semantics is implementable by their WFMSs. Instead, our aim is to define a good approximation of how a WFMS in general behaves.

We do not know of any commercial WFMS that allows for the specification of workflow models using UML activity diagrams. But few of the current commercial workflow systems offer some support for modelling events [30, 84]. We therefore expect that the constructs of our semantics related to events will be hard to express in workflow models of existing commercial WFMSs. On the other hand, our event broadcast semantics, in which one event can trigger more one edge, is similar to the publish-subscribe notification mechanism used in middleware applications, and also used in a recently adopted industry standard for workflow interoperability [133], defined by OMG and WFMC.

Recently, UML activity diagrams have been proposed to model e-business services in e-business standards like ebXML [153]. We expect that process management tools that support e-business services will use UML activity diagrams as specification language. In for example ebXML, the event features of activity diagrams are used quite extensively: events are the standard means of communication between different business partners. Events are also used quite extensively in business modelling, especially in UML-based approaches, for example [59]. In academia, several WFMS research prototypes use event-based workflow models (e.g. [30, 45, 77, 84, 85, 130]), often inspired by active databases [156] (see Section 9.6).

## 9.5 Other workflow modelling languages

There are several languages for modelling workflows, mostly informal ones. We here focus on languages that have some kind of execution semantics.

Event-driven process chains (EPCs) are part of the ARIS (Architecture of Integrated Information Systems) method [145]. EPCs are inspired by Petri nets, but they do not have a formal semantics. Nüttgens et al. [131] give a brief comparison of EPCs and UML activity diagrams.

Leymann and Roller [118] define a language that is close to the language used in IBM's workflow product [100]. The syntax does not resemble UML activity diagrams, as workflow specifications in their approach must be acyclic. The language does not have decision or merge nodes. The semantics of that language has

single-event processing. Each activity has its own copy of data it needs, whereas in our approach data is global.

Jackson and Twaddle [103] define a language for modelling workflows. The main difference with activity diagrams is that in their language an edge between two activities, say from $A$ to $B$, denotes "begin on start", so $B$ can begin when $A$ has begun. Whereas in activity diagrams an edge means "begin on end", so $A$ should have terminated when $B$ begins. Their language does not allow for multiple simultaneous instantiations of the same task.

Ould [135] uses a variant of Petri nets with some additional notation to model and animate business processes. The focus is on organisational roles that do activities and on how these roles cooperate. The semantics of the used language is not formally defined.

AMBER [54, 104] is a language for modelling businesses processes. AMBER stands for Architectural Modelling Box for Enterprise Redesign. Janssen et al. [104] define the semantics of AMBER in terms of the input language of the Spin model checker. Like statecharts, AMBER models activities as actions that are executed by the system under development, not by the environment. Models in AMBER are by definition safe. AMBER offers some support for modelling data.

Finally, in the Mentor project [130, 158] the STATEMATE toolset is used for workflow modelling, even though the STATEMATE toolset is intended for modelling embedded real-time systems. Wodtke and Weikum [158] use the clock-driven semantics, in which the system takes a step at every tick of the clock. They say that the event-driven semantics, with its supersteps, is too complex for workflow modelling, but they do not motivate this any further. Interpreting their approach in terms of our model, they seem to have a hybrid of our requirements-level and implementation-level semantics: an implementation-level like semantics with parallel event-processing triggered by ticks of the clock.

## 9.6 Active databases

In active database systems [156] an execution semantics for active rules is adopted that resembles the semantics for edges that we use. The major difference, however, is that active rules are typically executed in transactions. Whereas an edge in an activity diagram is typically taken when some activity terminates and the corresponding database transaction has finished as well. Also, the notion of state (node) is absent in active rules. These differences make it hard to compare active rules with our execution semantics.

Nevertheless, there are some striking similarities with the statechart semantics. For example, like in statecharts, in active databases a generated event has an effect in the next step, not in the current step. The possibility of nontermination of the rule processing algorithm due to rules that trigger each other, is a well known feature of active databases [156]. Nontermination resembles divergence

in statecharts (see page 50). Furthermore, Ceri and Fraternali [33] combine the syntax of statecharts with active rules: they use statecharts as a graphical front end for an active rule language, Chimera. Unfortunately, they do not relate the active rule language to the statechart semantics. We do not know of any work comparing active rules to statechart semantics.

## 9.7 Transactional workflows

Transactional workflow modelling focuses on the specification of transactional properties for workflows or parts of workflows [76]. Transactional workflow models can be at a higher or lower level of abstraction than the workflow models we use [79], but usually they are at a lower level. For example, usually in a transactional workflows, an activity (task) can have several substates, including several end states, such as aborted and committed. These substates correspond to the states of a transaction. If an activity is aborted, it has no effect on the case attributes, that is, it seems as if it was never executed. If an activity is committed, it does have an effect on the case attributes. Between activities (tasks), several dependencies can be modelled, for example, activity $B$ should begin executing if activity $A$ has aborted (begin-on-abort) [8, 15]. Figure 9.6 shows the state-transition diagrams of $A$ and $B$ and the begin-on-abort dependency between them. Labels on edges denote events.

By contrast, in our semantics, an activity state has just one state, in which the WFS waits for the activity to terminate. Once the activity terminates, the WFS leaves the activity state, thus finishing the activity. Moreover, the only kind of dependency between activities we have is that if one activity finishes, the next one starts.

Despite these differences, some of the constructs that are used in transactional workflows, for example the different end states, can be modelled in activity diagrams by encoding the end state of the activity as a separate variable. By testing this variable once the activity has completed, several of the dependencies used in transactional workflows can be modelled. For example, Figure 9.7 shows how the begin-on-abort dependency in Figure 9.6 can be modelled in activity diagrams.

## 9.8 Conclusion

We have discussed several other semantics for activity diagrams. These other semantics, including the informal OMG semantics, are not motivated by a particular application domain. We showed that they are not suitable for modelling workflows.

A merit of our approach compared to other approaches is that the design choices that we made in both our semantics are stated explicitly. In other approaches, design choices are not explicitly listed; rather, these approaches use an existing formal notation and do not mention the design choices that are made

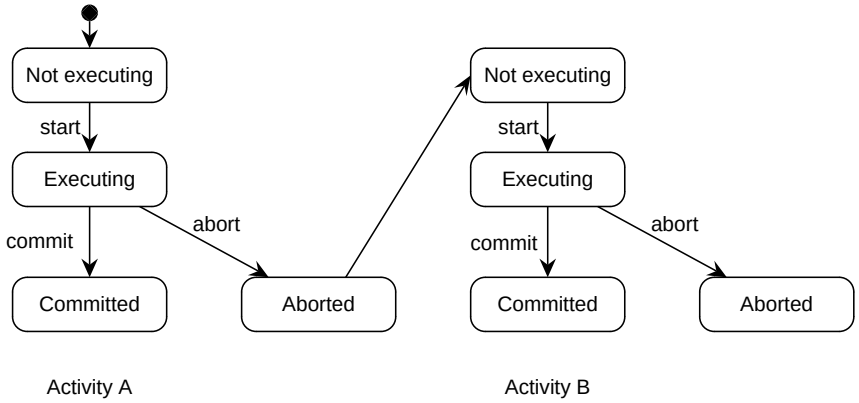



**Figure 9.6** *Transactional workflow specification with begin-on-abort dependency*

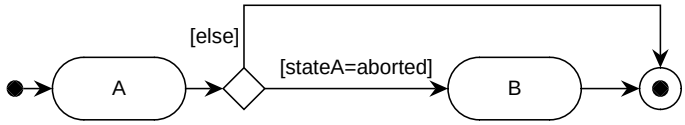



**Figure 9.7** *Activity diagram with begin-on-abort dependency*

in the semantics of these notations. Our semantics can be applied to any other application domain by just validating the choices against that domain. UML activity diagrams are also used for modelling the flow of control within a software procedure. We do not think that our semantics is suitable for this domain. In this domain, activities are done inside the scope of the system being modelled, whereas in workflow modelling they are outside the scope of the WFS.

Finally, we have shown with our semantics that a statechart-like semantics can be given to a notation with a Petri net-like syntax. As explained in Section 8.7, for UML 2.0 a Petri net-like semantics for activity diagrams seems to have been chosen because the statechart syntax was considered to be too restrictive for modelling concurrency. Our approach does not have this restriction, because we have defined a semantics directly in terms of activity diagrams. Unlike the UML 2.0 proposal, we still use a statechart-like reactive semantics.

# Chapter 10

# Verification of functional requirements

In this chapter we explain how functional requirements of activity diagrams can be verified using model checking [42]. Model checking is a technique for automatically verifying whether a finite transition system (Kripke structure) $K$ satisfies a temporal logic formula $\varphi$, i.e., whether $K \models \varphi$. The language we use for specifying functional requirements on workflows is therefore based on temporal logic [57]. In the next chapter we will discuss some case studies that we did, in which we model checked functional requirements of activity diagrams.

We have developed the following tool support for model checking. We have implemented the requirements-level semantics in the Toolkit for Conceptual Modeling (TCM) [48], a set of diagram editing tools, one of which is a tool for drawing activity diagrams. We have interfaced TCM with a model checker. Figure 10.1 shows the architecture of the tool that we have developed. The mapping of the activity diagram into the transition system implements the requirements-level execution semantics of Chapter 5.

The intended way of working with the verification tool is as follows. The workflow modeller specifies an activity diagram with TCM. This activity diagram is a workflow specification. Using a formal property language, the workflow modeller defines requirements that the intended workflow specification must satisfy. Note that both the activity diagram and the requirements must be formal, since they are interpreted by a software tool according to a formal semantics.

TCM generates a transition system from the activity diagram and translates the requirement into a temporal logic formula. Currently, the requirements language is a syntactic sugarring of the temporal logic language. In future work we intend to specify a more abstract requirements language that is closer to the business level. The most commonly used temporal logics are Linear Temporal Logic (LTL) [122] and Computation Tree Logic (CTL) [42]. These logics are explained

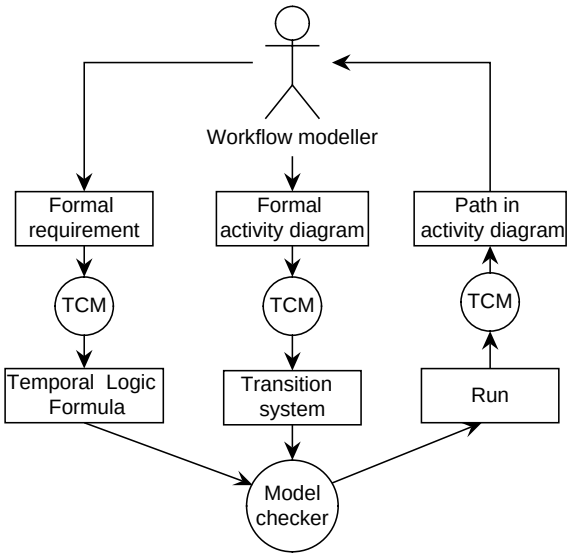



**Figure 10.1** *Tool architecture*

and defined in Section 10.1.

Both the transition system and the temporal logic formula are input for a model checker that checks whether the transition system satisfies the temporal logic formula. If the temporal logic formula fails to hold, the model checker generates an example run (also known as scenario or trace) which shows the sequence of states that lead to violation of the requirement. TCM then highlights the corresponding path in the activity diagram. The workflow modeller can then either change the requirement or the activity diagram and do the verification again.

Initially, we included support for as many model checkers as possible and have used for example NuSMV [38], Spin [99] and Kronos [163]. However, not every model checker turned out to be useful. As we will argue in Section 10.3, only model checkers supporting *strong fairness* constraints are useful for workflow specifications. Since strong fairness is an LTL property and most model checkers support CTL properties only, only a few model checkers are useful for our purpose. Even worse, most LTL model checkers do not use a special model checking algorithm for strong fairness, i.e., they do not support verification of strong fairness at the algorithmic level. Consequently, their performance is so bad that they cannot be used either. We therefore decided to implement in the NuSMV [38] model checker an existing LTL model checking algorithm of Kesten et al. [110] that deals with strong fairness constraints at the algorithmic level. Section 10.3 gives more details.

**Note.** In this chapter we restrict ourselves to the class of activity diagrams that have similar behaviour in RLS and ILS, as defined in Chapter 6. So the considered activity diagrams satisfy the constraints mentioned in Theorem 6.7 on page 101. This restriction can easily be relaxed. We will focus on requirements that are insensitive to the semantics being used, RLS or ILS.

The structure of this chapter is as follows. Section 10.1 defines the logic CTL$^*$. Both CTL and LTL are restricted logics of CTL$^*$. We also define CTL$^*_r$, a restricted version of CTL$^*$. Formulas in CTL$^*_r$ are insensitive to the semantics being used, RLS or ILS. Model checking requires the state space to be finite. Section 10.2 explains how a Clocked Transition System with an infinite state space is transformed into a finite transition system without clocks. Section 10.3 discusses what strong fairness is and why it is needed. Section 10.4 sketches the structure of our implementation in TCM. Section 10.5 gives some example verifications of requirements. Section 10.6 first analyses how parallelism, events, data, and real-time impact the size of the state space. Then it shows how the state space can be reduced while preserving the requirement to be verified. Section 10.7 discusses related work. We end with conclusions.

## 10.1 Temporal logic

**CTL$^*$.** The presentation in this paragraph is based on [36, 41]. CTL$^*$ can express both linear-time and branching-time properties. The following linear-time operators are used:

$\mathsf{X}$ – neXt  $\mathsf{U}$ – Until  $\mathsf{F}$ – sometimes (Future)  $\mathsf{G}$ – always (Global)

Linear-time operators are evaluated on infinite paths. A path is a sequence of states. In an infinite path, each state has a single successor. Operator $\mathsf{X}$ is the next-time operator: $\mathsf{X}\,\varphi$ is true iff in the next state $\varphi$ is true. Operator $\mathsf{U}$ is the until operator: $\varphi_1\,\mathsf{U}\,\varphi_2$ is true iff from now on, in the current state, $\varphi_1$ holds until in a certain state in the future $\varphi_2$ holds. Operator $\mathsf{F}$ specifies that sometime in the future a formula will hold: $\mathsf{F}\,\varphi$ abbreviates $\mathsf{true}\,\mathsf{U}\,\varphi$. Operator $\mathsf{G}$ specifies that always from now on, so globally, a formula holds: $\mathsf{G}\,\varphi$ abbreviates $\neg\,\mathsf{F}\neg\,\varphi$ which is equal to $\neg\,(\mathsf{true}\,\mathsf{U}\neg\,\varphi)$.

In CTL$^*$, a formula composed of linear-time operators can be prefixed by path quantifiers. Path quantifiers are branching-time operators. The following path quantifiers are used.

$\mathsf{A}$ – for All paths  $\mathsf{E}$ – for some paths (Exists)

Let $AP$ denote the set of atomic propositions. We define $AP$ below. There are two types of formulas in CTL$^*$: state formulas (which are true in a specific state) and path formulas (which are true along a specific path). We inductively define the class of state formulas and path formulas:

- An atomic proposition $ap \in AP$ is a state formula.
- If $p$, $q$ are state formulas then so are $p \wedge q$, $\neg\, p$.
- If $p$ is a path formula then $\mathsf{E}\, p$, $\mathsf{A}\, p$ are state formulas.

The path formulas are specified by the following inductive definition

- If $p$ is a state formula, it is also a path formula.
- If $p$, $q$ are path formulas then so are $p \wedge q$, $\neg\, p$, $\mathsf{X}\, p$ and $p\, \mathsf{U}\, q$.

CTL$^*$ is the set of state formulas defined by above formulas.

Both state formulas and path formulas are evaluated with respect to a Clocked Transition System (CTS) as defined in Chapter 5. State formulas are evaluated in states (valuations) of the CTS, whereas path formulas are evaluated in paths of the CTS. Let $\sigma$ denote an arbitrary state (valuation) and $\pi = \sigma_0, \sigma_1, \ldots$ a path (recall from Section 5.1 that a path is an infinite sequence of states (valuations) such that for every $i \geq 0$, $\sigma_i \rightarrow \sigma_{i+1}$). We denote by $\pi^i$ the suffix of $\pi$ starting at $\sigma_i$.

The satisfaction relation $\models$ is defined inductively as follows, where $s$ denotes a state formula and $p$ a path formula. We assume given the definition of the satisfaction relation for atomic propositions in valuations.

$$
\begin{array}{lcl}
\sigma \models \neg\, s & \Leftrightarrow & \sigma \not\models s \\
\sigma \models s_1 \wedge s_2 & \Leftrightarrow & \sigma \models s_1 \text{ and } \sigma \models s_2 \\
\sigma \models \mathsf{A}\, p & \Leftrightarrow & \text{for all paths } \pi \text{ starting with } \sigma,\ \pi \models p \\
\sigma \models \mathsf{E}\, p & \Leftrightarrow & \text{there exists a path } \pi \text{ starting with } \sigma \text{ such that } \pi \models p \\
\pi \models \neg\, p & \Leftrightarrow & \pi \not\models p \\
\pi \models p_1 \wedge p_2 & \Leftrightarrow & \pi \models p_1 \text{ and } \pi \models p_2 \\
\pi \models \mathsf{X}\, p & \Leftrightarrow & \pi^1 \models p \\
\pi \models p_1\, \mathsf{U}\, p_2 & \Leftrightarrow & \text{for some } k \geq 0, \pi^k \models p_2, \\
 & & \text{and for every } 0 \leq i < k, \pi^i \models p_1
\end{array}
$$

The abbreviations $\mathsf{true}$, $\mathsf{false}$, $\vee$, $\Rightarrow$, etc. are defined as usual. Note that $\mathsf{A}\, s$ is equivalent to $\neg\, \mathsf{E} \neg\, s$.

**Atomic propositions.** We focus on propositions that are defined in both the RLS and ILS. An atomic proposition $ap \in AP$ is either:

- a test on the configuration, $b \sqsubseteq C$, where $b$ is some bag of nodes, or
- some boolean expression on local variables, or
- predicate *stable* (defined for the RLS on page 65 and for the ILS on page 101).

We use predicate $\mathsf{in}(n)$ as an abbreviation of $[n] \subseteq C$. Events cannot be referred to because of Theorem 6.2(ii): in the RLS sometimes some extra events are needed to get the same effect as in the ILS. Atomic propositions can only be evaluated in stable states, i.e., we assume every atomic proposition is implicitly conjoined with predicate *stable*. This ensures that no inconsistent values of a variable are read, for example of a configuration in the ILS during the taking of a step.

Since no two stable states follow each other in both the RLS and ILS, it does not make sense to use the next time $\mathsf{X}$ operator. From now on, we will not use the $\mathsf{X}$ anymore.

✓ The next time $\mathsf{X}$ operator is not used.

**CTL and LTL.** Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) are restricted subsets of CTL$^*$.

The logic CTL is obtained by replacing the above definition of path formulas by the following definition:

- If $p$, $q$ are state formulas then $\mathsf{X}\,p$ and $p\,\mathsf{U}\,q$ are path formulas.

This definition enforces linear-time operators to be immediately preceded by a path quantifiers. There are eight basic CTL operators: $\mathsf{AX}$, $\mathsf{EX}$, $\mathsf{AG}$, $\mathsf{EG}$, $\mathsf{AF}$, $\mathsf{EF}$, $\mathsf{AU}$, and $\mathsf{EU}$.

Linear Temporal Logic (LTL) consists of the class of formulas of the form $\mathsf{A}\,p$ where $p$ is path formula in which the only state formulas are atomic propositions. The syntax of path formulas in LTL is defined as follows:

- Every atomic proposition is a path formula.
- If $p$, $q$ are path formulas then $\neg\, p$, $p \wedge q$, $\mathsf{X}\,p$, and $p\,\mathsf{U}\,q$ are path formulas.

In writing LTL formulas, we use the standard convention that the $\mathsf{A}$ quantifier is omitted. So formula $\mathsf{F\,G}\,p$ abbreviates $\mathsf{A\,F\,G}\,p$.

**Expressiveness.** Finally, we compare the expressiveness of CTL and LTL [39, 154]. It is obvious that CTL formulas that do not start with a for all paths $\mathsf{A}$ quantifier, for example $\mathsf{EG}\,p$ ("there is path along which $p$ is always true"), are not expressible in LTL. Moreover, in LTL existential path quantifiers cannot be used at all. For example, $\mathsf{AG}\,(p \Rightarrow \mathsf{EF}\,q)$ is not expressible in LTL. So possibility properties, which involve using the $\mathsf{E}$ quantifier, cannot be specified in LTL.

It might therefore seem as if every CTL formula that only contains $\mathsf{A}$ quantifiers is expressible in LTL. But this is not true. For example, CTL formula $\mathsf{AF\,AG}\,p$ and LTL formula $\mathsf{F\,G}\,p$ are not equivalent [39]. To illustrate this, consider the Kripke structure $K$ in the left hand side of Figure 10.2. $K$ satisfies LTL formula $\mathsf{F\,G}\,p$, so $K \models \mathsf{F\,G}\,p$ but $K$ does not satisfy CTL formula $\mathsf{AF\,AG}\,p$, so $K \not\models \mathsf{AF\,AG}\,p$. To

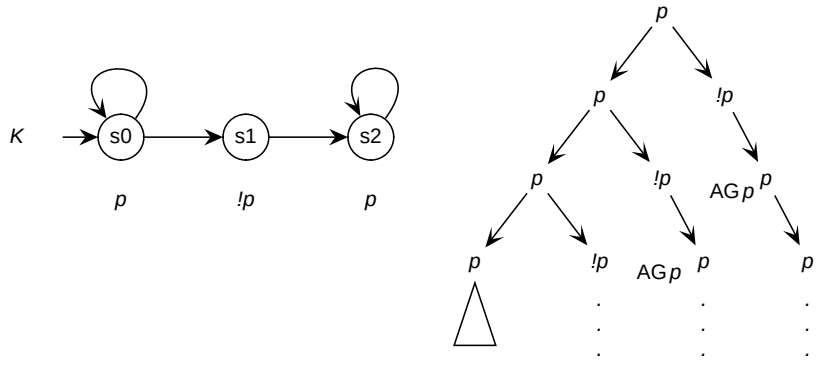



**Figure 10.2** *Kripke structure and corresponding computation tree. '!' denotes* $\neg$

see this, consider the paths of $K$, shown in the right hand side of Figure 10.2. Formula $\mathsf{F}\,\mathsf{G}\,p$ is true because in every path, eventually $p$ will stay true forever. Formula $\mathsf{AF}\,\mathsf{AG}\,p$ is not true, because there is a path, namely $s_0, s_0, ..$ in which it is possible that in a future state $p$ becomes false, namely if $s_1$ is entered. Clarke and Draghicescu proved that property $\mathsf{F}\,\mathsf{G}\,p$ cannot expressed in CTL [39]. So CTL and LTL are incomparable w.r.t. expressiveness.

**A restricted temporal logic: CTL$^*_r$.** We define a restricted version of CTL$^*$, called CTL$^*_r$. The reason for defining this restriction is that in Chapter 6 we have seen that the CTSs induced by the RLS and ILS do have stable valuations in common, but differ in how they reach these stable valuations. If we allowed unrestricted CTL$^*$ formulas violating above rules, the formulas could detect these differences. The restricted formulas cannot detect these differences. Thus, formulas in CTL$^*_r$ are insensitive to the semantics being used, RLS or ILS.

We replace the inductive definition of CTL$^*$ path formulas by the following definition.

- If $p$ is a state formula, it is also a path formula.
- If $p$, $q$ are path formulas then so are $p \wedge q$, $\neg\, p$, and $\mathsf{true}\,\mathsf{U}\,p$.
- If $p$ is a path formula, such that either $p$ is not a state formula or $p$ is equivalent to $\mathsf{true}$, then $\mathsf{false}\,\mathsf{U}\,p$ is also a path formula.

The $\mathsf{X}$ operator was already ruled out above, because atomic propositions are only evaluated in stable states and no two stable states follow each other immediately. The use of the until operator $\mathsf{U}$ is restricted for the same reason. The second and third rule forbid certain occurrences of $\mathsf{U}$ that require a continuous

evaluation of some atomic proposition. Thus, if $p$ is a state formula not equivalent to $\mathsf{true}$, for example $\mathsf{G}\, p$ and $p\, \mathsf{U}\, \varphi$ are forbidden, where $\varphi$ is an arbitrary $\mathrm{CTL}_r^*$ formula.

We now are able to prove the following theorem for $\mathrm{CTL}_r^*$ formulas. Let $R$ be a stable simulation relation as defined in Section 6.2.4.

**Theorem 10.1** *Given two valuations $\sigma, \sigma'$ such that $\sigma \underline{R}\, \sigma'$. Let $\varphi$ be an arbitrary $CTL_r^*$ formula. Then $\sigma \models \varphi \;\Leftrightarrow\; \sigma' \models \varphi$.*

*Proof.* By induction on the structure of $\mathrm{CTL}_r^*$ formulas, we prove that valuations $\sigma, \sigma'$ related by $R$ satisfy the same $\mathrm{CTL}_r^*$ formulas.

Most cases follow immediately from the definition of $R$, Theorem 6.7, and the induction hypothesis. We only treat the basic $\mathsf{U}$ case here. Consider path formula $\mathsf{true}\, \mathsf{U}\, p$ where $p$ is an arbitrary state formula, possibly equivalent to $\mathsf{true}$.

Consider an arbitrary path $\pi$ starting at $\sigma$ such that $\pi \models \mathsf{true}\, \mathsf{U}\, p$. We show that there is a path $\pi'$ starting at $\sigma'$ such that $\pi' \models \mathsf{true}\, \mathsf{U}\, p$.

Since $\pi \models \mathsf{true}\, \mathsf{U}\, p$, there is some future state $\sigma_k$ where $k \geq 0$ such that $\pi = \sigma \rightarrow \sigma_1 \ldots \rightarrow \sigma_k \rightarrow \ldots$ and $\sigma_k \models p$. By definition of atomic propositions, we know that $\sigma_k \models \mathit{stable}$. By definition of $R$, there is a $\sigma'_l$ such that $\sigma' \rightarrow \sigma'_1 \ldots \rightarrow \sigma'_l$ where $l \geq 0$ such that $\sigma_k \underline{R}\, \sigma'_l$ and $\sigma'_l \models p$. Let $\pi'$ be an arbitrary path with prefix $\sigma' \rightarrow \sigma'_1 \ldots \rightarrow \sigma'_l$. Clearly, $\pi' \models \mathsf{true}\, \mathsf{U}\, p$.

By similar reasoning, it can be shown that for every path $\pi'$ starting at $\sigma'$ such that $\pi' \models \mathsf{true}\, \mathsf{U}\, p$, there is a path $\pi$ starting at $\sigma$ such that $\pi \models \mathsf{true}\, \mathsf{U}\, p$.

Other induction cases can be proven by similar reasoning.

□

We now prove our main result, that formulas in $\mathrm{CTL}_r^*$ are insensitive to the semantics being used, RLS or ILS.

**Theorem 10.2** *Given an activity hypergraph $AH$. Denote by $CTS^{\mathsf{RLS}}$ the CTS induced by $AH$ under the RLS. Denote by $CTS^{\mathsf{ILS}}$ the CTS induced by $AH$ under the ILS. Let $\varphi$ be an arbitrary $CTL_r^*$ formula.*

*Then*

$$\sigma_{init}^{\mathsf{RLS}} \models \varphi \;\Leftrightarrow\; \sigma_{init}^{\mathsf{ILS}} \models \varphi$$

*Proof.* Both initial states lead to stable states that are related via $R$. Now apply Theorem 10.1 on these states. □

From now on, we will restrict ourselves to formulas in $\mathrm{CTL}_r^*$.

**Some final remarks.** The attentive reader may be surprised that events cannot be referred to in our property language, even though we did a hard job defining a reactive semantics which involves event-driven behaviour. Is it not a serious restriction that events cannot be referred to?

The main observation is that the semantics of events is different in both semantics. We give two examples. First, Theorem 6.2 shows that in the ILS, external non-termination events can trigger some extra hyperedges that are not triggered in the RLS; this was discussed already in Section 6.1.1. Therefore, events should not be referred to in $\mathrm{CTL}^*_r$. Of course, event expressions could be added to the property language, but then the model checking results no longer carry over to the ILS.

Second, in the RLS, due to the perfect synchrony hypothesis, an event is processed at the same time as it occurs, whereas in the ILS it is processed at some later time. Therefore, if in the RLS an event $e$ occurs in a certain configuration $C$, then the effect of $e$ follows immediately. In the ILS, if $e$ occurs in $C$ then all events still in the queue must be processed before the effect of $e$ becomes clear. This difference in semantics can be easily detected by writing a property that refers to events.

Similarly, we do not have an operator $\mathsf{X}_{\mathsf{stable}}$ that, given a certain stable state, refers to the next stable state. As shown by Theorem 6.7(ii), although the RLS and the ILS reach the same end configuration and same end stable state, they reach this state differently: in the RLS some additional intermediate stable states are reached, that are not reached in the ILS.

Finally, a useful extension of the property language can be made by including past-time temporal operators for LTL [119] and CTL$^*$ [112]. Allowing past-time temporal operators may make the specification of some requirements easier. Past temporal operators do not increase expressivity of LTL and some past versions of CTL$^*$. They can be model checked.

## 10.2 From infinite to finite state space

The requirements-level semantics defined in Chapter 5 is not yet suitable for model checking, since the transition system of the activity diagram can have an infinite state space whereas model checking requires that the state space be finite. In this section we describe how our implementation deals with infinite state spaces. The described approaches are not new, but taken from literature.

### 10.2.1 Unbounded nodes

Combining fork and merge nodes, we can specify workflow specifications and patterns in which multiple instances of the same node are active at the same time. Figure 10.3 shows two example activity diagrams in which a node can occur more

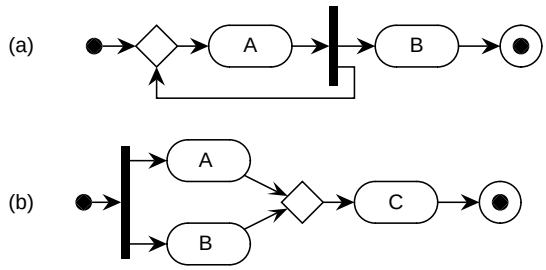



**Figure 10.3** *Examples of multiple node instances*

than once in the same configuration. In the top activity diagram, arbitrarily many instances of B can be active at the same time. In the lower activity diagram, C is executed twice; two instances of C can be active at the same time.

Such activity diagrams might have an infinite state space. For example, the activity diagram in Figure 10.3(a) has an infinite state space, because unboundedly many instances of B can be active at the same time. But the activity diagram in Figure 10.3(b) has a finite state space: there are no unbounded nodes. Formally, a node is *unbounded* if there is no bound on the maximum number of its active instances (this definition comes from Petri net theory [129]).

Model checking is decidable for bounded models [42], but for unbounded models it can easily become undecidable [68]. We therefore restrict ourselves to bounded models. In our implementation, the computation of the transition system is stopped if one of the nodes becomes unbounded. A node $n$ is unbounded iff there is a state $s$ that has $n$ in its configuration $C_s$ and $s$ has a predecessor state $s'$ such that its configuration $C_{s'}$ is strictly contained in $C_s$ and $C_{s'}$ does not contain $n$. For example, in the top activity diagram in Figure 10.3, node B is unbounded since a state with configuration [A,B] is reachable from a state with configuration [A]. (This criterion is derived from the Karp-Miller algorithm that computes the coverability graph of a possibly infinite vector addition system [108]. A vector addition system is similar to a Petri net.)

### 10.2.2 Abstracting from data

Since an activity hypergraph can have integer and string variables, the state space of the transition system can be infinite. We reduce this infinite transition system to a finite one as follows.

The key observation is that the only data that influences the execution of the activity hypergraph are the event and guard labels. The only relevant data, therefore, is the boolean valuation of the event and guard expressions. For example, suppose a guard tests whether variable $s$="*red*". Then we only need to know the truth value of the guard, if we want to know whether the associated hyperedge is

enabled.

A naive model checking strategy would therefore be to drop all data and to introduce for every guard expression a boolean representative. The guard is true iff its boolean representative is true. This strategy is naive in the sense that it ignores that guard expressions can be dependent upon each other. For example, if guard expression $[p \wedge q]$ is true then $[p]$ must also be true. And if $[s{=}\text{“}red\text{”}]$ is true then $[s{\neq}\text{“}red\text{”}]$ must be false, and vice versa. But in the naive model checking strategy, $[p \wedge q]$ and $[p]$ might be assigned conflicting truth values, for example $[p \wedge q] = true$ and $[p] = false$. Such valuations are infeasible, and therefore should not occur in the model.

We therefore consider *basic guard expressions*: those parts of the guard expressions not containing $\wedge, \vee$ and $\neg$ . This partly solves the problem sketched above (for example $[p \wedge q]$ and $[q]$ are dependent now). But not fully, since basic guard expressions too can be dependent upon each other. For example, basic guard expressions $s{=}\text{“}red\text{”}$ and $s{=}\text{“}blue\text{”}$ are not independent, since $s$ cannot be both red and blue. We have solved this problem in our implementation by enforcing that if two basic guard expressions refer to the same variable, then at most one of them can be true at the same time. To avoid that say $x < 10$ and $x > 12$ are true at the same time, the only boolean expressions referring to integers that we allow are equality tests, for example $[x = 10]$.

The approach above is based on existing approaches from modal logic theory, e.g. filtration [78]. Similar techniques are also applied in model checking under the name partition refinement [47]. Partition refinement can only be applied to a finite state space. Therefore, as far as we know, partition refinement is never applied to data abstraction, since data may induce an infinite state space.

### 10.2.3 Real time

Activity diagrams can contain simple real-time constructs of the form when and after (see Chapter 3). In our prototype, we have only implemented after constraints; when constraints can be dealt with similarly. In computing a transition system, we need to interpret after constraints in order to generate timeouts.

The problem is that our semantics uses a dense time model: between two points in time, there always exists another point in time. In a dense time model, clocks can have infinitely many values in a finite interval of time. For example, the clock used to generate the timeout in the activity diagram in Figure 10.4 has a limit of 1, but still there are infinitely many values the clock can have. Clearly, we cannot compute all these different values.

One obvious solution is to use discrete clocks. The problem then is to find the right discretisation such that at least the qualitative behaviour of the dense-time model is preserved. For example, discretising the example in Figure 10.4 with clock ticks of 2 makes configuration [WAIT-1,WAIT-4,WAIT-5] unreachable, whereas this configuration is reachable in the dense time model. In our case, we can use the

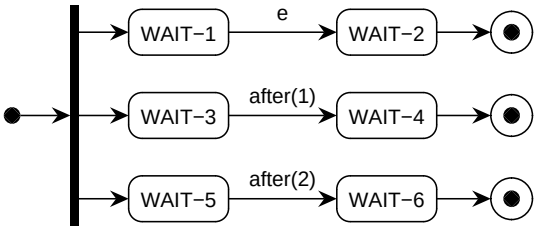



**Figure 10.4** *Example to illustrate discretisation of clocks*

result of Asarin et al. [14], based on results of Henzinger et al. [96], that dense time models in which all timing intervals are closed, can be discretised using clock ticks of 1. The dense time model that is used in our semantics falls in this class.

The discretisation preserves the untimed (reachability) properties of the original dense time model, but it may introduce some different timing behaviour [14, 96]. So it is not possible to use a real-time logic as property language. But since there is no real-time model checker supporting strong fairness constraints, we are subject to this limitation anyway.

## 10.3 Strong fairness

**The need for strong fairness.** Workflow specifications can contain loops. Consider for example the activity diagram in Figure 1.1 on page 3. There is a loop Send bill,WAIT-3,Handle payment,Notify customer,Send Bill,.... It is possible that this loop is never exited, that is, the payment may never be ok. This is not what is intended. Ideally, a workflow will eventually exit a loop, because the workflow will eventually terminate. So in Figure 1.1, ideally the payment will eventually be ok.

At first sight, it may seem as if loops are only introduced directly in the control flow, as in Figure 1.1. But even a workflow specification that has no loops in the control flow may have loops in its underlying transition system. This is due to event occurrences that can occur in a certain state but that are irrelevant and therefore ignored. For example, in Figure 10.5 event $e$ can occur while node A is active, but then it is simply ignored. Nothing in our semantics prevents $e$ from happening over and over again while A is active. The run in Figure 10.5 would therefore be a valid run. But we want to exclude such a run, because in it, activity $A$ never terminates while $e$ occurs infinitely often. (This behaviour resembles Zeno behaviour in timed systems.)

To exclude these infinite loops, we have to find a way to specify that the loops will be exited eventually. A useful way to specify this is to use strong fairness (also known as compassion) constraints. A strong fairness constraint $(p, q)$, where

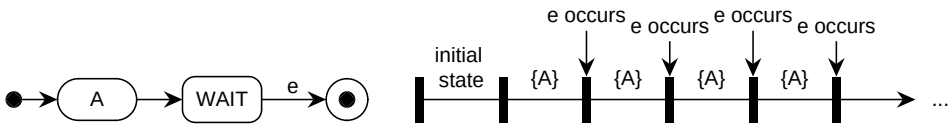



**Figure 10.5** *Example of hidden loops*

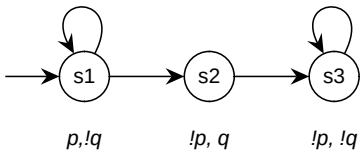



**Figure 10.6** *Kripke structure to illustrate strong fairness. '!' denotes* $\neg$

$p$ and $q$ are properties, states that if $p$ is true infinitely often in a run, then $q$ must be true infinitely often in the run as well [122]. Intuitively, a property $p$ can only be true infinitely often if there is a loop in the transition system in which $p$ is made true. So the strong fairness constraint $(p, q)$ says that if there is some loop which makes $p$ true infinitely often, then $q$ must be made true infinitely often by the loop as well. If this is not the case, the loop is not strongly fair and the loop must be exited after a finite number of iterations. For example, specifying strong fairness constraint $(p, q)$ for the Kripke structure in Figure 10.6, implies that run $s_1, s_1, ..$ is not strongly fair, since $q$ never becomes true. Strong fairness constraint $(p, q)$ only becomes true when we exit the loop around $s_1$. Using a strong fairness constraint, therefore, we can specify that some loop must be exited eventually. Note that run $s_1, s_1, .., s_1, s_2, s_3, s_3, ..$ does satisfy strong fairness constraint $(p, q)$. Even though $q$ is false infinitely often in this run, the strong fairness constraint is satisfied, since $p$ is false infinitely often as well.

**Encoding strong fairness.** We now address the question how strong fairness constraints can be encoded in workflow specifications. We have chosen to specify for every hyperedge $h$ that is triggered by an external event a strong fairness constraint that states that if $h$ is relevant infinitely often, it must be taken infinitely often. The strong fairness condition for the complete activity hypergraph is the conjunction of all individual strong fairness constraints[1]:

$$sf \stackrel{\text{df}}{=} \bigwedge_{h \in HyperEdges | \neg\, internal(h)} ( stable \wedge source(h) \sqsubseteq C, stable \wedge target(h) \sqsubseteq C )$$

[1] Strictly speaking, the formalisation does not express this since it does not state that $h$ should be taken. But for models in which no source and target of a hyperedge is contained in the source and target of another one, the formalisation is correct.

Predicate *internal* has been defined in Chapter 6 on page 83. This strong fairness constraint states that the environment must behave in a fair way: if a hyperedge is infinitely often relevant in a stable state, the environment must generate the trigger event of this hyperedge some time and must make the guard true some time. We assume that the guard is satisfiable. For the example activity diagram in Figure 10.5, strong fairness constraint $([\mathsf{A}] \sqsubseteq C, [\mathsf{WAIT}] \sqsubseteq C)$ states that activity node $\mathsf{A}$ must terminate some time. So the run in Figure 10.5 is not strongly fair, because node $\mathsf{A}$ is infinitely often contained in the configuration, but node $\mathsf{WAIT}$ is not. In this run, the environment does not behave in a fair way; consequently, we cannot give any guarantee about the correct functioning of the workflow system, for example termination of the workflow.

We do not put a strong fairness constraint for hyperedges that are triggered by the system itself, so by some internal event. The enabling of these hyperedges does not depend upon the environment, but solely upon the system itself. Since we are specifying the system, it does not make sense to put any assumptions upon it. Therefore, hyperedges triggered by an internal event do not have to be strongly fair.

An alternative way to encode strong fairness constraints is to specify a strong fairness constraint for each cycle in the generated Kripke structure. But this results in a far greater number of strong fairness constraints, since a workflow specification with external events will have cycles in almost every state (cf. Figure 10.5). In addition, we would have to take into account that for some cycles, namely those caused by internal hyperedges, no strong fairness constraints must be specified. Detection of such cycles is hard and cumbersome.

**Verifying strong fairness.** Each strong fairness constraint $(p, q)$ is equivalent to LTL constraint $\mathsf{G}\,\mathsf{F}\,p \Rightarrow \mathsf{G}\,\mathsf{F}\,q$, where (as explained before) $\mathsf{G}\,\varphi$ means that $\varphi$ is globally true in every state of the run and $\mathsf{F}\,\varphi$ means that $\varphi$ is true in some future state of the run. At first, we tried to encode the strong fairness constraints as antecedent of the LTL property that has to be verified and then use an ordinary LTL model checker like NuSMV or Spin. Since we have a lot of strong fairness constraints, however, verification of these models was undoable in practice. To illustrate this, our example has 21 hyperedges, so 21 strong fairness constraints. This already is too much for both NuSMV and Spin: we were not able to verify the simple property $sf \Rightarrow \mathsf{false}$ (true iff the model has no run), where $sf$ is the conjunction of the strong fairness constraints for every hyperedge, as explained and defined above.

We therefore decided to use an existing special algorithm for model checking LTL formulas with strong fairness constraints. The algorithm was defined by Kesten, Pnueli and Raviv [110]. With this algorithm, the strong fairness constraints are not encoded in the LTL formula that has to be verified, but the strong fairness constraints are added to the Kripke structure. The resulting Kripke structure with the strong fairness constraints is called a Fair Kripke Structure [110].

The algorithm restricts the evaluation of an LTL formula to strongly fair runs only. The algorithm has been implemented in a tool called Temporal Logic Verifier (TLV) [139]. TLV performed significantly better than NuSMV and Spin: TLV only took 20 seconds to verify *false* under the strong fairness constraints. But unfortunately, TLV does not support batch processing, so we could not integrate it into TCM. We therefore implemented the algorithm of Kesten et al. [110] in the open source model checker NuSMV, which does support batch processing. The resulting strong fairness model checker is called NuSMV$_{fair}$. It is now part of the NuSMV 2.1 model checker, which can be downloaded from the NuSMV homepage at `http://nusmv.irst.itc.it`.

## 10.4 Implementation

We first discuss some simplifying assumptions we made in our implementation. Next, the implementation itself is discussed. We only discuss the implementation made w.r.t. NuSMV. Although we have implemented support for other model checkers as well, these other model checkers, like for example Spin [99], do not have a special model checking algorithm for strong fairness, and are therefore not really useful for our purposes.

### 10.4.1 Assumptions

First, our semantics requires that for every activity $A$ the variables that $A$ reads and updates be specified. To deal with this, we have adopted the following assumptions:

1. If an activity reads a variable, we assume that it updates that variable too.
2. We assume that a variable is updated by an activity $A$ iff there exists an hyperedge $h$ such that one of $h's$ sources is labelled $A$ and the variable is tested in $h$'s guard expression. So in Figure 1.1 we assume that Check stock updates boolean variable insufficient stock.

These two assumptions make it possible to deduce for every activity automatically what variables it updates. So the user does not have to provide this information.

Next, we use the following data abstraction rule:

3. The effect of an activity is the possible change in valuation of the variables that the activity updates. Since the only relevant changes are changes in truth value of a guard, the effect of an activity is expressed in terms of the basic guards that are made true or false by that activity.

We use this assumption as follows. An activity updates those basic guard expressions that contain a variable that is updated by that activity. As explained in the previous section, we do not allow basic guard expressions that contain more than one variable.

4. The data that is updated in an activity is not updated by the environment.

The assumption is already introduced before as Constraint C1 on page 79. Not making this assumption would make some workflow specifications counterintuitive. For example, in Figure 1.1 the two choices based upon insufficient stock should have the same outcome. If above assumption is not made, the two choices might have different outcomes, which is undesirable. A nice effect of the assumption is that it reduces the state explosion.

The constraints mentioned in Chapter 6 have not been implemented, apart from afore mentioned Constraint C1.

### 10.4.2 Two implementations

We now proceed to sketch two different implementations. The first implementation is an extension of TCM, written in C++, with an execution algorithm that maps an activity hypergraph into a Kripke structure. Figure 10.7 shows a meta model in UML notation of our implementation. The model does not show how an activity diagram is translated into an activity hypergraph; see Section 3.3 and a technical report [60] for that.

The main part of our implementation consists of an iterative algorithm that processes states (valuations) of the Kripke structure. A state is processed in one of the following two ways.

- If a processed state is stable, timers can tick and/or some events can occur. If events occur, the next state becomes unstable. The algorithm computes all possible states that can occur next. If the next state is unstable, then some named events occur, or some basic guards change value, or some activity nodes terminate, or some timeouts occur.
- If a processed state is unstable, the algorithm computes all possible steps and all the resulting next states that are reached when those steps are taken. A resulting next state is stable if there are no events in the queue and no enabled hyperedges in this state; it is unstable otherwise.

The new states that are generated while a state is processed, are processed later. The algorithm stops if all states have been processed. The resulting transition system can be straightforwardly encoded as input for a model checker by enumerating every state of the structure. Section 10.6 analyses the structure and size of the state space and analyses different ways of reducing it.

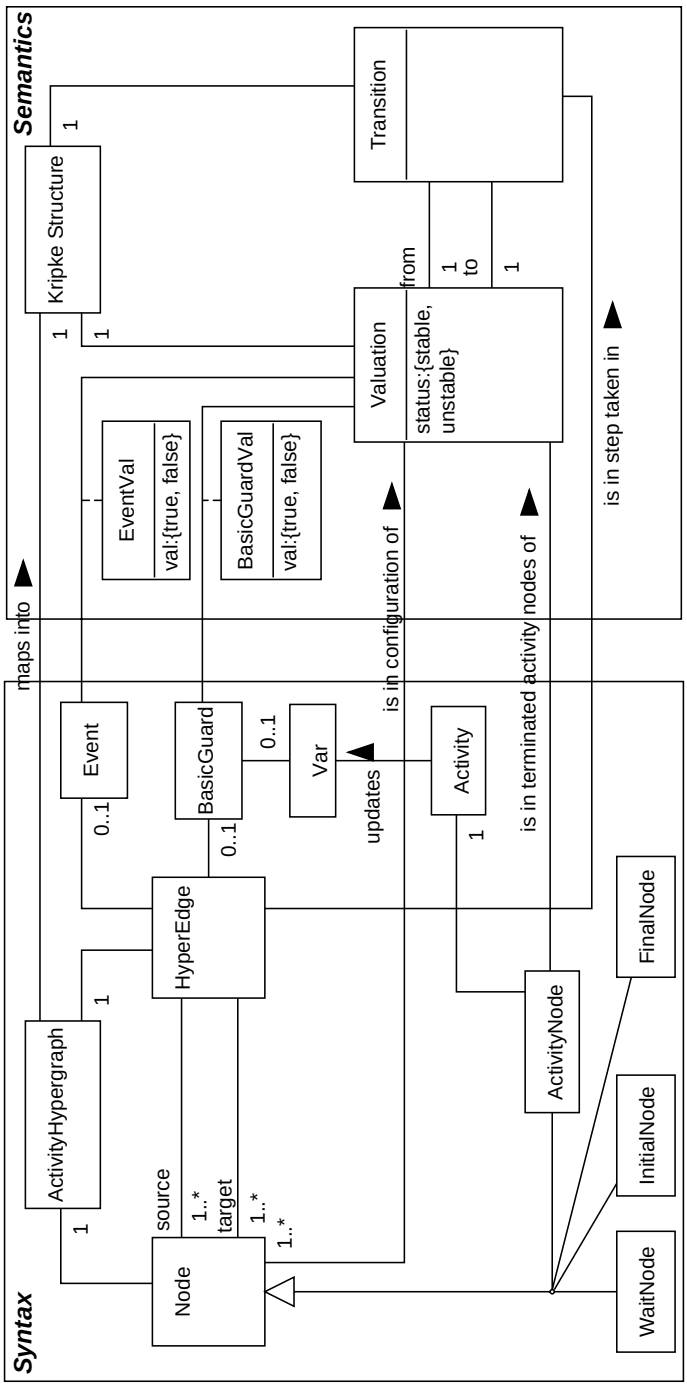



**Figure 10.7** *Meta model of implementation*

We have also experimented with a second implementation that is based on existing approaches [34, 126] to verify statecharts with symbolic model checkers like for example NuSMV. In the second implementation, the syntax of an activity hypergraph is encoded directly as input for a symbolic model checker; the semantics that the symbolic model checker attaches to the input coincides with the requirements-level semantics. The second implementation cannot deal with every possible activity diagram. It can only deal with safe activity diagrams, i.e., activity diagrams in which a node cannot be active more than once at the same time. Moreover, if some hyperedges share sources and targets, the implementation does not work any more, because then some constraints in the input will conflict with each other. Nevertheless, if the second implementation can be applied, it is more efficient than the first implementation.

Both implementations are available from the TCM homepage at `http://www.cs.utwente.nl/~tcm`.

## 10.5 Example verifications

We discuss some example verifications of requirements for the workflow specification of Figure 1.1. We distinguish general and ad-hoc requirements. A general requirement must hold for every possible activity diagram, while an ad-hoc requirement is specified for a specific activity diagram. Performance statistics are given at the end of this section.

**Note.** Throughout the thesis, each requirement is only defined for strongly fair runs, but this is not shown explicitly in the definitions in order to avoid cluttering. TCM automatically generates the appropriate strong fairness constraint for each workflow specification.

Moreover, each atomic proposition is implicitly conjoined with predicate *stable*, so for example $\mathsf{in}(\mathsf{WAIT})$ should be read as $\mathit{stable} \wedge \mathsf{in}(\mathsf{WAIT})$.

### 10.5.1 Ad-hoc requirements

Since every workflow specification might have its own ad-hoc requirements, we just give an example for the workflow specification shown in Figure 1.1.

Ad-hoc requirement R1 states that for each possible strongly fair run, either both Make production plan and Produce occur sometime in the future or both of them do not occur. We begin with formalising this property as:

$$\text{(R1):} \quad \mathsf{F}\,\mathsf{in}(\text{Make production plan}) \Leftrightarrow \mathsf{F}\,\mathsf{in}(\text{Produce})$$

where $\mathsf{in}(x)$ is true in a valuation $\sigma$ iff node $x$ is contained in the configuration of $\sigma$. (TCM translates in into an equivalent predicate on nodes.) This property fails

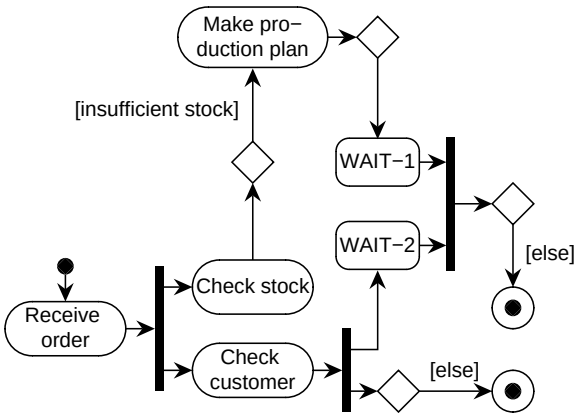



**Figure 10.8** *Path illustrating counterexample for* R1

to hold: The path that TCM highlights is shown in Figure 10.8. We see that if the customer check fails, the workflow stops while Make production plan may already have been executed.

There are two ways to repair this error: either change the requirement or the activity diagram. We decide to adapt the requirement. Apparently, only if the customer check does not fail, the requirement holds:

$$\text{(R1')}: \quad (\,\mathsf{F}\,\text{customer ok}) \Rightarrow (\mathsf{F}\,\text{in(Make production plan)} \Leftrightarrow \mathsf{F}\,\text{in(Produce)}\,)$$

NuSMV$_{fair}$ reports that this property is true. This property is true, because of our Assumption 4 in Section 10.4 which implies for this workflow specification that only Check stock can change variable insufficient stock. If we had allowed the environment to change insufficient stock, the property would not have been true. But that would have been counterintuitive.

Finally, we verify that in each strongly fair run, a bill is sent if and only if either something is produced or taken from stock:

$$\text{(R2)}: \quad \mathsf{F}\,(\text{in(Produce)} \vee \text{in(Fill order)}) \Leftrightarrow \mathsf{F}\,\text{in(Send bill)}$$

NuSMV$_{fair}$ reports that this property is true.

### 10.5.2 General requirements

We list four general requirements. Other general requirements are possible. The first general requirement is that for every strongly fair run, from the initial state a final state should be reachable. We formulate this requirement as the following LTL formula:

(R3): $\mathsf{F}\,\mathsf{G}\,\mathit{final}$

where *final* is true in a state iff the configuration only contains final nodes (bull's eyes). TCM translates *final* into an equivalent predicate on nodes:

$$\mathit{final} \quad \stackrel{\text{df}}{\Leftrightarrow} \quad \forall\, n \in \mathit{Nodes} \bullet n \sqsubseteq C \;\Rightarrow\; n \in \mathit{FN}$$

NuSMV$_{fair}$ reports that the property is true.

Note that this formula is not in CTL$^*_r$. Fortunately, for activity diagrams $\mathsf{F}\,\mathsf{G}\,\mathit{final}$ is equivalent to $\mathsf{G}\,\mathsf{F}\,\mathit{final}$, because final nodes have no outgoing hyperedges. And $\mathsf{G}\,\mathsf{F}\,\mathit{final}$ is a CTL$^*_r$ formula.

Another useful general requirement is that there are no dead nodes, i.e., for every node there is a strongly fair run in which that node becomes active.

(R4): for all $n \in \mathit{Nodes}, \mathsf{EF}\,\mathsf{in}(n))$

Note that this formula is neither an LTL nor a CTL formula, but a CTL$^*$ formula. It is not an LTL formula due to the existential path quantifier $\mathsf{E}$. It is not a CTL formula because it is defined for strongly fair runs only. Below we will address how this general requirement can be verified.

Yet another general requirement is that there are no dead hyperedges, i.e., every hyperedge should be taken in some strongly fair run. Strictly speaking, checking this requires a logic that can refer to hyperedges. But an equivalent check is that for every hyperedge $h$ there exists a strongly fair run in which the sources of $h$ are active in one state and the targets of $h$ in the next state (under the assumption that for every two hyperedges, the sources and targets of one are not contained in the sources and targets of the other).

(R5): for every hyperedge $h, \mathsf{E}\,(\mathit{sf} \wedge \mathsf{F}\,(\mathsf{in}(\mathit{source}(h)) \wedge \mathsf{X}\,\mathsf{in}(\mathit{target}(h))))$

where $\mathsf{in}(N)$ abbreviates $\bigwedge_{n \in N} \mathsf{in}(n)$. Note that this formula is neither an LTL nor a CTL formula, but a CTL$^*$ formula. Moreover, it is not a CTL$^*_r$ formula, since the next time $\mathsf{X}$ operator is used. For the ILS, a similar requirement could be formulated by referring to component $S$ (the variable in which the current step is stored). Below we will address how this general requirement can be verified.

The last general requirements states that the activity diagram does not diverge, that is, there is always a future in which the activity diagram is stable:

(R6): $\mathsf{G}\,\mathsf{F}\,\mathit{stable}$

NuSMV$_{fair}$ reports that this requirement is true.

**Checking possibility requirements.** Unfortunately, requirements R4 and R5 are CTL$^*$ constraints that are neither CTL formulas, due to the implicit strong fairness constraint, nor LTL constraints, due to the existential path quantifier $\mathsf{E}$.

Currently, NuSMV$_{fair}$ only supports LTL constraints with strong fairness. Luckily, as we will proceed to show, checking these two constraints does not require strong fairness: we can drop the strong fairness constraint and check the resulting CTL formula using a standard CTL model checker.

The following theorem asserts that for activity diagrams a non-strongly fair path can always be extended into a strongly fair path.

**Theorem 10.3** *Assume a path $\pi$ does not satisfy the strong fairness constraint $(stable \wedge source(h) \sqsubseteq C, stable \wedge target(h) \sqsubseteq C)$ for some non-internal hyperedge $h$. Then $\pi$ can always be extended into a path $\pi'$ which does satisfy that strong fairness constraint.*

*Proof.* According to Kesten et al. [110], a path $\pi$ always has the following structure:

$$\pi = \overbrace{\sigma_0, .., \sigma_k}^{prefix}, \overbrace{\sigma_{k+1} .. \sigma_k}^{period}, \overbrace{\sigma_{k+1}, .., \sigma_k}^{period}, \overbrace{\sigma_{k+1}, .., \sigma_k}^{period}, \overbrace{\sigma_{k+1}, .., \sigma_k}^{period}, \ldots$$

If path $\pi$ does not satisfy $(stable \wedge source(h) \sqsubseteq C, stable \wedge target(h) \sqsubseteq C)$, then in the period part there must be some state $\sigma_m$, $m > k$, such that $\sigma_m \models stable \wedge source(h) \sqsubseteq C$, and no state in the period part satisfies $stable \wedge target(h) \sqsubseteq C$.

Since $h$ is relevant and the event can be generated and the guard can become true, it is clear that there does exist a state $\sigma_n$ such that $\sigma_n \models stable \wedge target(h) \sqsubseteq C$. Moreover, $\sigma_n$ is reachable from $\sigma_m$.

We construct a new path $\pi'$ by extending $\pi$: the prefix of $\pi$ consists of the prefix and period of $\pi$ followed by the part of the period leading to $\sigma_m$. Next we construct a path $\pi''$ from $\sigma_m$ to $\sigma_n$. If $\sigma_m$ is reachable from $\sigma_n$ by path $\pi'''$, the period part of $\pi'$ is $\pi''$ followed by $\pi'''$. If $\sigma_m$ is not reachable from $\sigma_n$, path $\pi''$ also belongs to the prefix of $\pi'$ and the period part of $\pi'$ can be anything. (Because the transition relation is total, there always exists a period.) □

Using this theorem, the following can be easily proven. As before, *sf* denotes the strong fairness condition.

**Corollary 10.4** *Given a valuation $\sigma$ under the RLS. Then*

*(i)* $\sigma \models \mathsf{E}\,(sf \wedge \mathsf{F}\,p) \Leftrightarrow \sigma \models \mathsf{EF}\,p$

*(ii)* $\sigma \models \mathsf{A}\,(sf \Rightarrow \mathsf{G}\,(\,p \Rightarrow \mathsf{E}\,(sf \wedge \mathsf{F}\,q)\,)\,) \Leftrightarrow \sigma \models \mathsf{AG}\,(p \Rightarrow \mathsf{EF}\,q)$

*where $p$, $q$ are simple state formulas, not containing any temporal operator.*

| Requirement | Result | Time (sec) | BDD nodes (nr) | Memory (kB) |
|---|---|---|---|---|
| R1 | False | 20.21 | 138280 | 10107 |
| R1' | True | 19.99 | 146188 | 10227 |
| R2 | True | 20.01 | 131619 | 9991 |
| R3 | True | 20.82 | 132525 | 10015 |
| R6 | True | 20.34 | 130877 | 9979 |

**Table 10.1** *Resources used by* NuSMV$_{fair}$ *for running example*

**Performance statistics.** General requirements R3 and R6 are simple LTL formulas that can be checked using NuSMV$_{fair}$. General requirements R4 and R5 are equivalent to CTL formulas by Corollary 10.4. But they can be checked immediately in TCM during generation of the transition system, and therefore do not need to be checked by NuSMV$_{fair}$.

The resources used by NuSMV$_{fair}$ during this analysis are shown in Table 10.1. The analysis was performed on a PC with a Pentium III 450 MHz processor with 128Mb of RAM under Red Hat Linux 6.0. We verified the same properties by hand with TLV; the outcomes were the same. NuSMV$_{fair}$ is slightly faster, probably since a more efficient BDD library is used. Most time during analysis was spent by the execution algorithm in TCM that computes the input transition system for the model checker; this time is not shown. TCM can be optimised by applying BTrees or hash lists.

## 10.6 State explosion

We first analyse the state explosion problem for activity diagrams. Based on this analysis, we define four reduction rules on activity diagrams that alleviate this problem.

### 10.6.1 Analysing state explosion

The greatest problem of verification with model checking is the state explosion. Realistic models tend to get very large; in fact so large, that verification is no longer possible. For example, even the small example in Figure 1.1 has already over 350 states (see Table 10.2 below), although the number of nodes in the workflow specification is only 19. There are several causes for state explosion.

- Parallel threads. These are started by hyperedges that have more than one target, and stopped by hyperedges having more than one source. The product of two parallel threads that have $x$ and $y$ states respectively, will

have $x \times y$ states. In the example of Figure 1.1, due to parallelism there are 47 different configurations, although there are only 19 nodes in the underlying hypergraph.

- Events. There are named external and activity termination events in the running example. Named external events can occur in any stable state, whereas activity termination events can only occur if the corresponding activity node is in the current configuration. Hence, activity termination events do not cause state explosion, since their occurrence is limited, but named external events do cause state explosion.

  Combining this, if there are $k$ external events and in a given stable state the current configuration contains $l$ activity nodes, then there are $2^{k+l} - 1$ (the non-occurrence of events is excluded) possible combinations of events. This means that from the stable state, $2^{k+l} - 1$ unstable states can be reached. This explains why the example in Figure 1.1 has over 350 different states, even though it has only 47 configurations and 1 named external event.

  Temporal events (timeouts) can only occur if some timers have reached their limits, see below. Their occurrence is therefore limited. Nevertheless, we show in the next item that due to the discretisation some state explosion may occur.

- Real time. Due to clock ticks, some extra states are introduced. For example, if a timeout is $m$, then $m$ ticks are needed before the timeout is generated. This means that $m$ extra states are introduced in the corresponding branch. For example, the timer for after(2 weeks) in Figure 1.1 introduces two extra states in the branch of the Financial department. In combination with parallel branches and events, mentioned above, this can cause a state explosion.

- Data (basic guard conditions). Due to Assumption 4 in Section 10.4, a local variable is not updated by the environment. So a local variable only changes value after termination of some activity that updates the variable. Moreover, because every data is tested in a choice after an activity terminates, there are specific combinations of data and nodes. For example, in Figure 1.1 if node Notify customer is active then payment ok has to be false. Hence, basic guard conditions that are updated by activities do not blow up the state space with respect to stable states.

  But basic guard conditions *do* blow up the number of unstable states. If after an activity node terminates a choice is made out of $n$ alternatives, then there are $n$ different terminations. So $n$ different unstable states are possible. In Figure 1.1, for example, Notify customer can terminate in 2 possible ways. In combination with named external event occurrences and activity termination events in parallel branches (see the previous item), this can result in a blowup in the number of unstable states that are possible.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| receive payment/ after(2 weeks) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| customer ok | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| insufficient stock | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| payment ok | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| nr of states | 369 | 347 | 290 | 272 | 419 | 393 | 316 | 296 |
| receive payment/ after(2 weeks) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| customer ok | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| insufficient stock | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| payment ok | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| nr of states | 167 | 156 | 128 | 119 | 188 | 175 | 139 | 129 |

**Table 10.2** *Effects of presence of events and data upon size of the model. Entry '1' denotes the presence of an item, '0' denotes its absence*

For example, Table 10.2 shows that including payment ok leads to around 20 additional states.

To illustrate the effect of events and data upon the size of a model, we have computed several variants of the activity diagram in Figure 1.1. Table 10.2 shows the results of removing event and guard conditions upon the number of states of the activity diagram. Event labels receive payment and after(2 weeks) must be removed at the same time; otherwise, if only one of these events is removed, the unlabelled hyperedge (a completion hyperedge) will have priority over the labelled hyperedge and some parts of the activity diagram will become unreachable. In the next subsection we define several reduction rules.

There are several things worth noticing. First, in this case, including one event doubles the state space. As a test, we included another dummy event in the activity diagram of Figure 1.1 on a new edge between WAIT and Handle payment; the resulting model had 715 states. We also computed a variant of Figure 1.1 in which an event only occurs when it is relevant, i.e., when it triggers some relevant hyperedge. The resulting model has 229 states.

Second, abstracting from data that is used in one choice only, for example payment ok, does not have a big impact on the state space: the model is reduced by around 20 states. This effect we already explained above in the third item.

Third, perhaps a bit surprisingly, the table shows that removing guards may increase the state space, rather than decreasing it: if guard customer ok is removed, the state space becomes larger. The reason for this is that some choices in parallel branches can be dependent upon each other (in this case the two choices based upon customer ok). This dependency is lost if these choices are made nondeterministic (in this case if customer ok is abstracted from). Then, some configurations

that do not exist when the guard is included, *do* exist if the guard is not included (in this case, when customer ok is not modelled, the branch starting with node Send bill can be active whereas the other parallel branch immediately stops and does not do Produce or Fill order). In the example, removing guard customer ok introduces 9 extra configurations. By the way, removing guard customer ok does have the effect as described under the last item (Data), but this effect apparently does not compensate for the extra configurations and states that are introduced.

### 10.6.2 Fighting state explosion

We now define four rules to reduce the state space of an activity hypergraph, given a $\text{CTL}^*_r$ formula $\varphi$. The first two rules are defined on the semantics of activity hypergraphs. The last two rules are defined on the syntax of activity hypergraphs. For each of the rules, we discuss when they can be applied. Every reduction rule $r$ is sound and complete for $\varphi$, that is, $\varphi$ holds for the CTS of the original activity hypergraph $AH$ iff $\varphi$ holds for the CTS of $AH$ if $r$ is applied. Unless stated otherwise, the reduction rules are also valid for activity hypergraphs not satisfying the constraints mentioned in Theorem 6.7 on page 101.

**Rule 1: No irrelevant event occurrences.** A named external event is irrelevant in a certain state iff it does not trigger any relevant hyperedge. By disallowing irrelevant named external event occurrences, in other words only allowing an event $e$ to occur if it triggers a relevant hyperedge, we can reduce the state space. The rule rules out hidden loops in an activity diagram: for example the run of the activity diagram in Figure 10.5 would not be computed if this reduction rule was used.

This reduction rule is allowed for every activity hypergraph and every $\text{CTL}^*_r$ formula under the condition of strong fairness.

**Rule 2: Interleaved named external event occurrences.** Only allow interleaved named external event occurrences; that is, no two named external events can occur at the same time. Note that this rule does not apply to temporal events and condition change events.

This reduction rule is allowed for every activity hypergraph satisfying constraints C2, C4(b), C10, C15, C17 (see Table 6.3 on page 91) and for every $\text{CTL}^*_r$ formula. For these constraints, motivating examples have been presented in Chapter 6.

**Rule 3: Remove local variables.** Remove local variable $v$ from the activity hypergraph and remove every basic guard condition that refers to $v$.

This reduction rule is allowed if:

- $v$ is not updated by two concurrent activities;

- the requirement $\varphi$ does not refer to $v$;
- the only hyperedges referring to $v$ are the ones leaving the activity node A in which $v$ is updated, $v \in \mathit{Upd}(\mathit{act}(\mathsf{A}))$;
- in a decision, the disjunction of basic guard expressions referring to $v$ is true. This can be easily ensured by including an else branch in every decision.

The first constraint is needed because otherwise two interfering activity nodes can become active at the same time in the reduced activity hypergraph, which is impossible in the original activity hypergraph due to our step semantics. The second constraint ensures that the requirement $\varphi$ can still be evaluated on the reduced activity hypergraph. The third constraint ensures that the reduced activity hypergraph does not have more configurations than the original one. In our running example (Figure 1.1), variables insufficient stock and customer ok cannot be removed because they are contained in the guards of the hyperedges leaving nodes WAIT-1 and WAIT-2. We already saw above that if they are removed, some extra configurations, and thus states, are introduced. Then the truth value of the requirement for the reduced activity hypergraph may differ from the truth value for the original activity hypergraph. If for example customer ok is removed, requirement F G *final* is no longer true, even though it is true in the original activity hypergraph. The fourth constraint ensures that if the original activity hypergraph contains a deadlock, the reduced one contains a deadlock as well.

Applying this reduction rule to our running example (Figure 1.1), if the requirement to be verified is F G *final*, then variable payment ok can be removed and the corresponding guard conditions can be removed as well.

**Rule 4: Remove nodes.** If there is an activity or wait node $n$ with only one outgoing external hyperedge $h$ such that $\mathit{source}(h) = \{n\}$, so $h$ does not conflict with any other hyperedge, then both $n$ and $h$ can be removed from the activity diagram, by replacing every occurrence of $n$ in the target of some hyperedge with the targets of $h$. If $h$ was the last hyperedge referring to some trigger event and/or local variable, these can be removed from the set of events and local variables respectively.

This reduction rule is allowed if:

- the requirement $\varphi$ does neither refer to $n$, nor to the label of $h$, nor to the target nodes of $h$;
- neither $n$ nor the target nodes of $h$ are referred to by some in predicate in the activity diagram;
- the trigger event of $h$ is a named external event or a termination event;
- the trigger event $e$ of $h$ can only occur in this state, either because (i) $n$ is an activity node and $e$ denotes termination of $n$, or because (ii) $n$ is a wait

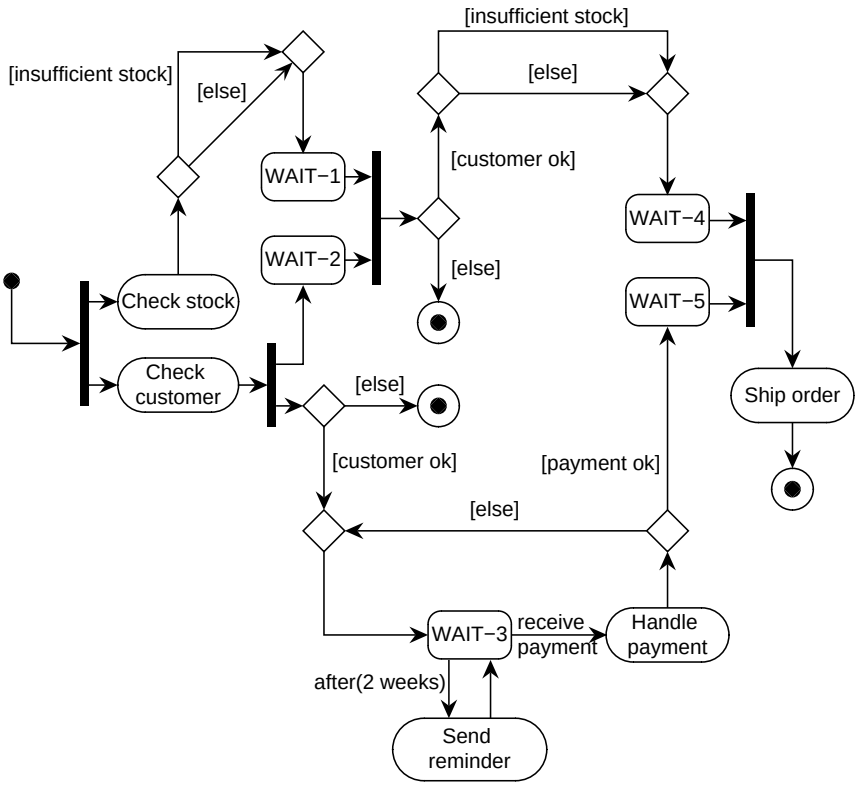



**Figure 10.9** *Reduced activity diagram of Figure 1.1. The used requirement is* $\mathsf{F}\,\mathsf{G}\,$*final*

node, $e$ is a named external event, Constraint C15 is satisfied, and reduction rule 1 (no irrelevant events) is used.

If we want to apply Theorem 6.7, we must furthermore ensure that the reduced activity diagram satisfies Constraint C16, i.e. there is no hyperedge with some wait node both as source and target.

The first constraint ensures that the requirement can still be evaluated on the reduced activity hypergraph. The second constraint prevents that an $\mathsf{in}$ predicate has an undefined value. The third and fourth constraint ensure that the trigger event of $h$ only triggers $h$ and moreover can only occur in $n$. Thus, removal of both $n$ and $h$ will not affect other parts of the activity hypergraph.

Figure 10.9 shows a reduced activity diagram of Figure 1.1, where the requirement to be verified is $\mathsf{F}\,\mathsf{G}\,$*final*.

The following theorem states that the reduction rules are sound and complete

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| rule 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| rule 2 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| rule 3 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| rule 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| nr of states | 369 | 229 | 209 | 172 | 347 | 218 | 203 | 166 |
| | | | | | | | | |
| rule 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| rule 2 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| rule 3 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| rule 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| nr of states | 149 | 99 | 91 | 78 | 139 | 94 | 88 | 75 |

**Table 10.3** *Effects of applying reduction rules upon size of the model of the running example. Entry '1' denotes that the rule is applied, '0' is otherwise. If rule 3 is applied, variable* payment ok *is removed. Rule 4 is applied with requirement* F G *final; the activity diagram of Figure 10.9 is obtained*

for a $\mathrm{CTL}_r^*$ formula $\varphi$.

**Theorem 10.5** *Let $\varphi$ be the $CTL_r^*$ formula to be verified and let $AH$ be the activity hypergraph. If $\varphi$ satisfies the constraints of reduction rule $r$, then $\varphi$ holds for $CTS(AH)$ if and only if $\varphi$ holds if $r$ is applied, written $CTS(AH)_r$:*

$$CTS(AH) \models \varphi \;\Leftrightarrow\; CTS(AH)_r \models \varphi$$

*Proof.* For every reduction rule, the claim can be proven by induction on the structure of $\mathrm{CTL}_r^*$ formulas. Rule 1, 2 and 3 are straightforward, because the reduced CTS has the same reachable configurations as the original CTS. The reduced CTS has less transitions than the original one, but this cannot be sensed by $\mathrm{CTL}_r^*$ formulas. Rule 4 follows from the fact that any state with $n$ in its configuration has the same successors, apart from $target(h)$, as any state with $target(h)$ in its configuration. By the constraints of rule 4, $\varphi$ does not refer to nodes in $target(h)$. So the reachable states that are relevant for the truth value of $\varphi$ are not removed from $CTS(AH)$ when rule 4 is applied. □

All four reduction rules have been implemented in TCM. The reduction rules are applied recursively to an activity hypergraph until no rule can be applied anymore.

Table 10.3 shows the effect of applying the reduction rules on the size of the state space of our running example. Especially the fourth rule has a spectacular effect on the size of the state space. The third rule has the least effect. In the next chapter, in which we discuss some real-life case studies, we will see how these reduction rules can be used to deal with large state spaces.

| Requirement | Result | Reachable states (nr) | Time (sec) | BDD nodes (nr) | Memory (kB) |
|---|---|---|---|---|---|
| R1 | False | 107 | 1.76 | 40454 | 3715 |
| R1' | True | 107 | 1.75 | 41055 | 3767 |
| R2 | True | 133 | 2.81 | 47729 | 4367 |
| R3 | True | 75 | 1.03 | 26221 | 3067 |
| R6 | True | 71 | 0.84 | 22252 | 2875 |

**Table 10.4** *Resources used by* NuSMV$_{fair}$ *for running example using all four reduction rules. The original model has 369 states*

Table 10.4 shows the performance results for model checking our running example with NuSMV$_{fair}$ when all four reduction rules are applied. Comparing this to the original model, in which no reduction is applied, shown in Table 10.1, it is clear that even for this small example applying the reduction rules improves the performance of the verification considerably.

**Ad-hoc reduction.** Above we saw that abstracting from local variables that are tested twice in two different choices, like customer ok in our running example, may introduce new reachable states, including some new configurations. It may seem, therefore, that abstracting from such guard conditions makes the resulting Kripke structure useless for analysis. For example, F G *final* is no longer true if guard customer ok is removed from our example. But if the dependency between the different choices can be stated as an antecedent to the property to be verified, such abstract models can still be useful. Then, however, the dependencies must be defined by hand. For example, in the following property, the antecedent states that only those runs should be checked in which a bill is sent if and only if something is produced or an order is filled.

$$(\mathsf{F}\,\mathsf{in}(\mathsf{Send\ bill}) \Leftrightarrow \mathsf{F}\,(\mathsf{in}(\mathsf{Produce}) \vee \mathsf{in}(\mathsf{Fill\ order}))) \Rightarrow \mathsf{F}\,\mathsf{G}\,\mathit{final}$$

Even if guard customer ok is removed from the activity diagram, this property will hold. This shows that reduction is also possible on an ad hoc basis, depending upon the model, the requirement to be verified, and especially the insight of the modeller. Of course, verification of such properties only works if the property does not refer to an item that is abstracted from.

## 10.7 Related work

**Restricted temporal logic.** In a seminal paper, Lamport [113] argues not to use the next time X operator in temporal logic specifications, because this operator makes it possible to distinguish between a high-level specification and a lower

level implementation. Lamport does not, however, restrict usage of the until $\mathsf{U}$ operator, because in his formalism variables referred to by a property must change instantaneously. By contrast, in our implementation-level semantics, variables that a property refers to do not have to change instantaneously. Consequently, it is possible to observe an inconsistent state. For example, in the ILS it is possible to observe a configuration in which some nodes have been left but still some nodes have to be entered. That is why the only states we allow to observe are stable ones. That is also why we need to restrict usage of the until $\mathsf{U}$ operator, whereas Lamport does not need to do so.

**General requirements.** The conjunction of the first three general requirements resembles the soundness criterion that Van der Aalst [3] uses for Petri nets that model workflows. There is however a subtle difference. The proper termination property in the soundness criterion amounts to the CTL formula $\mathsf{AG}\,\mathsf{EF}\,\mathit{final}$, which states that the workflow *can* terminate properly. This formula is weaker than the property we use, $\mathsf{F}\,\mathsf{G}\,\mathit{final}$, which states that the workflow *will* terminate properly. The difference between the two formulations pops up in the case of divergence: a diverging workflow specification would pass the soundness criterion but not our proper termination requirement.

Hofstede and Orlowska [98] also discuss some general requirements for workflow models that are formalised in process algebra. They focus on the computational complexity of verifying these requirements. They do not consider divergence. Sadiq and Orlowska [143] also focus on verifying general requirements like absence of deadlock by applying graph reductions. They do not consider divergence.

**Verification tools.** There are several workflow verification tools. Woflan [155] is a tool for verification of textual workflow specifications without data and real-time. Feedback is also textual. The workflow specifications are based on low-level Petri nets. In Woflan the properties that are verified, like soundness, are fixed and cannot be changed by the user. The issue of strong fairness is not addressed.

FlowMake [143] is a tool for verification of workflows that are notated in a subset of the WFMC workflow notation. Data and real-time are not modelled. FlowMake verifies some fixed properties of a workflow by applying graph reduction techniques; if the reduction does not lead to an empty graph, apparently the graph contains an error and it should be possible for the user to find this error using the reduced graph. The issue of strong fairness is not addressed.

The tool developed in the Mentor project [130] uses a CTL model checker for statecharts [101]. The tool is not integrated with the model checker. The authors do not use strong fairness. They do not provide any details on how the feedback is presented to the user.

The Testbed Studio tool [104] supports model checking of business process models with Spin [99]. The process modelling language neither has external events nor temporal events. Models can have loops, but the analysis results on such

models may be counterintuitive, since it cannot be specified that loops are exited. The authors do not use strong fairness.

Karamanolis et al. [107] use the existing LTSA toolkit for model checking workflow specifications. Workflow specifications are translated manually into input for LTSA; the output of verification is shown graphically in the LTSA input, not in the original workflow specification. LTSA is based on process algebra; data and real-time cannot be explicitly modelled. Strong fairness constraints can also be specified in LTSA, but Karamanolis et al. [107] do not focus on loops in workflow schemas.

Our work is also closely related to the work done on model checking STATEMATE and UML statecharts. Chan et al. [34] and Mikk [126] have defined model checking for STATEMATE statecharts or variants thereof, using SMV [125] and Spin [99]. Latella et al. [116] present a translation for a subset of UML statecharts to Spin [99]. None of the implementations discussed in these papers provide a graphical representation of the feedback of the model checker. All these papers encode the syntax of the statechart explicitly in the input language and let the model checking tool derive the step semantics implicitly. We, on the other hand, have programmed our execution algorithm [65] in TCM, so TCM generates the semantic structure directly. These syntactic encodings only work for simple models with a restricted syntax. Amongst others, every node can be active at most once at the same time: it must have a bound of one (i.e., the activity diagram must be safe). This is true for a statechart but not for an activity diagram. Also, syntactic encodings are error prone (see for example a discussion by Mikk on errors he found in such translations [126]).

We have also implemented a syntactic encoding for activity diagrams in TCM in order to compare it with our own encoding. We found that if the syntactic encoding can be applied, it is more efficient than the enumerative encoding we use. But in order to decide whether the syntactic encoding can be applied, still the semantics of the activity diagram needs to be computed using our first implementation in order to check that the activity diagram is safe.

Lilius and Paltor [120] present vUML, a tool for model checking a communicating set of objects whose behaviour is modelled by UML statecharts. They use Spin [99] as their underlying model checker. No details are given on how the statechart is encoded. The feedback of the tool is graphically represented by a UML sequence diagram. They neither address strong fairness nor real-time.

It is difficult to compare our work with this work on statecharts, since our semantics differs somewhat from the statechart semantics, both UML and STATEMATE, in particular since we have atomic activity states whose effect is declaratively specified (these are not present in statecharts). Next, we have configurations and steps that are bags rather than sets, since nodes in our models can have a bound of more than one. None of the references above use strong fairness constraints, since these are apparently not required in the domain they model (usually embedded real-time systems). Neither do they analyse the state space nor discuss possible

reductions.

**Reduction rules.** Our reduction rules are similar to slicing rules in program analysis (see Tip [151] for a survey). In program analysis, slicing is used to increase program understanding and to make program debugging and program testing more easy. We use reduction rules to decrease the state space of the models we check. The reduction rules may however also be beneficial for a workflow modeller to increase understanding of the workflow models.

Recently, slicing has been used in combination with model checking by Chan et al. [35], Clarke et al. [40], and Hatcliff et al. [93]. Chan et al. [35] slice a RSML statechart with respect to a given property $\varphi$ by removing parallel nodes in the statechart that are not (in)directly referred to by $\varphi$. In particular, if a certain node is relevant, all its predecessors are relevant as well; these are not removed. Difference with our approach is that we sometimes cut away predecessor nodes (rule 4). We do not remove parallel nodes. Clarke et al. [40] apply slicing to VHDL programs. Hatcliff et al. [93] slice Java-like programs for model checking. Difference of our approach with both these references is that their programming languages are at a lower level of abstraction than activity diagrams, and that the sliced programs are not concurrent, whereas activity diagrams are. Heimdahl and Whalen [95] also apply slicing to RSML statecharts, but their purpose is to facilitate manual review of requirements on RSML statecharts, not formal automatic verification of these requirements.

## 10.8 Conclusion and future work

We presented a prototype implementation that supports workflow modellers in verifying workflows specified in UML activity diagrams. The tool translates an activity diagram into an input for a model checker according to the requirements-level semantics defined in Chapter 5. Our tool supports the specification of event-driven behaviour, data, real-time and loops in workflow specifications. Also, the properties that are checked can be specified by the user himself and are not fixed. The appropriate strong fairness constraints are generated automatically by the tool. The used model checker is under the hood of our tool; the user merely has to know an LTL based input language. If the model checker returns a counterexample, the tool translates this counterexample back into the activity diagram by highlighting a corresponding path. The tool can automatically reduce the input activity diagram by applying the reduction rules defined in Section 10.6. These features makes our tool different from other workflow verification tools and also from existing verification tools for the related UML statecharts. As far as we know, our tool is the first verification tool for UML activity diagrams.

The most interesting result is that workflow specifications require strong fairness constraints. Although there are some model checkers that support verification

of models that have strong fairness constraints, only model checkers that use a special model checking algorithm for strong fairness perform well enough to be useful. Since no existing model checker was suitable for our purposes, we have extended NuSMV with an existing model checking algorithm for strong fairness.

Future work is to implement the stable implementation-level semantics in the tool. Moreover, a more abstract requirement specification language is required, since temporal logic is difficult to understand for users that do not have a mathematical background. It is also interesting to see whether verification of statecharts would benefit from our approach, i.e., the restricted logic and the reduction rules.

# Chapter 11

# Case studies

In the previous chapter we have explained how the requirements-level semantics of UML activity diagrams can be used to verify workflow models. To see whether the verification also works for non-toy examples, we do two case studies in this chapter. Both case studies are based on existing workflow models that are being used in organisations. Our main concern will be scalability: what is the size of the models that still can be verified in reasonable time? And is this size reasonable, or is only verification of toy examples possible?

In Section 11.1 we verify a workflow for the seizure of goods at the Dutch Public Prosecution Service. In Section 11.2 we verify the order procedure within an IT department of a large company. Section 11.3 discusses the lessons that we learned. We end with conclusions.

## 11.1 Seizure of goods under criminal law

This section describes a workflow at the Dutch Public Prosecution Service (PPS). The Public Prosecution Service's main tasks, as laid down by law, are investigating criminal offences, prosecuting offenders, and ensuring that sentences are carried out properly.

Part of these responsibilities is the seizure of goods. The PPS can seize goods for example because it is necessary for discovering the truth, or because the goods have been unlawfully obtained.

**Workflow description.** The now following workflow description is based on a Petri net based workflow description [94] and my own interpretation of the Dutch criminal law [44]. (I do not claim that it is an accurate description of Dutch law.) Figure 11.1 shows the activity diagram of this workflow. Note that the diagram contains an OR node (diamond) with more than one incoming and outgoing edge,

even though in Chapter 3 we stated that an OR node must have either one incoming or one outgoing edge. Using two OR nodes, the same construct be modelled, but introducing more nodes would make the diagram harder to understand.

The workflow starts when the registration desk of the PPS receives an official report and a corresponding report of seizure of some goods from the police. Among others, the registration desk checks whether all the necessary documents are present. Next, the public prosecutor must make a decision what to do which each of the seized goods. To keep the presentation simple, in Figure 11.1 we assume that only one good has been seized. Seizure of multiple goods can be modelled using the dynamic concurrency construct; we have not done this because such a construct does not introduce potential new errors, as the different subworkflows for the different goods are isolated from each other. If the public prosecutor does not make a decision within one week, a reminder is sent to him.

The public prosecutor can make the following decisions. If the person that has been seized has not parted with the good, the public prosecutor can decide to confiscate the good (if the good is illegal), or, if the good is stolen, to give it back to the person entitled to the good, the rightful claimant, (if the good is not needed anymore for the lawsuit) or to let it be kept by the rightful claimant (if the good is needed), or to keep the good in custody. The public prosecutor sends his decision to the person whose goods have been seized. If that person does not lodge a complaint within two weeks, the decision is carried out in some activity whose name corresponds to the made decision. If the good is kept by the rightful claimant, then as soon as the good is not needed anymore, the good is officially given back to that person, modelled by event give back good.

If the person whose goods have been seized does lodge a complaint, the court sitting in chambers judges the claim. Then the decision is not yet carried out. If the person does not agree with the judgement of the court sitting in chambers he may appeal to the court of cassation. If he does not appeal within two weeks, the judgement becomes final and the decision is carried out. The decision of the court of cassation is always final; that decision is subsequently carried out.

If the person whose goods have been seized has parted with the good, the following decisions are possible. If the good has been stolen, then the public prosecutor can either decide to give the good back to the rightful claimant, or, if the rightful claimant is unknown, to keep the good in custody until that person becomes known.

If the good has not been stolen, then the public prosecutor can confiscate the good and withdraw it from social and economic life. If the confiscated good is illegal, it is destroyed. Otherwise, it is kept in custody: a deposit order is given. Since it may be expensive to keep confiscated goods in custody, under some circumstances the public prosecutor may decide to sell or destroy the good. The decision is carried out by the custodian.

The custodian may also decide himself not to keep a good in custody any longer. But then he must ask the public prosecutor for authorisation to carry out

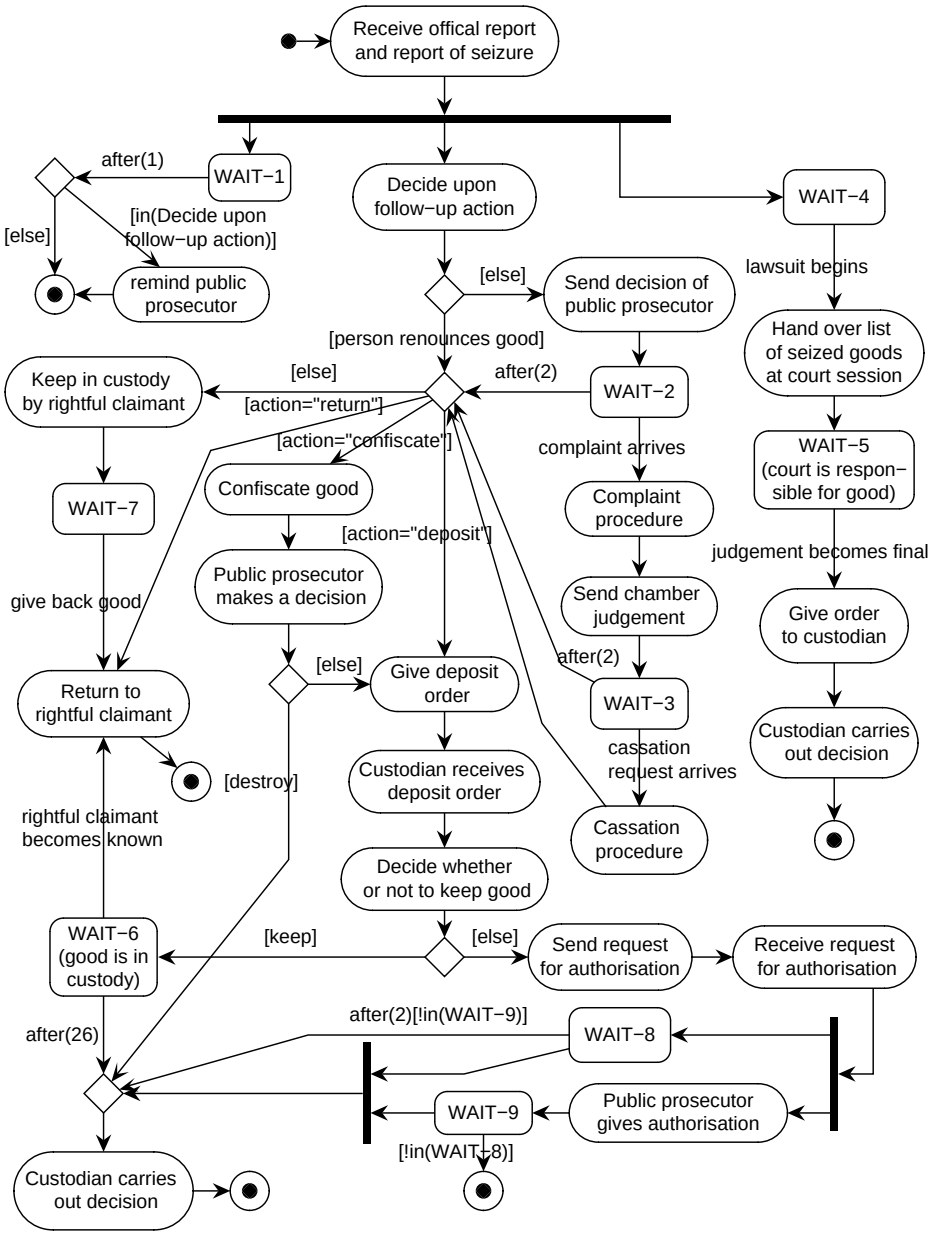



**Figure 11.1** *Workflow seizure of goods*

the decision, for example to destroy or sell a good that he keeps in custody. If the public prosecutor does not respond within two weeks, the custodian may carry out the decision. If a good has been in custody for over two years, the custodian can decide to sell or destroy the good without having to ask for authorisation. In Figure 11.1 we have modelled this as half a year.

As soon as the lawsuit begins that deals with the offence for which the goods were seized, the court becomes responsible for the goods. The public prosecutor then must ask the court for authorisation if he wishes to make a decision about the seized goods. This cannot be modelled well in activity diagrams; we simply assume that this happens.

Most of the constructs used in Figure 11.1 are standard, except the race construct after node Receive request for authorisation[1]. The race is between two events that can occur in parallel. The first event is the termination event of the authorisation activity of the public prosecutor. The second event is a timeout that is generated two weeks after the authorisation activity has become enabled. The event that occurs first wins the race and triggers activity *Custodian carries out decision*. The event that occurs later is ignored; it does not trigger *Custodian carries out decision*. So activity *Custodian carries out decision* is only executed once. The race construct should not lead to a deadlock, i.e., the workflow should terminate properly.

To fully grasp the meaning of the construct, let us consider the three possible orders of event occurrences:

- If the authorisation activity terminates before the timeout occurs, node WAIT-9 is entered while node WAIT-8 is already active. Then the only enabled hyperedge is the one leaving both WAIT-8 and WAIT-9 and entering node Custodian carries out decision. (The hyperedge with the after label is not enabled due to its guard condition.) If that enabled hyperedge is taken, node WAIT-8 is left, disabling generation of the timeout.

- If the timeout occurs before the authorisation activity terminates, then the hyperedge with label after is taken and node Custodian carries out decision is entered. If the authorisation activity subsequently terminates, node WAIT-9 is entered and then the final node.

- If the authorisation activity terminates simultaneously with the occurrence of the timeout, the hyperedge with label after is taken and the hyperedge entering WAIT-9 is taken. The sequel is similar to the previous item.

The race construct is probably the most complex construct used in workflow modelling. To illustrate why the race construct is so difficult, in Figure 11.2 we show some flawed variants of the race construct. The construct in Figure 11.2(a)

[1]The race construct is similar to the discriminator workflow pattern that Van der Aalst et al. [7] have found in existing workflow specifications.

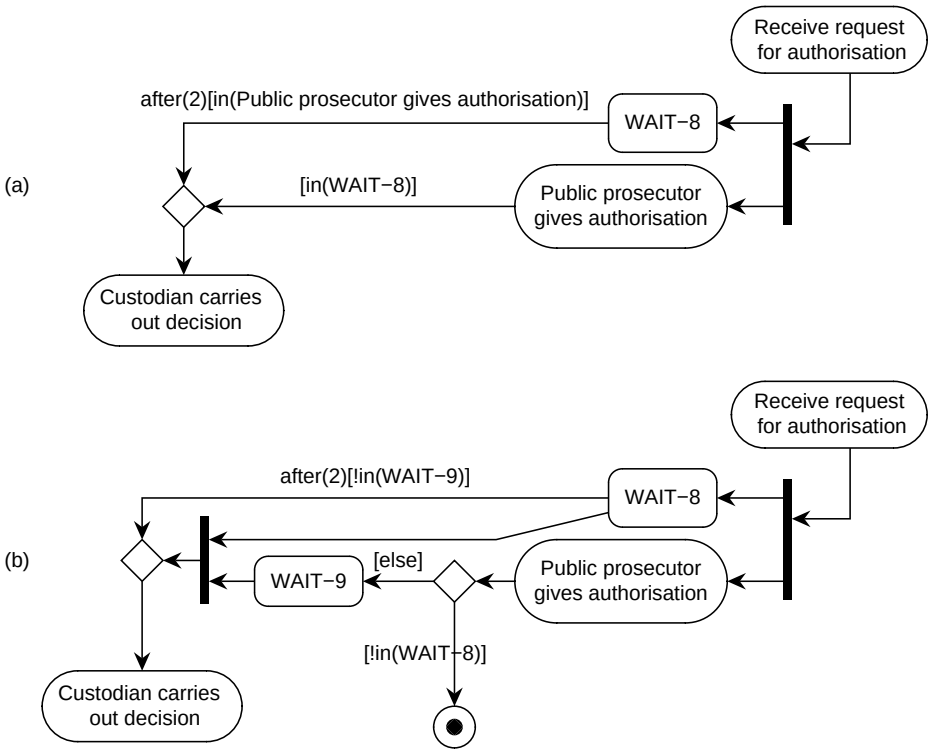



**Figure 11.2** *Two flawed race constructs*

is flawed because it ignores that the authorisation activity can terminate simultaneously with the timeout. According to the activity diagram in Figure 11.2(a), in that case node Custodian carries out decision is entered twice, meaning that activity *Custodian carries out decision* is executed twice. This is not the intended meaning of the race construct.

The construct in Figure 11.2(b) is flawed because it too ignores that the authorisation activity can terminate simultaneously with the timeout. If the authorisation activity terminates simultaneously with the timeout, both node Custodian carries out decision and WAIT-9 are entered. Next, WAIT-9 cannot be left, so the workflow does not terminate properly.

Both flaws were found by model checking property $\mathsf{F}\,\mathsf{G}\,\mathit{final}$.

**Requirements.**[2] We check two general and three ad-hoc requirements. We first observe that the activity diagram fails to satisfy the constraints mentioned in

[2] As in the previous chapter, strong fairness conditions and the conjunction of predicate *stable* with atomic propositions is not shown explicitly.

Theorem 6.5. Constraint C4(b) on page 91 is not fulfilled, because there is a completion hyperedge, leaving WAIT-9, whose guard contains predicate in. Moreover, there is a cycle in the $\prec$ relation due to the completion hyperedge leaving WAIT-9 and the external hyperedge leaving WAIT-8. Thus, we cannot apply Theorem 6.5.

However, we now show that for this particular example the requirements-level semantics and implementation-level semantics have similar behaviour, i.e., we can prove for this particular example a similar theorem as Theorem 6.5, even though Constraint C4(b) is not fulfilled. First, we observe that every RLS superstep in which the completion hyperedge leaving WAIT-9 is taken, starts if activity node Public prosecutor gives authorisation terminates. If the current configuration does not contain WAIT-8, or if the after event in WAIT-8 does not occur, the two semantics have similar behaviour. If the current configuration does contain WAIT-8 and the after event occurs, under the requirements-level semantics configuration [Custodian carries out decision,final] is reached. Under the implementation-level semantics, this configuration can be reached by first processing the after event.

So for this particular example the requirements-level semantics and implementation-level semantics have similar behaviour, i.e., for this particular example we can prove a theorem similar to Theorem 6.5 without using Constraint C4(b). Thus, by Theorem 10.1, we can use $\mathrm{CTL}_r^*$ formulas and obtain results valid for both the RLS and ILS. We apply all of the four reduction rules defined in Section 10.6.2 using TCM. TCM cannot abstract from variable action, so rule 3 cannot be applied, because some of the edges whose guards refer to action, leave non-activity node WAIT-2. However, for this particular example action can be safely abstracted from, i.e., by abstracting no new configurations are introduced. We therefore remove variable action by hand from the activity diagram. Figure 11.3 shows the reduced activity diagram for the first requirement; the reductions for the other requirements are similar.

First we check two general requirements. The first requirement is that the activity diagram does not deadlock.

(R1): $\mathsf{F}\,\mathsf{G}\,\mathit{final}$

This requirement is true.

The second requirement is that the activity diagram does not diverge.

(R2): $\mathsf{G}\,\mathsf{F}\,\mathit{stable}$

This requirement is true. The other general requirements mentioned in Chapter 10 we verified to be true as well.

Next, we verify as third requirement whether in WAIT-6 it is possible to reach Custodian carries out decision. This is a kind of sanity check on the model [115].

(R3): $\mathsf{AG}\ (\mathsf{in}(\text{WAIT-6}) \Rightarrow \mathsf{EF}\ \mathsf{in}(\text{Custodian carries out decision}))$

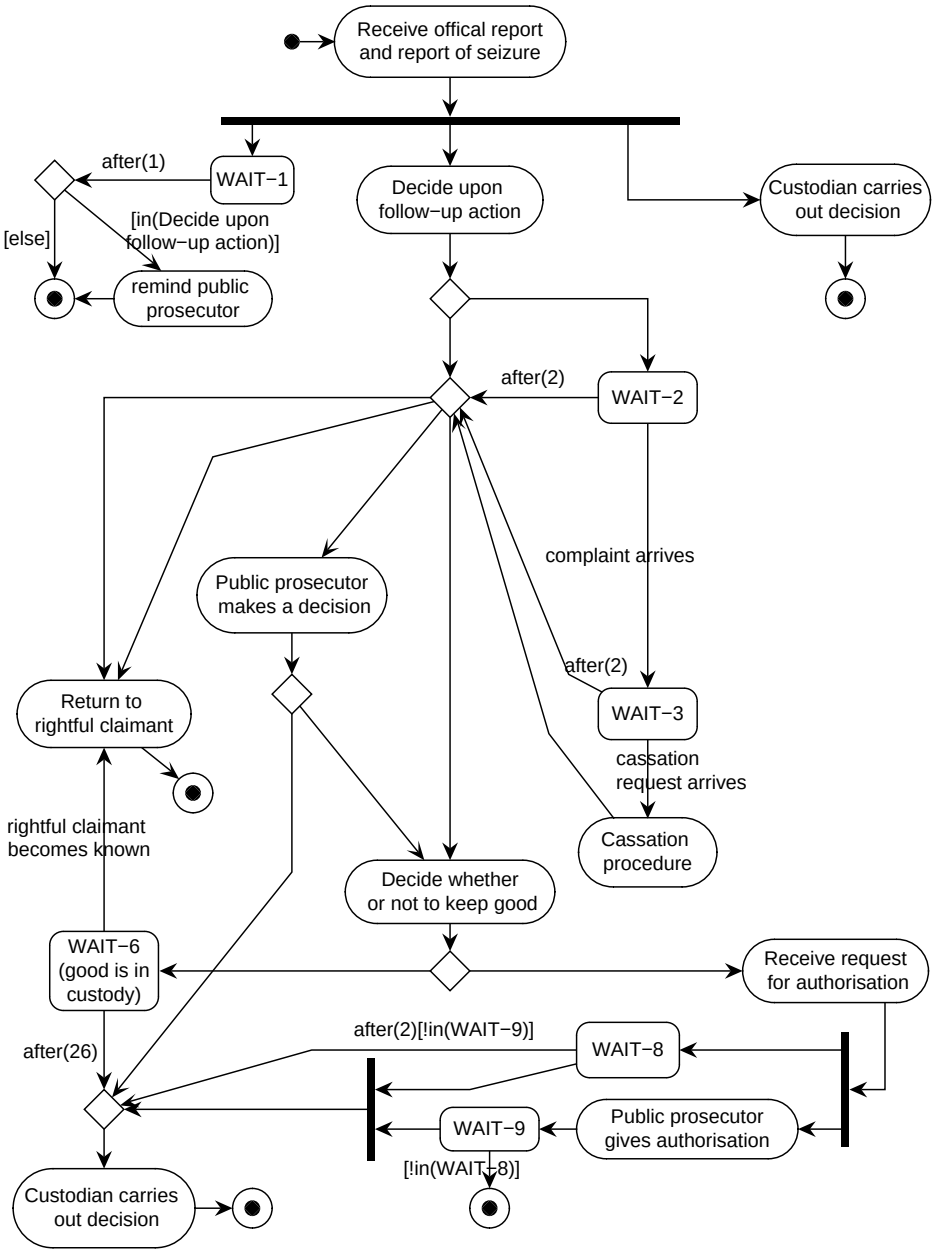



**Figure 11.3** *Reduced activity diagram of Figure 11.1. The requirement that is verified is* $\mathsf{F}\,\mathsf{G}\,final$

| Requirement | Result | Reachable states[3](nr) | Time (sec) | BDD nodes (nr) | Memory (kB) |
|---|---|---|---|---|---|
| R1 | True | 656 | 66.55 | 185656 | 16547 |
| R2 | True | 121 | 2.66 | 36405 | 4279 |
| R3 | True | 502 | 36.56 | 80137 | 11619 |
| R4 | True | 502 | 36.91 | 109992 | 12179 |
| R5 | False | 136 | 3.82 | 65708 | 5271 |
| R5' | True | 305 | 15.43 | 94417 | 8963 |

**Table 11.1** *Resources used by* NuSMV$_{fair}$

This requirement is true.

We proceed with some ad-hoc requirements. The fourth requirement states that nodes WAIT-6, WAIT-8, WAIT-9 and Custodian carries out decision are not active simultaneously. This formalises that the race construct only starts one copy of activity *Custodian carries out decision*.

$$\text{(R4):} \quad \neg\, \mathsf{F}\,(\,\mathsf{in}(\text{WAIT-6}) \wedge \mathsf{in}(\text{WAIT-8}) \wedge \mathsf{in}(\text{WAIT-9}) \wedge \mathsf{in}(\text{Custodian carries out decision})\,)$$

The requirement is true.

The fifth requirement states that a confiscated good is not returned to its owner.

$$\text{(R5):} \quad \mathsf{F}\,(\,\mathsf{in}(\text{Confiscate good})\,) \Rightarrow \neg\, \mathsf{F}\,(\,\mathsf{in}(\text{Return to rightful claimant})\,)$$

The requirement is not true. Figure 11.4 shows the counterexample returned by TCM. From node Confiscate good, via node Give deposit order, node Return to rightful claimant is reachable. The activity diagram can be repaired by adding guard [!action="confiscate"] to the label of the hyperedge leaving WAIT-6 and entering Return to rightful claimant. For the repaired activity diagram the requirement is true.

We conclude by giving the performance statistics for verification of these requirements; see Table 11.1. The analysis was performed on a PC with a Pentium III 450 MHz processor with 128Mb of RAM under Red Hat Linux 6.0. Note that R5 is verified twice; the first verification is done on the original activity diagram; the second verification is done on the repaired activity diagrams as sketched above. For the second verification of R5, the reduced model is much larger, as variable action cannot be abstracted from any longer!

[3]The original model has 554,470 states. With a timeout of two years the model has 1,093,610 states.

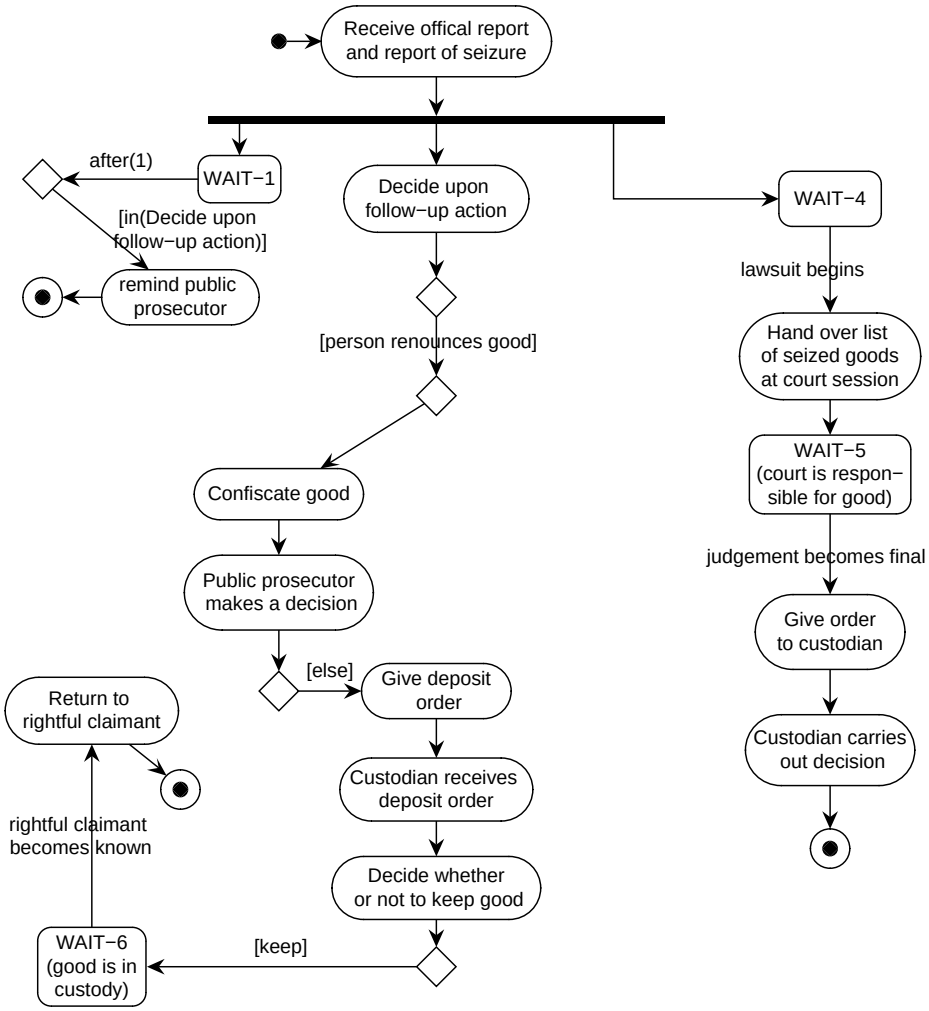



**Figure 11.4** *Counterexample for requirement* R5

## 11.2 Order procedure within IT department

In this section we model the workflow of ordering a good within the IT department of a company. It is based upon an existing procedure at Océ [32] which has been modelled using the Logistic Model [26].

**Workflow description.** Figure 11.5 shows an activity diagram that describes the workflow. The workflow starts by checking the request for a good. If the request is not ok, it is returned. Otherwise, next the request is split into subrequests. A subrequest is either a hard or software request or a computer relocation request. We here assume that there is at most one hard or software request and at most one computer relocation request. (Relaxing this constraint would require dynamic concurrency nodes; as in the previous example, these do not impact verification, but would blow up the state space.)

A hard or software request is handled by first making a specification. If the required hardware is already available in some left-over computer in stock, the hardware can be installed after removing it from that computer. Otherwise, the hardware must be ordered, i.e., a formal request must be made. Then, a form is filled in. Next it is checked whether the request counts as investment, i.e., the investment requires a large amount of money. If so, an additional form has to be filled in.

Next, the head of the IT department needs to approve the request. If he disapproves the request, the subworkflow stops. If he approves, an order is made and sent to the supplier by the purchase department. If the supplier is not known, some administrative information, among others a supplier number, is created.

If the order arrives, the delivered product can be installed. If the bill arrives, it is paid.

If the request is a computer relocation request, the request is inspected and it is checked whether a new ethernet access point is needed. If not, the computer is relocated. Otherwise, an estimation of the costs must be made. If the costs exceed a certain threshold, the request counts as an investment, and the head of the IT department must approve of the request. If he disapproves the request, the subworkflow stops.

Next, a new ethernet access point is made by a supplier. The documentation of the building is updated with the location of the new ethernet access point. Then the computer can be relocated. Meanwhile, the supplier sends his bill to the organisation and the organisation pays the bill.

**Requirements.** We first observe that the activity diagram satisfies the constraints mentioned in Theorem 6.7 and Theorem 10.1. Thus we can use $\mathrm{CTL}^*_r$ formulas and obtain results valid for both the RLS and ILS. We apply the four reduction rules defined in Section 10.6.2. Figure 11.6 shows the reduced activity diagram for the first two requirements; the other reductions are similar.

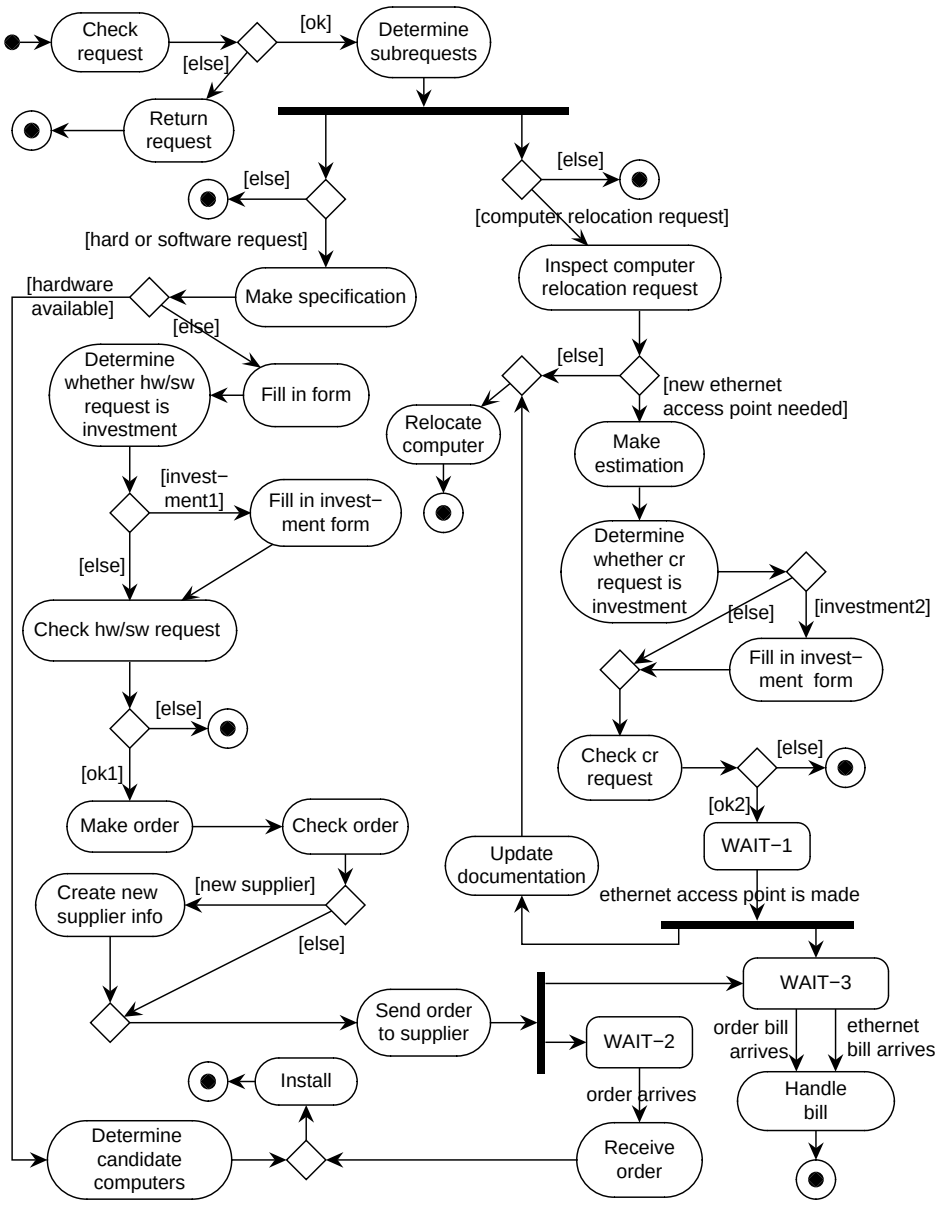



**Figure 11.5** *Workflow for handling order requests*

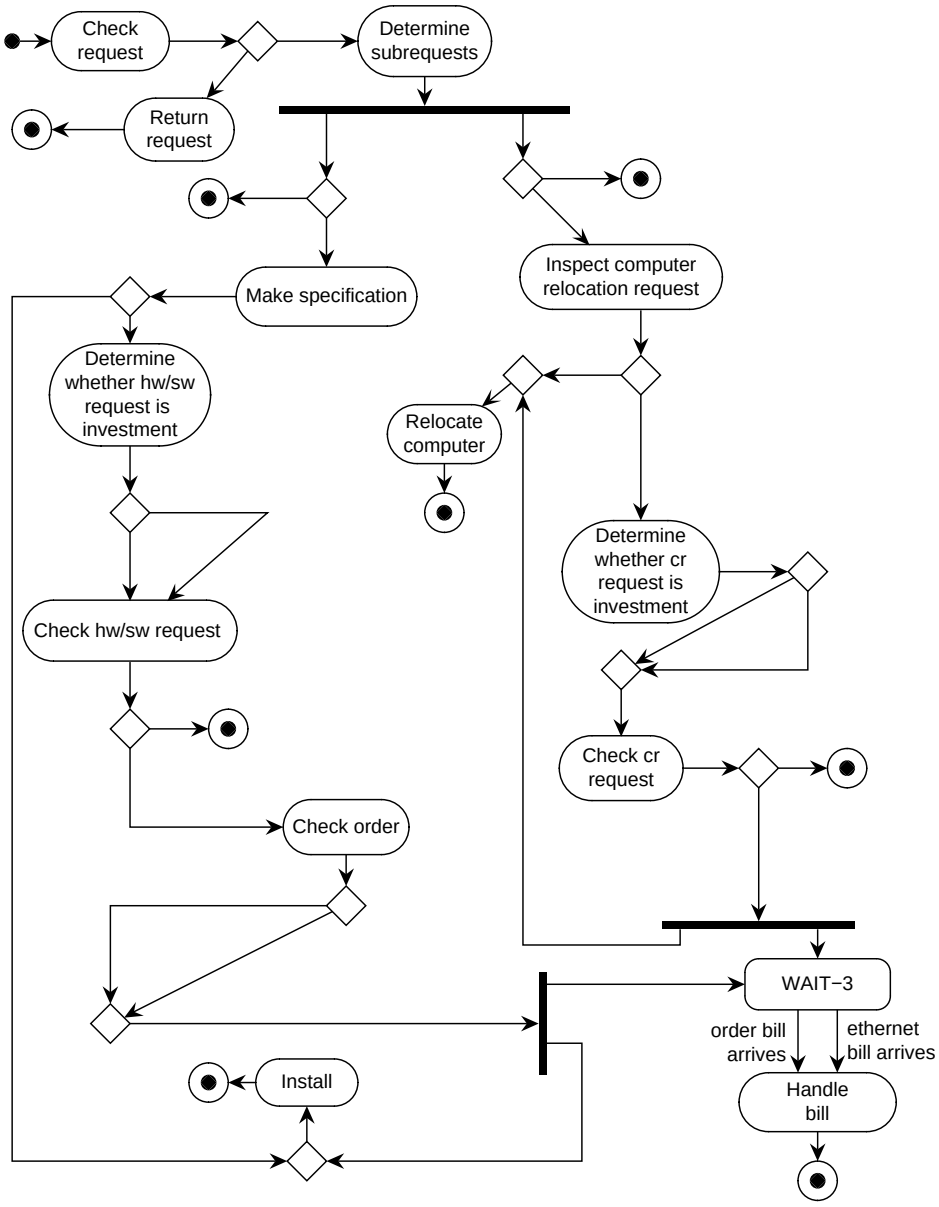



**Figure 11.6** *Reduced activity diagram of Figure 11.5*

| Requirement | Result | Reachable states[4](nr) | Time (sec) | BDD nodes (nr) | Memory (kB) |
|---|---|---|---|---|---|
| R1 | True | 458 | 34.59 | 230912 | 13863 |
| R2 | True | 162 | 4.33 | 61288 | 5295 |
| R3 | True | 324 | 16.02 | 69577 | 8627 |
| R4 | True | 323 | 17.87 | 149256 | 10247 |

**Table 11.2** *Resources used by* NuSMV$_{fair}$

Again, we first check two general requirements. The first requirement is that the activity diagram does not deadlock.

(R1): $\mathsf{F}\,\mathsf{G}\,\mathit{final}$

This requirement is true.

The second requirement is that the activity diagram does not diverge.

(R2): $\mathsf{G}\,\mathsf{F}\,\mathit{stable}$

This requirement is true.

The third requirements states that it is possible to relocate a computer if a computer relocation request has been made. This is a sanity check on the workflow specification.

(R3): $\mathsf{AG}\ (\mathsf{in}(\mathsf{Inspect\ computer\ relocation\ request}) \Rightarrow \mathsf{EF}\,\mathsf{in}(\mathsf{Relocate\ computer}))$

This requirement is true.

Finally, the fourth requirement states that executing activity *Make order* implies that sometime in the future activity *Install* is executed. We formalise this by referring to the corresponding activity nodes.

(R4): $\mathsf{G}\,(\mathsf{in}(\mathsf{Make\ order}) \Rightarrow \mathsf{F}\,\mathsf{in}(\mathsf{Install}))$

This requirement is true.

The performance statistics for verification of these requirements are shown in Table 11.2. As before, the analysis was performed on a PC with a Pentium III 450 MHz processor with 128Mb of RAM under Red Hat Linux 6.0.

[4]The numbers refer to the reduced model. The original model has around 100,000 states.

## 11.3 Lessons learned

Before we discuss the lessons learned, we make some caveats concerning the case studies. First, I have by now gained experience in using activity diagrams to specify workflows. Persons that are less experienced might come up with workflow specifications that I would not even consider, because I know beforehand that they are flawed. It would therefore be interesting to see how workflow modellers not used to activity diagrams would model the workflows that were discussed in this chapter.

Second, the workflows are of moderate size and do not seem overly complex. It remains to be seen how our verification approach performs for larger and more complex workflows. However, we observe that reduction rule 2 ensures that the scale factor is linear in the number of events, whereas if rule 2 is not used the scale factor is exponential in the number of events.

Third, the workflows were constructed from existing workflow descriptions in other workflow notations. The workflows were not modelled immediately in activity diagrams. Hence, it might be that certain aspects of the workflows would have been modelled in a different way if activity diagrams had been used from the outset. This means that the workflow models in this chapter might not be completely realistic.

Taking into account these caveats, we now discuss what we have learned. First, writing properties is difficult. If I verified a property to be false, quite frequently the property was wrong, not the model. Writing correct properties is an art in itself. Moreover, writing CTL properties is considerably harder than writing LTL properties.

Second, CTL properties turned out to be rather weak for our purposes: they say something about possibilities ("what can happen"), not about necessities ("what will happen"). For reactive systems, including workflow systems, possibility properties are not really interesting, unless as a sanity check on the model that certain nodes and hyperedges are indeed reachable. This agrees with the view of Lamport [115].

Third, temporal logic properties and activity diagrams are complementary, not substitutes for one each other. Some constructs are more easily modelled by a property, others more easily by an activity diagram. For example, the requirement that some activities $A$ and $B$ should always be done together in the same run is easy to express in temporal logic, but not so easy in activity diagrams, as there is no order information provided as to which activity executes first. And the requirement that some set of activities $A$, $B$, and $C$ should be done in a certain order, say $A$ before $B$ before $C$, and that the sequence should be repeated until a certain condition holds is easy to model in activity diagrams but difficult in temporal logic (for example, property $\mathsf{in(A)} \wedge \mathsf{F}\,(\mathsf{in(B)} \wedge \mathsf{F}\,\mathsf{in(C)}\,)$ does not model the iteration).

Fourth, by model checking we were able to find hidden assumptions. For example, the need for strong fairness constraints may seem obvious at first sight, but the counterexample returned by the model checker, showing that irrelevant events can prevent other events from occurring (see Section 10.3) came as a surprise to me. The point is that things that seem obvious at hindsight are not so obvious at foresight. Model checking proved to be useful for gaining insight.

Fifth, the original workflow descriptions did not list any requirements, not even in natural language. So I had to invent myself several requirements. I conjecture that for most workflows that are used in real-life, their requirements are not documented at all. This makes the validation of workflow specifications difficult, if not impossible.

## 11.4 Conclusion and future work

We have verified two activity diagrams that specify workflows that are in use in real organisations. The size of both workflows is moderate. All requirements could be verified in reasonable time, after having applied the reduction rules as defined in Chapter 10. This shows that reasonable complex workflow specifications of moderate size can be verified efficiently using a state of the art computer.

To fully ascertain whether or not our approach scales up to larger examples, some more and especially larger case studies need to be done. But since one of the reduction rules (rule 2) in Section 10.6 transforms the exponential blowup of the state space into a polynomial blowup, we expect that larger examples can also be handled.

Future work includes applying the verification to more complex workflow specifications that do not fit on one sheet of paper. The correctness of such workflows cannot be determined by visual inspection anymore, and it is in this case that verification really pays off. Next, the effectiveness of our verification should be studied in more detail. Temporal logic is hard to use for non-experts, and even experts sometimes make mistakes in it. A high-level specification language for properties that is easy to use for non-experts should be defined.

# Chapter 12

# Conclusion and future work

## 12.1 Conclusion

In the introduction we have stated that the goal of this thesis is to define a semantics of activity diagrams that is suitable for workflow modelling. The semantics should allow verification of functional requirements using model checking. The semantic should be accurate, yet easy to analyse by a model checker. We now discuss whether this goal has been met.

We have proposed two formal semantics for activity diagrams. As an activity diagram is used as workflow specification, and a workflow specification prescribes how a workflow system should behave, both semantics are defined and motivated in terms of workflow systems. In both semantics a workflow system is viewed as a reactive system. Both semantics support the modelling of event-driven behaviour, data, real time and loops, thus supporting the specification of complex, realistic workflows. The requirements-level semantics assumes that workflow systems are infinitely fast with respect to their environment and react immediately to input events (perfect synchrony) whereas the implementation-level semantics does not. Since the implementation-level semantics stays close to the way a workflow system actually operates, it is accurate. The requirements-level semantics, on the other hand, is not so accurate, since it assumes perfect synchrony. But for a model checker the requirements-level semantics is easier to analyse than the implementation-level semantics, since the state space in the requirements-level semantics is much smaller than the one in the implementation-level semantics.

For activity diagrams satisfying the constraints identified in Chapter 6 and requirements in $\text{CTL}^*_r$ (Chapter 10), the requirements-level semantics gives the same outcome as the implementation-level semantics. Thus, for such activity diagrams and requirements, the requirements-level semantics is as accurate as the implementation-level semantics. In that case the requirements-level semantics is both accurate and easy to analyse by a model checker. This class of activity

diagrams and requirements is fairly broad, but of course rules out certain activity diagrams and requirements. For example, requirements cannot refer to events.

Our approach of defining two semantics, rather than one, acknowledges that activity diagrams are first and foremost informal, like the other UML diagrams. Different people may interpret the same diagram in completely different ways. Our approach recognises that UML activity diagrams can be interpreted in different ways, and puts this difference in a more formal setting by relating two completely different formal semantics. We do not know of any other work in which different formal semantics for the same diagram are related with each other.

Comparing both our semantics with Petri nets, both semantics are admittedly more difficult to analyse than a Petri net token-game semantics. But this is because the paradigm of Petri nets presupposes that systems are closed and active, whereas the paradigm of statecharts presupposes that systems are open and reactive. Clearly, open reactive systems are more difficult to understand than closed active systems. Since workflow systems are reactive systems, we think that the Petri net token-game semantics is not accurate enough for workflow modelling. We therefore have taken the statechart step semantics as starting point of our two semantics.

We have implemented the requirements-level semantics in an activity diagram editing tool and interfaced it with a model checker. Thus we have shown that the requirements-level semantics can be used for model checking. The (stable) implementation-level semantics can be used for model checking in a similar way. The tool supports verification of workflow models that have event-driven behaviour, data, real time and loops. Existing verification approaches do not support all these features. The feedback of the model checker is presented in terms of the original activity diagram. Thus, the tool completely hides the model checker from the user of the tool, allowing the model checker to be used by people that are unfamiliar with formal methods.

We were able to verify some non-trivial real-world examples with the tool. As usual in model checking, the state space explosion prevented that model checking could be applied straightaway to these examples. Instead, we had to use reduction rules to reduce the state space of the examples. To fully validate the semantics and our verification approach, some more and some larger case studies need to be done. The presented results merely indicate feasibility. However, the presented results are encouraging in this respect. We do not expect problems with scalability, as one of the reduction rules transforms the exponential blowup of the state space into a polynomial blowup.

## 12.2 Summary of main contributions

This thesis has made several contributions. We list the main ones.

- Two completely different formal semantics for UML activity diagrams are

defined.

- Both semantics are motivated entirely by the domain of workflow modelling. Other approaches that formalise UML activity diagrams or workflow models just define or take a formal semantics without providing any motivation. Formal semantics are rarely justified by the intended application domain.
- All the design choices that are made are listed explicitly. Thus, our semantics can be applied to another domain by validating all design choices on that domain.
- The two formal semantics are related by showing for which activity diagrams and for which requirements the two semantics are similar.
- A realistic justification of the perfect synchrony hypothesis is provided by relating a semantics that satisfies perfect synchrony with a semantics that does not satisfy perfect synchrony. The existing justification puts an assumption upon the system and environment (the system should be fast enough w.r.t. pace of change of the environment). Our justification allows the system to be slower than the environment.
- The thesis shows that a statechart-like step semantics can be given to a notation with a Petri net-like syntax.
- The thesis clarifies the difference between statecharts and Petri nets by comparing their semantics, rather than their syntax. We have shown that the Petri net token-game semantics cannot model reactive behaviour. Since workflow systems are reactive systems, we think that the Petri net token-game semantics is not accurate enough for modelling workflows.
- We have developed a powerful verification tool for realistic real-life workflow models, written in UML activity diagrams, that have event-driven behaviour, data, real time, and loops.
- The model checker used by the verification tool is completely hidden from the user, thus increasing user friendliness of the tool. In particular, feedback of the model checker is translated in terms of the activity diagram.
- Our tool is the first verification tool for UML activity diagrams.
- A restriction on CTL$^*$ is defined that extends the restriction proposed by Lamport [113]. The restriction can be used in other approaches in which at the concrete lower level variables do not change instantaneously, but can have an inconsistent intermediate value that should not be observed.
- Several reduction rules for activity diagrams have been defined. The reduction rules ensure that the reduced activity diagram preserves the requirement to be verified. One reduction rule transforms the exponential blowup of the

state space into a polynomial blowup. Thus, this rule considerably improves the scalability of the verification approach.

- We have applied model checking, in particular strong fairness model checking, to real-life workflow specifications. Strong fairness model checking is necessary to accurately verify requirements on workflow models that have loops.

## 12.3 Future work

There are several extensions of our work possible. First, the semantics could be extended to deal with for example object flows and interrupt regions, as sketched in Chapter 7. Moreover, the semantics could be updated to UML 2.0, once that has been endorsed by OMG. Also case management, in which several workflow instances can synchronise and communicate with each other, poses an interesting challenge. The current semantics deals with one workflow instance in isolation.

Second, our semantics, especially the implementation-level semantics, could be used to generate code for supporting workflows at run-time. The code implements workflow management functionality. This can be particular useful for inter-organisational workflows in the setting of e-commerce [80], as e-commerce frameworks like ebXML [153] have adopted UML activity diagrams as modelling language. To deal with inter-organisational workflows our semantics probably has to be extended. We then have to model parties. Each party executes its own part of a workflow. Parties cooperate to deliver a combined service.

Third, we have identified commonalities in behaviour between two completely different semantics. We could extend this by identifying a whole class of different semantics that have common behaviour. The ultimate benefit of such an identification is that the precise semantics attached to activity diagrams can be left open, as long as the semantics used satisfies some common constraints. Then different tools do not have to implement exactly the same semantics, but nevertheless can cooperate with each other, even though they use different semantics. Putting this one step further, then the UML standard can leave open the actual semantics of activity diagrams, and instead put some constraints that a semantics must satisfy.

Fourth, the verification tool could be enhanced in several ways. Some of the reduction rules have been implemented in the tool; the remaining reduction rules should also be implemented. Next, a more abstract property language should be defined that is preferably graphical and in terms of the diagram. This allows for a more user-friendly and intuitive specification of user requirements. Finally, the tool could be extended with a simulator that allows animation of activity diagrams. Then the user can play the error scenarios that are returned by the model checker. Furthermore, animation of activity diagrams allows the user to better understand the behaviour of activity diagrams according to our semantics.

# Appendix A

# Notational conventions

Throughout the thesis we use a variant of the Z notation [148]. Most of the notation is standard set theory, and therefore straightforward. In this chapter we give a short introduction to some of the unfamiliar parts of Z that we use frequently in this thesis, in particular bags and sequences.

The symbol $\bullet$ separates a declaration from an expression. For example, the property that every natural number is nonnegative is expressed in Z as:

$$\forall\, n \in \mathbb{N} \bullet n \geq 0$$

Symbol $\longrightarrow$ denotes a function, not an implication. An implication is denoted by symbol $\Rightarrow$. Symbol $\rightarrowtail\!\!\!\!\!\twoheadrightarrow$ denotes a bijective function.

Notation $x/val$ represents substitution of the value of variable $x$ by value $val$. Valuation $\sigma[x/val]$ assigns to variable $x$ value $val$ and to every other variable $y$, $y \neq x$, the value $\sigma(y)$. Symbol & denotes a bulk update:

$$\sigma[\&_{x \in X}\, x/val_x] \quad \stackrel{\text{df}}{=} \quad \sigma[x_1/val_1, .., x_n/val_n]$$

where $n = \#X$. (Function $\#$ returns the size of a set.)

A bag or a multiset is a set in which an element can occur more than once. A bag $b$ on a set $X$ is a function from $X$ to natural numbers. The set of all bags on a set $X$ is denoted $\text{bag}\, X$, which abbreviates

$$\text{bag}\, X \quad \stackrel{\text{df}}{=} \quad X \longrightarrow \mathbb{N}_1$$

where $\mathbb{N}_1$ is the set of all strictly positive natural numbers, i.e., excluding 0. So $b : \text{bag}\, X$ is equivalent to $b \in X \longrightarrow \mathbb{N}_1$. The number associated with each element $x \in X$ represents how many times $x$ is in the bag. A set is a bag in which every element can occur at most once.

A bag on set $X$ can be represented textually by listing between square brackets the elements of $X$ as many times as they are in the bag. For example, given a set

$\{x_1, x_2\}$, bag $b = \{x_1 \mapsto 2, x_2 \mapsto 1\}$ can be written as $[\, x_1, x_1, x_2 \,]$, $[\, x_1, x_2, x_1 \,]$, or $[\, x_2, x_1, x_1 \,]$.

Given a bag $b$ on set $X$, the number of times an element $x \in X$ occurs in $b$ is denoted $b \sharp x$. Union of two bags $b_1$, $b_2$ on set $X$ into a new bag $b$, denoted $b = b_1 \uplus b_2$, is defined as the addition of the number of times each element $x \in X$ occurs in the individual bags. So $b \sharp x = b_1 \sharp x + b_2 \sharp x$. For example, if $b_1(x) = 2$ and $b_2(x) = 4$ then $b(x) = 6$. Bag difference, denoted by $\uplus$, is defined similarly.

A bag $b_1$ on set $X$ is contained in bag $b_2$ on $X$, denoted $b_1 \sqsubseteq b_2$, iff for each element $x \in X$ the number of times $x$ occurs in $b_1$ is lower than or equal to the number of times $x$ occurs in $b_2$:

$$b_1 \sqsubseteq b_2 \quad \stackrel{\text{df}}{\Leftrightarrow} \quad \forall\, x \in X \bullet b_1 \sharp x \;\leq\; b_2 \sharp x$$

A sequence $s$ on set $X$ is a list of elements of $X$. It is denoted by a function from natural numbers to $X$. The set of sequences on $X$ is denoted $\mathrm{seq}\, X$.

Concatenation of two sequences $s$ and $t$ is written $s \frown t$. The concatenation contains the elements of $s$ followed by the elements of $t$.

# Bibliography


[1] W. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management*. Lecture Notes in Computer Science 1806. Springer, 2000.

[2] W.M.P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[3] W.M.P. van der Aalst. Workflow verification: Finding control-flow errors using Petri-net-based techniques. In Aalst et al. [1], pages 161–183.

[4] W.M.P. van der Aalst. Personal communication, 2001.

[5] W.M.P. van der Aalst and T. Basten. Inheritance of workflows: An approach to tackling problems related to change. *Theoretical Computer Science*, 270(1-2):125–203, 2001.

[6] W.M.P. van der Aalst, K.M. van Hee, and G.J. Houben. Modelling workflow management systems with high-level Petri nets. In G. De Michelis, C. Ellis, and G. Memmi, editors, *Proc. 2nd Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, pages 31–50, 1994.

[7] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced workflow patterns. In O. Etzion and P. Scheuermann, editors, *Proc. CoopIS 2000*, Lecture Notes in Computer Science 1901, pages 18–29. Springer, 2000. An extended version has appeared as: QUT Technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane, 2002.

[8] N.R. Adam, V. Atluri, and W. Huang. Modeling and analysis of workflows using Petri nets. *Journal of Intelligent Information Systems*, 10:131–158, 1998.

[9] G. Agha, F. Decindio, and G. Rozenberg, editors. *Concurrent Object-Oriented Programming and Petri Nets*. Lecture Notes in Computer Science 2001. Springer, 2001.

[10] A. Agostini and G. de Michelis. A light workflow management system using simple process models. *Computer Supported Cooperative Work*, 9(3/4):335–363, 2000.

[11] Alcatel, Computer Associates, Enea Business Software, Ericsson, Hewlett-Packard, I-Logix, IONA, International Business Machines, Jaczone AB, Inc. Kabira Technologies, Motorola, Oracle, Rational Software, Softeam, Telelogic AB, Unisys, and WebGain. Update to the U2 partners initial submission for UML2 superstructure, 2001. Object Management Group document ad/01-11-01. Available at `http://www.omg.org`.

[12] Alcatel, I-Logix, Kennedy-Carter, Inc. Kabira Technologies, Inc. Project Technologies, Rational Software, and Telelogic AB. Action semantics for the UML, 2001. Object Management Group document ad/01-03-01. Available at `http://www.omg.org`.

[13] L. Apvrille, P. de Saqui-Sannes, C. Lohr, P. Sénac, and J.-P. Courtiat. A new UML profile for real-time system formal design and validation. In M. Gogolla and C. Kobryn, editors, *Proc. ≪UML≫ 2001*, volume 2185 of *Lecture Notes in Computer Science 2185*, pages 287–301. Springer, 2001.

[14] E. Asarin, O. Maler, and A. Pnueli. On discretization of delays in timed automata and digital circuits. In R. de Simone and D. Sangiorgi, editors, *Proc. CONCUR '98 – Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 470–484. Springer, 1998.

[15] P.C. Attie, M.P. Singh, A.P. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In R. Agrawal, S. Baker, and D.A. Bell, editors, *Proc. International Conference on Very Large Data Bases (VLDB '93)*, pages 134–145. Morgan Kaufmann, 1993.

[16] P. Baldan, A. Corradini, H. Ehrig, and R. Heckel. Compositional modeling of reactive systems using open nets. In K.G. Larsen and M. Nielsen, editors, *Proc. CONCUR 2001 – Concurrency Theory*, Lecture Notes in Computer Science 2154, pages 502–518. Springer, 2001.

[17] R.N. Bateson. *Introduction to Control System Technology.* Prentice Hall, 6th edition, 1999.

[18] M. von der Beeck. A comparison of statecharts variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science 863, pages 128–148. Springer, 1994.

[19] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[20] B.S. Blanchard and W.J. Fabrycky. *Systems Engineering and Analysis*. Prentice Hall, 1998.

[21] C. Bock. Unified behavior models. *Journal of Object-Oriented Programming*, 12(5), 1999.

[22] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[23] C. Bolton and J. Davies. Activity graphs and processes. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Proc. Integrated Formal Methods (IFM 2000)*, Lecture Notes in Computer Science 1945, pages 77–96. Springer, 2000.

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

[25] E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In T. Rus, editor, *Proc. International Conference on Algebraic Methodology and Software Technology (AMAST 2000)*, Lecture Notes in Computer Science 1826, pages 293–308. Springer, 2000.

[26] N.A. Brand and J.R.P. van der Kolk. *Werkstroomanalyse en -ontwerp (in Dutch)*. Kluwer BedrijfsInformatie, 1995.

[27] R. Bruni and U. Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Information and Computation*, 156(1-2):46–89, 2000.

[28] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[29] C. Bussler. Enterprise-wide workflow management. *IEEE Concurrency*, 7(3):32–43, 1999.

[30] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Transactions on Database Systems*, 24(3):405–451, 1999.

[31] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual modeling of workflows. In M.P. Papazoglou, editor, *Proc. International Object-Oriented and Entity-Relationship Modelling Conference (OOER'95)*, Lecture Notes in Computer Science 1021, pages 341–354. Springer, 1995.

[32] A. Caspers. Workflow management: Analyse, modellering en implementatie (in Dutch). Master's thesis, Vrije Universiteit Amsterdam, 1998.

[33] S. Ceri and P. Fraternali. *Designing Database Applications with Objects and Rules: The IDEA Methodology*. Addison-Wesley, 1997.

[34] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, 1998.

[35] W. Chan, R.J. Anderson, P. Beame, D.H. Jones, D. Notkin, and W.E. Warner. Optimizing symbolic model checking for statecharts. *IEEE Transactions on Software Engineering*, 27(2):170–190, 2001.

[36] E. Chang, Z. Manna, and A. Pnueli. The saftey-progress classification. In F.L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, NATO/ASI, pages 143–202. Springer, 1993.

[37] S. Christensen and N.D. Hansen. Coloured Petri nets extended with channels for synchronous communication. Technical Report PB-390, Aarhus University, 1992.

[38] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[39] E.M. Clarke and I.A. Draghicescu. Expressibility results for linear-time and branching-time logics. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. of the School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Lecture Notes in Computer Science 354, pages 428–437. Springer, 1989.

[40] E.M. Clarke, M. Fujita, S.P. Rajan, T.W. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In L. Pierre and T. Kropf, editors, *Proc. Correct Hardware Design and Verification Methods (CHARME'99)*, Lecture Notes in Computer Science 1703, pages 298–312. Springer, 1999.

[41] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[42] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

[43] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[44] C.P.M. Cleiren and J.F. Nijboer, editors. *Strafrecht: de tekst van het Wetboek van Strafrecht en enkele aanverwante wetten voorzien van commentaar (in Dutch)*. Kluwer, 2000.

[45] S. Dami, J. Estublier, and M. Amiour. Apel: A graphical yet executable formalism for process modeling. *Automated Software Engineering*, 5(1):61–96, 1998.

[46] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Proc. Compositionality: The Significant Difference (COMPOS '97)*, Lecture Notes in Computer Science 1536, pages 186–238. Springer, 1998.

[47] D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, 1996.

[48] F. Dehne, R. Wieringa, and H. van de Zandschulp. Toolkit for conceptual modeling (TCM) — user's guide and reference. Technical report, University of Twente, 2000. Available at `http://www.cs.utwente.nl/~tcm`.

[49] W. Deiters and V. Gruhn. Process management in practice applying the FUNSOFT net approach to large-scale processes. *Automated Software Engineering*, 5(1):7–25, 1998.

[50] J. Desel and T. Erwin. Modeling, simulation and analysis of business processes. In Aalst et al. [1], pages 129–141.

[51] J. Desel and G. Juhás. What is a Petri net? Informal answers for the informed reader. In H. Ehrig, G. Juhás, J. Padberg, and G. Rozenberg, editors, *Unifying Petri Nets*, Lecture Notes in Computer Science 2128, pages 1–27. Springer, 2001.

[52] E.W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[53] M. Dumas and A. ter Hofstede. UML activity diagrams as a workflow specification language. In M. Gogolla and C. Kobryn, editors, *Proc. ≪UML≫ 2001*, Lecture Notes in Computer Science 2185, pages 76–90. Springer, 2001.

[54] H. Eertink, W. Janssen, P. Oude Luttighuis, W. Teeuw, and C. Vissers. A business process design language. In J.M. Wing, J. Woodcock, and J. Davies, editors, *Proc. FM'99—Formal Methods, Volume I*, Lecture Notes in Computer Science 1708, pages 76–95. Springer, 1999.

[55] C.A. Ellis and G.J. Nutt. Modelling and enactment of workflow systems. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, Lecture Notes in Computer Science 691, pages 1–16. Springer, 1993.

[56] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamins/Cummings, 1994.

[57] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 995–1072. North-Holland, 1990.

[58] W. Emmerich and V. Gruhn. FUNSOFT nets: A Petri-net based software process modeling language. In *Proc. of the International Workshop on Software Specification and Design*, pages 175–184. IEEE Computer Society Press, 1991.

[59] H.-E. Eriksson and M. Penker. *Business Modeling With UML: Business Patterns at Work*. Wiley Computer Publishing, 2000.

[60] R. Eshuis. Model checking activity diagrams in TCM. Technical report, University of Twente, 2001. Available at `http://www.cs.utwente.nl/~tcm`.

[61] R. Eshuis, D.N. Jansen, and R. Wieringa. Requirements-level semantics and model checking of object-oriented statecharts. *Requirements Engineering Journal*, pages ??–??, 2002. To appear.

[62] R. Eshuis and R. Wieringa. Requirements-level semantics for UML statecharts. In S.F. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS) IV*, pages 121–140, 2000.

[63] R. Eshuis and R. Wieringa. A formal semantics for UML activity diagrams. Technical Report TR-CTIT-01-04, University of Twente, 2001.

[64] R. Eshuis and R. Wieringa. A real-time execution semantics for UML activity diagrams. In H. Hussmann, editor, *Proc. Fundamental Approaches to Software Engineering (FASE 2001)*, Lecture Notes in Computer Science 2029, pages 76–90. Springer, 2001.

[65] R. Eshuis and R. Wieringa. An execution algorithm for UML activity graphs. In M. Gogolla and C. Kobryn, editors, *Proc. ≪UML≫ 2001*, Lecture Notes in Computer Science 2185, pages 47–61. Springer, 2001.

[66] R. Eshuis and R. Wieringa. Verification support for workflow design with UML activity graphs. In *Proc. International Conference on Software Engineering (ICSE 2002)*, pages 166–176. ACM Press, 2002.

[67] R. Eshuis and R. Wieringa. Comparing Petri net and activity diagram variants for workflow modelling – a quest for reactive Petri nets. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication Based Systems*, Lecture Notes in Computer Science xxxx. Springer, 2002. To appear.

[68] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[69] E.S. Ferguson. *Engineering and the Mind's Eye*. MIT Press, 1993.

[70] A. Foremniak and P.H. Starke. Analyzing and reducing simultaneous firing in signal-event nets. *Fundamenta Informaticae*, 43:81–104, 2000.

[71] M. Fowler and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2nd edition, 1999.

[72] N. De Francesco, U. Montanari, and G. Ristori. Modelling concurrent accesses to shared data via Petri nets. In U. Montanari and E.R. Olderog, editors, *Proc. Conference on Programming Concepts, Methods and Calculi (PROCOMET '94)*, pages 489–508. North-Holland, 1994.

[73] T. Gehrke, U. Goltz, and H. Wehrheim. The dynamic models of UML: Towards a semantics and its application in the development process. Hildesheimer Informatik-Bericht 11/98, Institut für Informatik, Universität Hildesheim, 1998.

[74] H.J. Genrich. Predicate/transition nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, Lecture Notes in Computer Science 254, pages 207–247. Springer, 1987.

[75] H.J. Genrich and P.S. Thiagarajan. A theory of bipolar synchronization schemes. *Theoretical Computer Science*, 30(3):241–318, 1984.

[76] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modelling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[77] A. Geppert, D. Tombros, and K.R. Dittrich. Defining the semantics of reactive components in event-driven workflow execution with event histories. *Information Systems*, 23(3-4):235–252, 1998.

[78] R. Goldblatt. *Logics of Time and Computation*. CSLI Lecture notes 7, Stanford University, 2nd edition, 1992.

[79] P. Grefen. Transactional workflows or workflow transactions? In R. Cicchetti, A. Hameurlain, and R. Traunmüller, editors, *Proc. 13th International Conference on Database and Expert Systems Applications (DEXA 2002)*, pages 60–69. Springer, 2002.

[80] P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig. CrossFlow: Cross-organizational workflow management in dynamic virtual enterprises. *International Journal of Computer Systems Science and Engineering*, 15(5):277–290, 2000.

[81] P. Grefen and R. Remmerts de Vries. A reference architecture for workflow management systems. *Journal of Data & Knowledge Engineering*, 27(1):31–57, 1998.

[82] P. Grefen, B. Pernici, and G. Sánchez. *Database Support for Workflow Management: the WIDE Project*. Kluwer Academic Publishers, 1999.

[83] P. Grefen, J. Vonk, and P. Apers. Global transaction support for workflow management systems: from formal specification to practical implementation. *The VLDB Journal*, 10(4):316–333, 2001.

[84] J.C. Grundy and J.G. Hosking. Serendipity: Integrated environment support for process modelling, enactment and work coordination. *Automated Software Engineering*, 5(1):27–60, 1998.

[85] C. Hagen and G. Alonso. Beyond the black box: Event-based inter-process communication in process support systems. In *IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 450–457. IEEE Computer Society Press, 1999.

[86] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[87] H.-M. Hanisch and A. Lüder. A signal extension for Petri nets and its use in controller design. *Fundamenta Informaticae*, 41(4):415–431, 2000.

[88] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[89] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[90] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. on Software Engineering and Methodology*, 5(4):293–333, 1996.

[91] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO/ASI*, pages 447–498. Springer, 1985.

[92] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: the STATEMATE approach*. McGraw-Hill, 1998.

[93] J. Hatcliff, M.B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[94] F.H. Heida, R. Jobse, and M. Deterink. Model beslag (in Dutch), 1999. Internal report Dutch Public Prosecution Service.

[95] M.P.E. Heimdahl and M.W. Whalen. Reduction and slicing of hierarchical state machines. In M. Jazayeri and H. Schauer, editors, *Proc. of the Sixth European Software Engineering Conference (ESEC/FSE '97)*, Lecture Notes in Computer Science 1013, pages 450–467. Springer, 1997.

[96] T.A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In W. Kuich, editor, *Proc. International Colloquium on Automata, Languages, and Programming (ICALP'92)*, Lecture Notes in Computer Science 623, pages 545–558. Springer, 1992.

[97] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[98] A.H.M. ter Hofstede and M.E. Orlowska. On the complexity of some verification problems in process control specifications. *The Computer Journal*, 42(5):349–359, 1999.

[99] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[100] IBM. MQ Series Workflow (Websphere). `http://www.ibm.com`.

[101] J. Helbig and P. Kelb. An OBDD-representation of statecharts. In *Proc. of the European Design and Test Conference*, pages 142–151. IEEE Computer Society Press, 1994.

[102] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, 1996.

[103] M. Jackson and G. Twaddle. *Business Process Implementation: Building Workflow Systems*. Addison-Wesley, 1997.

[104] W. Janssen, R. Mateescu, S. Mauw, and J. Springintveld. Verifying business processes using Spin. In E. Najm, A. Serhrouchni, and G. Holzmann, editors, *Proc. International Spin workshop*, 1998. Available at `http://netlib.bell-labs.com/netlib/spin/ws98/sjouke.ps.gz`.

[105] K. Jensen. *Coloured Petri Nets. Basic concepts, analysis methods and practical use*. EATCS monographs on Theoretical Computer Science. Springer, 1992.

[106] S. Joosten. Trigger modelling for workflow analysis. In G. Chroust and A. Benczur, editors, *Proc. CON'94: Workflow Management, Challenges, Paradigms and Products*, pages 236–247, 1994.

[107] C. Karamanolis, D. Giannakopoulou, J. Magee, and S. Wheater. Modelling and analysis of workflow processes. Technical Report 99/2, Department of Computing, Imperial College, 1999.

[108] R.M. Karp and R.E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

[109] Y. Kesten, Z. Manna, and A. Pnueli. Verification of clocked and hybrid systems. *Acta Informatica*, 36(11):837–912, 2000.

[110] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In K.G. Larsen, S. Skyum, and G. Winskel, editors, *Proc. International Colloquium on Automata, Languages and Programming (ICALP'98)*, Lecture Notes in Computer Science 1443, pages 1–16. Springer, 1998.

[111] A. Kleppe and J. Warmer. Unification of static and dynamic semantics of UML—a study in redefining the semantics of the UML using the pUML OO meta modelling approach. Technical report, Klasse Objecten, 2000. Available at `http://www.klasse.nl`.

[112] O. Kupferman and A. Pnueli. Once and for all. In *Proc. IEEE Symposium on Logic in Computer Science (LICS '95)*, pages 25–35. IEEE Computer Society Press, 1995.

[113] L. Lamport. What good is temporal logic? In R.E.A. Mason, editor, *Proc. of the IFIP Congress on Information Processing*, pages 657–667. North-Holland, 1983.

[114] L. Lamport. How to write a long formula (short communication). *Formal Aspects of Computing*, 6(5):580–584, 1994.

[115] L. Lamport. Proving possibility properties. *Theoretical Computer Science*, 206(1–2):341–352, 1998.

[116] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[117] N.L. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.

[118] F. Leymann and D. Roller. *Production Workflow – Concepts and Techniques.* Prentice Hall, 2000.

[119] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In R. Parikh, editor, *Proc. 3rd Workshop on Logics of Programs*, Lecture Notes in Computer Science 193, pages 196–218. Springer, 1985.

[120] J. Lilius and I. Porres Paltor. vUML: A tool for verifying UML models. In *Proc. IEEE International Conference on Automated Software Engineering*, pages 255–258. IEEE Computer Society Press, 1999.

[121] M. Löwe, D. Wikarski, and Y. Han. Higher-order object nets and their application to workflow modeling. Technical Report 95-34, Informatik, Technical University Berlin, 1995.

[122] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer, 1992.

[123] Z. Manna and A. Pnueli. Clocked transition systems. In A. Pnueli and H. Lin, editors, *Logic and Software Engineering*, pages 3–42. World Scientific, 1996.

[124] S. McMenamin and J. Palmer. *Essential Systems Analysis*. Yourdon Press, 1984.

[125] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[126] E. Mikk. *Semantics and Verification of Statecharts*. PhD thesis, University of Kiel, 2000.

[127] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[128] U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 32(6):545–596, 1995.

[129] T. Murata. Petri nets: Properties, analysis, and applications. *Proc. of the IEEE*, 77(4):541–580, 1989.

[130] P. Muth, D. Wodtke, J. Weissenfels, G. Weikum, and A. Kotz-Dittrich. Enterprise-wide workflow management based on state and activity charts. In A. Dogac, L. Kalinichenko, T. Özsu, and A. Sheth, editors, *Workflow Management Systems and Interoperability*, NATO/ASI. Springer, 1998.

[131] M. Nüttgens, T. Feld, and V. Zimmermann. Business process modeling with EPC and UML: Transformation or integration? In M. Schader and A. Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications*, pages 250–261. Physica-Verlag, 1998.

[132] A. Oberweis, R. Schätzle, W. Stucky, W. Weitz, and G. Zimmermann. INCOME/WF: A Petri net based approach to workflow management. In H. Krallmann, editor, *Wirtschaftsinformatik '97*, pages 557–580. Springer, 1997.

[133] Object Management Group. Workflow management facility specification, 2000. OMG Document Number formal/00-05-02. Available at `http://www.omg.org`.

[134] E.-R. Olderog. *Nets, Terms and Formulas*. Cambridge Tracts in Theoretical Computer Science 23. Cambridge University Press, 1991.

[135] M.A. Ould. *Business Processes: Modelling and Analysis for Re-engineering and Improvement*. Wiley, 1995.

[136] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[137] P. Pinheiro da Silva. A proposal for a LOTOS-based semantics for UML. Technical Report UMCS-01-06-1, Department of Computer Science, University of Manchester, 2001.

[138] A. Pleyer and W. Jekeli. SAP response against the UML 2.0 RFI, 1999. Object Management Group document ad/99-12-26.

[139] A. Pnueli and E. Shahar. A platform combining deductive with algorithmic verification. In R. Alur and T.A. Henzinger, editors, *Proc. International Conference on Computer Aided Verification (CAV '96)*, Lecture Notes in Computer Science 1102, pages 184–195. Springer, 1996.

[140] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science 526, pages 244–265. Springer, 1991.

[141] W. Reisig. *Petri Nets: An Introduction*. Number 4 in EATCS Monographs on Theoretical Computer Science. Springer, 1985.

[142] W. Reisig and G. Rozenberg, editors. *Lectures on Petri nets I: Advances in Petri nets*, Lecture Notes in Computer Science 1491. Springer, 1998.

[143] W. Sadiq and M.E. Orlowska. Analyzing process models using graph reduction techniques. *Information Systems*, 25(2):117–134, 2000.

[144] K. Salimifard and M. Wright. Petri net-based modelling of workflow systems: An overview. *European Journal of Operational Research*, 134(3):218–230, 2001.

[145] A.-W. Scheer. *ARIS – Business Process Modeling*. Springer, Berlin, 2nd edition, 1999.

[146] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object Oriented Modeling*. John Wiley & Sons, 1994.

[147] Software Ley. Cosa. `http://www.cosa.de`.

[148] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.

[149] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[150] UML Revision Taskforce. *OMG UML Specification v. 1.4*. Object Management Group, 2001. OMG Document Number formal/01-09-67. Available at `http://www.omg.org`.

[151] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[152] J. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, 2001.

[153] UN/CEFACT and OASIS. ebXML. `http://www.ebxml.org`.

[154] M.Y. Vardi. Branching vs. linear time: Final showdown. In T. Margaria and W. Yi, editors, *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, Lecture Notes in Computer Science 2031, pages 1–22. Springer, 2001.

[155] H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing workflow processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.

[156] J. Widom and S. Ceri, editors. *Active Database Systems – Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, 1996.

[157] R.J. Wieringa. *Design Methods for Software Systems: Yourdon, Statemate and the UML*. Morgan Kaufmann, 2002. To be published.

[158] D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In F.N. Afrati and P. Kolaitis, editors, *Proc. International Conference on Database Theory (ICDT '97)*, Lecture Notes in Computer Science 1186, pages 230–246. Springer, 1997.

[159] Workflow Management Coalition. The workflow reference model, 1995. WFMC document WFMC-TC-1003. Available at `http://www.wfmc.org`.

[160] Workflow Management Coalition. Workflow management coalition interface 1: Process definition interchange process model, 1999. WFMC document WFMC-TC-1016-P. Available at `http://www.wfmc.org`.

[161] Workflow Management Coalition. Workflow management coalition specification — terminology & glossary, 1999. WFMC document WFMC-TC-1011. Available at `http://www.wfmc.org`.

[162] E. Yourdon. *Modern Structured Analysis*. Prentice Hall, 1989.

[163] S. Yovine. KRONOS: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1(1/2):123–133, 1997.

# Index

# Abstract

This thesis defines a formal semantics for UML activity diagrams that is suitable for workflow modelling. The semantics allows verification of functional requirements using model checking. Since a workflow specification prescribes how a workflow system behaves, the semantics is defined and motivated in terms of workflow systems. As workflow systems are reactive and coordinate activities, the defined semantics reflects these aspects. In fact, two formal semantics are defined, which are completely different. Both semantics are defined directly in terms of activity diagrams and not by a mapping of activity diagrams to some existing formal notation. The requirements-level semantics, based on the STATEMATE semantics of statecharts, assumes that workflow systems are infinitely fast w.r.t. their environment and react immediately to input events (this assumption is called the perfect synchrony hypothesis). The implementation-level semantics, based on the UML semantics of statecharts, does not make this assumption. Due to the perfect synchrony hypothesis, the requirements-level semantics is unrealistic, but easy to use for verification. On the other hand, the implementation-level semantics is realistic, but difficult to use for verification. A class of activity diagrams and a class of functional requirements is identified for which the outcome of the verification does not depend upon the particular semantics being used, i.e., both semantics give the same result. For such activity diagrams and such functional requirements, the requirements-level semantics is as realistic as the implementation-level semantics, even though the requirements-level semantics makes the perfect synchrony hypothesis. The requirements-level semantics has been implemented in a verification tool. The tool interfaces with a model checker by translating an activity diagram into an input for a model checker according to the requirements-level semantics. The model checker checks the desired functional requirement against the input model. If the model checker returns a counterexample, the tool translates this counterexample back into the activity diagram by highlighting a path corresponding to the counterexample. The tool supports verification of workflow models that have event-driven behaviour, data, real time, and loops. Only model checkers supporting strong fairness model checking turn out to be useful. The feasibility of the approach is demonstrated by using the tool to verify some real-life workflow models.

# Samenvatting

In dit proefschrift wordt een formele semantiek voor UML activiteitendiagrammen gedefinieerd die geschikt is voor workflowmodellering. De semantiek maakt verificatie van functionele eigenschappen met behulp van model checking mogelijk. Aangezien een workflowspecificatie voorschrijft hoe een workflowsysteem zich gedraagt, wordt de semantiek gedefinieerd en gemotiveerd in termen van workflowsystemen. Omdat workflowsystemen reactief zijn en activiteiten coördineren, weerspiegelt de gedefinieerde semantiek deze aspecten. Eigenlijk worden er twee semantieken gedefinieerd, die totaal verschillend zijn. Beide semantieken worden direct gedefinieerd in termen van activiteitendiagrammen en niet door een activiteitendiagram te vertalen naar een bestaande formele taal. De requirements-level semantiek neemt aan dat workflowsystemen oneindig snel zijn ten opzichte van hun omgeving en dat ze direct reageren op gebeurtenissen (deze aanname wordt de 'perfect synchrony' aanname genoemd). De implementation-level semantiek maakt deze aanname niet. De requirements-level semantiek is gebaseerd op de STATEMATE semantiek van statecharts. De implementation-level semantiek is gebaseerd op de UML semantiek van statecharts. Door de perfect synchrony aanname is de requirements-level semantiek onrealistisch, maar makkelijk te gebruiken voor verificatie. Daarentegen is de implementation-level semantiek realistisch, maar moeilijk te gebruiken voor verificatie. Er wordt een klasse van activiteitendiagrammen en een klasse van functionele eisen gedefinieerd waarvoor geldt dat de uitkomst van de verificatie niet afhangt van de gebruikte semantiek, dat wil zeggen, beide semantieken geven hetzelfde resultaat. Voor zulke activiteitendiagrammen en zulke eisen is de requirements-level semantiek even realistisch als de implementation-level semantiek, hoewel de requirements-level semantiek de perfect synchrony aanname maakt. De requirements-level semantiek is geïmplementeerd in een softwaregereedschap voor verificatie. Het softwaregereedschap maakt als volgt gebruik van een model checker. Het softwaregereedschap vertaalt een activiteitendiagram naar een invoer voor een model checker volgens de requirements-level semantiek. De model checker controleert of het invoermodel voldoet aan de gewenste eigenschap. Als de model checker een tegenvoorbeeld teruggeeft, vertaalt het softwaregereedschap dit tegenvoorbeeld terug in termen van het activiteitendiagram door een pad dat correspondeert met het tegenvoor-

beeld te laten oplichten. Het softwaregereedschap ondersteunt verificatie van workflowmodellen die gebeurtenis-gedreven gedrag, data, real-time en iteraties beschrijven. Alleen model checkers die strong fairness model checking ondersteunen blijken nuttig te zijn. De haalbaarheid van de aanpak wordt aangetoond door het softwaregereedschap te gebruiken om enkele workflowmodellen die op de praktijk gebaseerd zijn, te verifiëren.

# CTIT Ph.D.-Thesis Series

**ISSN: 1381-3617**

| | |
|---|---|
| 02-41 | H.E. Blok, *Database Optimization Aspects for Information Retrieval* |
| 02-40 | R. van Zwol, *Modelling and searching web-based document collections* |
| 01-39 | G. Hiddink, *Educational Multimedia Databases* |
| 01-38 | C.R. Guareis de Farias, *Architectural Design of Groupware Systems: a Component-Based Approach* |
| 01-37 | H.T.J.A. Uitermark, *Ontology-Based Geographic Data Set Integration* |
| 01-36 | I. Vatcheva, *Computer-Supported Experiment Selection for Model Discriminiation* |
| 01-35 | J.T. van der Veen, *Telematic Support for Group-Based Learning* |
| 01-34 | T.C. Ruys, *Towards Effective Model Checking* |
| 01-33 | C. de Barros Barbosa, *Frameworks for Protocol Implementation: A Model Based Approach* |
| 01-32 | D. Hiemstra, *Using language Models for Information Retrieval* |
| 00-31 | M.J.J. Garvels, *The Splitting Method in Rare Event Simulation* |
| 00-30 (00-01) | P.T. de Boer, *Analysis and Efficient Simulation of Queueing models of Telecommunication Systems* |
| 00-29 | D. Remondo Bueno, *Performance Evaluation of Communication Systems via Importance Sampling* |
| 99-28 | D.T. van Veen, *All-Optical Multiwavelength Ring Networks* |
| 99-27 | E. Oltmans, *A Knowledge-based Approach to Robust Parsing* |
| 99-26 | A.P. de Vries, *Content and Multimedia Database Management Systems* |
| 99-25 | P.R. D'Argenio, *Algebras and Automata for Timed and Stochastic Systems* |
| 99-24 | D. Spelt, *Verification support for object database design* |
| 98-23 | H. de Jong, *Computer-supported analysis of scientific measurements* |
| 98-22 | M. Rijnders, *Optical Signal Processing – a novel approach to modifying digital information in the optical domain* |
| 98-21 | L. Heerink, *Ins and Outs in Refusal Testing* |
| 97-20 | J. Skowronek, *Distributed Optimization of Nested Queries* |
| 97-19 | F. Tempelman, *Structured Representations in case-based reasoning* |
| 98-18 | D.A.C. Quartel, *Action relations: Basic design concepts for behaviour modelling and refinement* |
| 98-17 | M. de Weger, *Structuring of Business Processes – an architectural approach to distributed systems development and its application to business processes* |
| 97-16 | M. van Keulen, *Formal Operation Definition in Object-oriented Databases* |
| 97-15 | H.M. Veenhof, *Processing Multi-Way Spatial Joins* |
| 97-15 | A. Olah, *Design and Analysis of Transport Protocols for Reliable High-Speed Communications* |
| 97-14 | W.N. Borst, *Construction of Engineering Ontologies for Knowledge Sharing and Reuse* |
| 97-13 | P. Kars, *Process-algebraic Transformation in Context* |
| 97-12 | P. Leydekkers, *Multi-media Services in Open Distributed Telecommunications Environments* |
| 97-11 | M. Vermeer, *Semantic Interoperability for Legacy Databases* |
| 96-10 | Franken, L.J.N., *Quality of Service Management: a Model-Based Approach* |
| 96-09 | Katoen, J.P., *Quantitative and Qualitative Extensions of Event Structures* |

# SIKS Dissertatiereeks

1998-1 Johan van den Akker (CWI), *DEGAS – An Active, Temporal Database of Autonomous Objects*

1998-2 Floris Wiesman (UM), *Information Retrieval by Graphically Browsing Meta-Information*

1998-3 Ans Steuten (TUD), *A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective*

1998-4 Dennis Breuker (UM), *Memory versus Search in Games*

1998-5 E. W. Oskamp (RUL), *Computerondersteuning bij Straftoemeting*

1999-1 Mark Sloof (VU), *Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products*

1999-2 Rob Potharst (EUR), *Classification using decision trees and neural nets*

1999-3 Don Beal (UM), *The Nature of Minimax Search*

1999-4 Jacques Penders (UM), *The practical Art of Moving Physical Objects*

1999-5 Aldo de Moor (KUB), *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems*

1999-6 Niek J. E. Wijngaards (VU), *Re-design of compositional systems*

1999-7 David Spelt (UT), *Verification support for object database design*

1999-8 Jacques H. J. Lenting (UM), *Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation*

2000-1 Frank Niessink (VU), *Perspectives on Improving Software Maintenance*

2000-2 Koen Holtman (TUE), *Prototyping of CMS Storage Management*

2000-3 Carolien M. T. Metselaar (UvA), *Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief*

2000-4 Geert de Haan (VU), *ETAG, A Formal Model of Competence Knowledge for User Interface Design*

2000-5 Ruud van der Pol (UM), *Knowledge-based Query Formulation in Information Retrieval*

2000-6 Rogier van Eijk (UU), *Programming Languages for Agent Communication*

2000-7 Niels Peek (UU), *Decision-theoretic Planning of Clinical Patient Management*

2000-8 Veerle Coupé (EUR), *Sensitivity Analyis of Decision-Theoretic Networks*

2000-9 Florian Waas (CWI), *Principles of Probabilistic Query Optimization*

2000-10 Niels Nes (CWI), *Image Database Management System Design Considerations, Algorithms and Architecture*

2000-11 Jonas Karlsson (CWI), *Scalable Distributed Data Structures for Database Management*

2001-1 Silja Renooij (UU), *Qualitative Approaches to Quantifying Probabilistic Networks*

2001-2 Koen Hindriks (UU), *Agent Programming Languages: Programming with Mental Models*

2001-3 Maarten van Someren (UvA), *Learning as problem solving*

2001-4 Evgueni Smirnov (UM), *Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets*

2001-5 Jacco van Ossenbruggen (VU), *Processing Structured Hypermedia: A Matter of Style*

2001-6 Martijn van Welie (VU), *Task-based User Interface Design*

2001-7 Bastiaan Schonhage (VU), *Diva: Architectural Perspectives on Information Visualization*

2001-8 Pascal van Eck (VU), *A Compositional Semantic Structure for Multi-Agent Systems Dynamics*

2001-9 Pieter Jan 't Hoen (RUL), *Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes*

2001-10 Maarten Sierhuis (UvA), *Modeling and Simulating Work Practice – BRAHMS: a multiagent modeling and simulation language for work practice analysis and design*

2001-11 Tom M. van Engers (VUA), *Knowledge Management: The Role of Mental Models in Business Systems Design*

2002-01 Nico Lassing (VU), *Architecture-Level Modifiability Analysis*

2002-02 Roelof van Zwol (UT), *Modelling and searching web-based document collections*

2002-03 Henk Ernst Blok (UT), *Database Optimization Aspects for Information Retrieval*

2002-04 Juan Roberto Castelo Valdueza (UU), *The Discrete Acyclic Digraph Markov Model in Data Mining*

2002-05 Radu Serban (VU), *The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents*

2002-06 Laurens Mommers (UL), *Applied legal epistemology; Building a knowledge-based ontology of the legal domain*

2002-07 Peter Boncz (CWI), *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*

2002-08 Jaap Gordijn (VU), *Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas*

2002-09 Willem-Jan van den Heuvel (KUB), *Integrating Modern Business Applications with Objectified Legacy Systems*

2002-10 Brian Sheppard (UM), *Towards Perfect Play of Scrabble*

2002-11 Wouter C. A. Wijngaards (VU), *Agent Based Modelling of Dynamics: Biological and Organisational Applications*

2002-12 Albrecht Schmidt (UvA), *Processing XML in Database Systems*

2002-13 Hongjing Wu (TUE), *A Reference Architecture for Adaptive Hypermedia Applications*

2002-14 Wieke de Vries (UU), *Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems*