Dissertation zur Erlangung
des Grades des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Applying the *Engineering Statechart Formalism* to the Evaluation of Soft Real-Time in Operating Systems

-

A Use Case Tailored Modeling and Analysis Technique

Vorgelegt von
Alexander Koenen-Dresp
Glons, Belgien

Tag der Einreichung: 4. Juli 2008

Tag des Kolloquiums:	28. Oktober 2008
Dekan:	Prof. Dr. Joachim Weickert
Vorsitzender der Prüfungskommission:	Prof. Dr. Reinhard Wilhelm
1. Berichterstatter:	Prof. Dr. Helge Scheidig
2. Berichterstatter:	Prof. Bernd Finkbeiner, PhD
Akademischer Mitarbeiter:	Dr. Mark Hillebrand

Acknowledgements

First of all I thank my primary advisor Professor Helge Scheidig for the unique opportunity to participate in his research. I am also very grateful for the excellent supervision and the guidance Professor Bernd Finkbeiner gave me during my research. Without his clear guidelines on the methodology, this thesis would never have been possible. My colleges Sebastian Schöning and Reinhart Spurk also greatly contributed to this work - thank you.

Words cannot express my gratitude towards my wife Wiebke. Her never-ending fight for my success was the best support for which I could ever have hoped.

I am very grateful to my three proof readers, Calogero Cumbo, Christian Franze and Robert Koch. To Robert I owe thanks for his friendship and personal support over the last decade.

Since my research and the writing of this thesis was conducted while I was a full-time employee of the German Armed Forces Command Control and Information System Regiment (GAFCCIS Rgt) in Cologne and later of the NATO Programming Centre (NPC) in Glons (Belgium), I would like to thank all my superior officers for their support. In particular, Colonel Scholz, Commander of the GAFCCIS Rgt and his deputy Lieutenant Colonel Domke. At the NPC my thanks go to Colonel Eisenreich, Commander of the NPC, Lieutenant Colonels Nauth and Feuerbach, the Senior National Officers of Germany and to all my colleges for taking over my duties during my absent periods.

Finally I would like to express my sincere appreciation to my family and all those not mentioned who have contributed, directly or indirectly to this thesis.

For my beloved wife Wiebke and my dearly departed grandfather Paul

All we have to decide is what to do with the time that is given to us.
Gandalf the Grey, J.R.R. Tolkien

Abstract

Multimedia applications that have emerged in recent years impose unique requirements on an underlying general purpose operating system (GPOS). The suitability of a GPOS for multimedia processing is judged by its soft real-time capabilities. To date, the question of how these capabilities can be assessed has scarcely been addressed: this is a gap in GPOS research.

By answering questions on the impacts of the Interrupt Handling Facility (IHF) on the overall soft real-time capabilities of a GPOS, this thesis contributes to the filling of this blank space. The *Engineering Statechart Formalism* (*ESF*), a use case tailored formal method of modeling real-world OS, is syntactically and semantically defined. Models of the IHF of selected real-world operating systems are then created by means of this technique.

As no appropriate real-time concept fitting the goals of this thesis as yet exists, a suitable definition is constructed. By projecting this system-wide idea to the interrupt subsystem, specific indicators for this subsystem are derived. These indicators are then evaluated by applying formal techniques such as graph-based analysis and temporal logic model checking to the *ESF* models. Finally, the assertions derived from this evaluation are interpreted with respect to their impacts on real-time multimedia processing in different general purpose operating systems.

Kurzzusammenfassung

Multimedia-Anwendungen haben in den letzten Jahren weite Verbreitung erfahren. Solche Anwendungen stellen besondere Anforderungen an das Betriebssystem (BS), auf dem sie ausgeführt werden. Insbesondere Echtzeitfähigkeiten des Betriebssystems sind von Bedeutung, wenn es um seine Eignung für Multimedia-Verarbeitung geht. Bis heute wurde die Frage, wie sich diese Fähigkeiten konkret innerhalb eines BS manifestieren, nur unzureichend untersucht.
Die vorliegende Arbeit leistet einen Beitrag zur Füllung dieser Lücke in der BS-Forschung. Die Effekte des Subsystems zur Unterbrechungsbehandlung in BS auf die Echtzeitfähigkeit des Gesamtsystems werden detailliert auf Basis von Modellen dieses Subsystems in verschiedenen BS analysiert. Um eine formale Auswertung zu erlauben, wird eine auf den Anwendungsfall zugeschnittene formale Methode zur BS-Modellierung verwendet. Die spezifizierte Syntax und Semantik dieses *Engineering Statechart Formalism (ESF)* basieren auf dem klassischen Statechart-Formalismus.
Da bislang kein geeigneter Echtzeit-Begriff existiert, wird eine konsistente Definition hergeleitet. Durch die Abbildung dieser sich auf das Gesamtsystem beziehenden Eigenschaft auf die Unterbrechungsbehandlung werden spezifische Indikatoren für dieses Subsystem hergeleitet. Die Ausprägungen dieser Indikatoren für die verschiedenen untersuchten Betriebssyteme werden anhand formaler Methoden wie graphbasierter Analyse und Temporal Logic Model Checking ausgewertet. Die Interpretation der Untersuchungsergebnisse liefert Aussagen über die Effekte der Implementierung der Unterbrechungsbehandlung auf die Echtzeitfähigkeit der untersuchten Betriebssysteme bei der Verarbeitung von multimedialen Daten.

Ausführliche Zusammenfassung

Multimedia-Anwendungen haben in den letzten Jahren weite Verbreitung erfahren. Solche Anwendungen stellen besondere Anforderungen an das Betriebssystem (BS), auf dem sie ausgeführt werden. Insbesondere Echtzeitfähigkeiten des Betriebssystems sind von Bedeutung, wenn es um seine Eignung für Multimedia-Verarbeitung geht. Es existieren zwar zahlreiche Lösungen für die Verbesserung des systeminternen Scheduling, aber ganzheitliche Ansätze, die andere Komponenten des Systems mit einbeziehen, sind bisher kaum vorhanden. Die damit einhergehende Frage, wie sich die geforderten Echtzeitfähigkeiten in konkreten Subsystemen innerhalb eines BS manifestieren, ist daher bis heute nur unzureichend untersucht.

Die vorliegende Arbeit leistet einen Beitrag zur Füllung dieser Lücke in der Betriebssystem-Forschung. Der interaktive Teil eines Betriebssystems, die Unterbrechungsbehandlung, wird hier eingehend hinsichtlich seiner Einflüsse auf das Systemverhalten hin untersucht. Dazu kommen Methoden aus den Disziplinen des Software Engineering und der rechnergestützten Verifikation zum Einsatz: die Effekte des Subsystems zur Unterbrechungsbehandlung in verschiedenen BS auf die Echtzeitfähigkeit des Gesamtsystems werden detailliert mittels formaler Modelle abgebildet und analysiert.

Da sich Betriebssysteme bezüglich ihrer inhärenten Eigenschaften signifikant von anderen Software-Systemen unterscheiden, ist der erste notwendige Schritt die Entwicklung einer formalen, anwendungsfallspezifischen Modellierungstechnik. Ein auf klassischen Statecharts basierender Formalismus – der *Engineering Statechart Formalism (ESF)* – wird hergeleitet und seine Syntax sowie Semantik definiert. Die zusätzlichen Statechart-Elemente erleichtern die Modellierung von Rekursion, reduzieren die Komplexität der Modelle und stellen ein Werkzeug zur Verfügung, um sowohl logisch gruppierte als auch parallele Abläufe ohne Mehraufwand abzubilden.

Drei unterschiedliche Betriebssysteme für die Intel IA32-Architektur, namentlich Linux, OpenBSD und L4Ka::Pistachio werden ausgewählt und die relevanten Bestandteile des Interrupt-Subsystems mittels Techniken des Reverse Engineering unter Verwendung des *ESF* modelliert.

Da bislang kein für die Zielsetzung dieser Arbeit geeigneter Echtzeit-Begriff existiert, wird eine konsistente Definition basierend auf *Time Utility Functions (TUF)* hergeleitet. Die somit erreichte Abgrenzung zwischen hartem und weichem Echtzeitbegriff erlaubt die Festlegung des Konzeptes von Multimedia-orientierter weicher Echtzeit für Betriebssysteme.

Dieser sich auf das Gesamtsystem beziehende Echtzeitbegriff wird anschließend auf die Unterbrechungsbehandlung projiziert. Da dieser Projektionsvorgang eine komplette Systemarchitektur voraussetzt, wird die am Lehrstuhl für Betriebssysteme der Universität des Saarlandes entwickelte *Component Extension (CE)* als Referenz-Architektur angenommen. Mittels der Projektion werden spezifische Indikatoren für das Interrupt-Subsystem hergeleitet. Diese Indikatoren werden ISO-Qualitätsfaktoren für Anwendungssoftware zugeordnet.

Die Ausprägungen dieser Indikatoren für die drei verschiedenen untersuchten Betriebssyteme werden anhand unterschiedlicher formaler Methoden ausgewertet. Dabei werden in Abhängigkeit von der Natur jedes Indikators verschiedene Herangehensweisen spezifiziert und angewandt. Architekturelle Eigenschaften werden mittels statischer, syntaktischer Analyseverfahren ausgewertet, für Kontrollflussbasierte Indikatoren wird eine Methode zur graphbasierten Analyse entwickelt. Die verbleibenden, eingabeabhängigen Indikatoren werden dann mittels Temporal Logic Model Checking evaluiert. Dafür wird ein umfassendes Regelwerk zur Umwandlung von *ESF*-Modellen in Kripkestrukturen definiert.

Die Interpretation der Untersuchungsergebnisse liefert quantitative und qualitative Aussagen über die Effekte der Implementierung der Unterbrechungsbehandlung auf die Echtzeitfähigkeit der untersuchten Betriebssysteme bei der Verarbeitung von multimedialen Daten. Aus diesen Ergebnissen wird eine Empfehlung für das Betriebssystem gegeben, welches optimal als Basis für das multimediale Gesamtsystem geeignet ist. Weiterhin induzieren die Analyseergebnisse Implementierungsempfehlungen für die CE.
Ein ausführlicher Ausblick auf durch die geleistete Forschungsarbeit neu erschlossene Themenbereiche rundet die Arbeit ab.

Contents

Chapter 1

Introduction

1.1 Motivation

In recent years, the focus of operating systems (OS) research has shifted to multimedia applications. These dramatically change the requirements to be met by general purpose OS (GPOS). Nowadays, a variety of solutions exist for speeding up the processing of multimedia streams. Special middleware (see e.g. [KS05]) enables those applications to run efficiently and fast.
Scheduling paradigms for multimedia purposes are subject to intensive research and development. Numerous dedicated multimedia schedulers have already been created and tested [NL97]. So far, research has focused on scheduling. Thus, there is hardly any commonly accepted understanding about the overall real-time capabilities a system suitable for multimedia processing must possess. Moreover, the question of how those capabilities manifest has scarcely been addressed.
This thesis contributes to the filling of this blank space in operating systems research. The focus will be on the investigation of the interactive component of an OS, namely the interrupt handling facility (IHF). The handling of indeterministic interaction with the environment that is conducted in this subsystem of an OS is a crucial part of any real-time consideration.
Furthermore, the application and development of formal modeling and analysis methods tailored to this specific use case lays the foundation for future research in this area.

1.2 Research Questions and Scientific Goals

Current research of the chair of operating systems at the Universität des Saarlandes focuses on graph-oriented processing of multimedia streams. This includes the specification of a system architecture that incorporates this paradigm on a GPOS platform. Within this context, questions about soft real-time capabilities of general purpose operating systems arise. A suitable definition of real-time as such needs to be constructed as a basis for deriving any assertions.

So far, existing formal methods and techniques of recent software development and system design have only been applied to the specification and analysis of minimal as well as embedded and hard real-time systems while not at all to GPOS that are by far more com-

plex. A formal modeling method will be extended on a use case tailored basis to alleviate the effort of creating an efficient and simple, while still formally precise, abstraction of GPOS. Using this methodology, models of interrupt processing in chosen GPOS of various architectural types will be created.

As the multi-faceted questions about the systems real-time capabilities cannot be simply answered by evaluating the IHF models, a set of quality factors and corresponding indicators for the devised real-time definition must be established. Along with these indicators, different formal analysis techniques are to be introduced. Applying the analysis techniques to the concrete system models allows conclusions about the applicability of a system for multimedia-oriented stream processing to be derived.

1.3 Thesis Outline

Modeling the behavior of an IHF in a non-stochastic way is a novel approach to the problem of analyzing real-time capabilities. In order to identify the most adequate formalism, an examination and evaluation of common formalisms of systems design is conducted in Chapter 2. The usability of graph-based formalisms, transition nets, queuing theories and algebras as well as description techniques or languages is assessed. Within the scope of this thesis, a formalism is considered usable if it is suitable for managing large, hierarchical models with a significant amount of concurrency and recursion while still providing a formally defined semantics. The statechart formalism turns out to be a suitable basis for the desired modeling method.
Although the chosen formalism has all the necessary basics for serving as a vehicle for operating system modeling, a number of extensions become necessary to keep the models small but select, to use statecharts more efficiently and to achieve a close approximation of the resulting models to reality. The pursued paradigm is on the one hand to shift certain commonly used patterns such as synchronization, parallelism or recursion from being explicitly modeled towards being implicitly given. On the other hand, system-inherent principles such as code locality must be exploited to gain simplicity. Therefore, a formal extension to statecharts – called the engineering statechart formalism (*ESF*) – is established in Chapter 3.
Equipped with the *ESF* as a use case tailored formal method, Chapter 4 provides detailed models of the IHF of three real-world operating systems. Since the way of interrupt processing implemented within the micro-architecture of the hardware platform is the crucial criterion for selecting a certain platform, the Intel Architecture is chosen for all models. For further modeling, a pool of suitable operating systems is derived. The systems are chosen by investigating and comparing their inherent properties such as architecture or implementation.
To create a correct model of an OS subsystem, a general modeling approach is given which makes use of the black and white box paradigm. Thus, subsystems that are not of particular interest can be blanked out. Chapter 4 also covers all aspects of the modeling process itself, starting from the technical issues when performing reverse engineering of an operating system up to a guideline of how to model implementation patterns common to operating system kernels. Finally, the models of the interrupt handling facilities of Linux,

OpenBSD and Pistachio are presented in detail.
Since it turns out to be surprisingly hard to find a fitting and consistent definition of real-time that covers hard real-time as well as soft real-time suitable for multimedia-oriented data processing, a sophisticated distinction between hard and soft real-time is made in Chapter 5.
This definition is based on sufficiency functions that combine provided and perceived values of tasks at a point in time. It clearly expresses soft real-time from the perspective of operating system activities such as processes or threads. Projecting this notion of real-time to the IHF requires the assumption of a certain system architecture. The stated architecture is a component extension system as developed at the chair of operating systems at the Universität des Saarlandes.
The projection onto the IHF results in a set of indicators for soft real-time properties that are traceable in the IHF subsystem of a GPOS. According to best practices of software quality measurement, these indicators are assigned to the ISO quality factors concerning soft real-time capabilities.
The need for the developed indicators to be formalized and then shown in the concrete models follows immediately. From the grouping into architectural, control-based and data-based indicators, three different techniques for indicator analysis are developed. Architectural indicators are evaluated based on static model characteristics. For control-based indicators, a graph representation of the statechart models is used. Finally, the data-based indicators are verified by means of exhaustive state space enumeration. For the latter, a model representation allowing for temporal logic model checking is created. These techniques are elaborated in Chapter 6. In Chapter 7, the analysis results are presented and interpreted with respect to their impact on real-time multimedia processing in different GPOS.
A summary of the work and an outline of future research topics related to the scientific goals conclude the thesis in Chapter 8.

1.4 Related Work: Requirements Verification in Operating Systems

There exist a variety of approaches to formally verify specifications in operating systems. Bevier [Bev89] reported on verifying parts of the minimal and highly experimental operating system KIT (Kernel for isolated tasks) by means of the Boyer-Moore logic in 1988. The target architecture was a virtual machine and the operating system itself only covered a few fundamental aspects of process handling.
A further variety of verification approaches for specifications with narrow scope can be found in the literature. Klein [KT04] et al. reported on how to verify the virtual memory described by the L4 API specification and link their formal models to the code. Although they discussed the Pistachio implementation, the work has not been applied to the Pistachio system so far. A formal specification of the system call interface and the existing system calls in the Mach kernel was presented by Bevier [BS94].
The VFiasco project was a more comprehensive approach to directly verify security properties in the Fiasco μ-kernel operating system, a C++ reimplementation of the L4 API. A series of reports starting with [THH01] outline the efforts spent by Hohmuth et al. to

directly apply verification techniques to the C++ implementation of Fiasco.
Since all the systems investigated so far are quite small (i.e. about 10k to 15k lines of code), all those verification approaches derive their representations directly from the code instead of creating standalone models. For GPOS with their huge number of of code lines, this approach is not feasible.
In the context of the Verisoft project[1], an academic system based on a μ-Kernel operating system, pervasive verification of the OS is part and parcel of the research activities [DHP05] [GHLP05]. The goal of this verification approach is to prove the correctness of all operating system modules. To achieve this, all modules are written in a reduced dialect of C. These research activities differ from those performed for this thesis where the focus is on evaluation of a somewhat abstract property (real-time) in original implementations of GPOS. While the Verisoft approach is bottom-up in nature to answer questions on how a verifiable system has to be designed, this thesis presents a top-down approach to real-world OS.

All in all, there are two essential differences between all these related research activities and the approach in this thesis: First, only minimal, modified or experimental systems with strict prerequisites are regarded, and there is as yet no investigation of real-world GPOS. Secondly, the questions to be answered are very precise and strongly related to the facet that is investigated (e.g. memory management verification). General, abstract properties such as real-time capabilities of a complete system have not been analyzed so far.

1.5 Basic Definitions

Since this thesis uses some terms that need a common understanding, these are given in this short section.

Definition 1.1 ***System***
A system is a set of entities or components that work together in an integrated way. Every system is characterized by its boundaries, i.e. its frontiers with the environment.

A special case for a system is a reactive system as defined by Schneider [Sch04]:

Definition 1.2 ***Reactive Systems***
In a reactive system, the environment freely determines the points in time when an interaction is desired. The system itself only reacts to the occurring external stimuli.

Applying those definitions, every operating system can be seen as a reactive system. The interrupts that are triggered by the external hardware circuits occur randomly from an internal perspective.

Definition 1.3 ***System under Investigation (SuI)***
Derived from the term SuD: System under Development [HLN+90], a system under investigation is an existing real-world system that must be examined and whose properties are to be investigated.

[1] http://www.verisoft.de

In this thesis, all SuIs are operating systems and thus reactive systems according to the above definition.

1.6 Conventions

In this thesis, a few typographical conventions are used. Source code (assembly language as well as high level language code) and microprocessor internal handling are depicted verbatim, i.e. a function is denoted as `function_name()`.
Statechart elements are depicted in the typographical style of the statechart figures, e.g. STATE.
Finally, all kinds of formal definitions and formulae are expressed by means of italic letters, e.g. $function : \mathbb{N} \rightarrow \mathbb{R}^{+}$. Sets are depicted by capital Latin letters, whereas elements are represented by small Latin letters.
Free variables within formulae are usually depicted as small Greek letters. Certain predefined functions also use small Greek letters. Auxiliary functions are depicted by abbreviations consisting of three small German letters, e.g. $\mathfrak{aux}()$.
Special, uncommon operators or characters such as $\perp$, $\beth$ or $\div$ will be introduced as they occur.

Chapter 2

Modeling Methods

Modeling is a widely used methodology in computer science. A model can serve multiple different purposes such as design, development, investigation or evaluation of complex systems. According to Stachowiak [Sta73], a model of a system is (1) based upon an original, it (2) reflects only a relevant subset of attributes and properties of the original and (3) serves some dedicated purpose. Kühne [Küh06] gives a brief definition that describes the nature of a model in a nutshell:

"A model is an abstraction of a system allowing predictions or inferences to be made."

One of the perpetual issues that exist when a system is modeled is whether the proper definition of the relevant subset that is necessary for answering the given questions has been captured. In other words, is the chosen *level of abstraction* adequate? It is obvious that a too abstract model might lead to inaccurate conclusions whereas a too detailed model is difficult to manage. The adequateness of the chosen formalism is crucial: its characteristics such as quantitative vs. qualitative modeling, stochastic vs. non-stochastic models and deterministic vs. indeterministic behavior must be carefully chosen.

Early Applications. First serious efforts to model complex computational systems were already taken in the sixties (see e.g. [RB69]). The goal then was to describe systems in a mathematically exact way while the need for abstraction was secondary. At that time, computational systems were small, without parallelism and closely coupled with the underlying hardware circuits. The methods used were based mainly on graph theory.

Modeling Hardware. One of the most popular fields of application for modeling is the design and specification of hardware. The evaluation of computational hardware is also an established area for formal modeling and verification techniques. Precise, quantitative methods are very often applied to deduce or improve the behavior of distributed or parallel systems as well as communication networks [Edw01]. The behavior of a fully loaded system, its reliability and fault tolerance are of special interest. The methods used are often derived from descriptive statistics. Input data for those is usually collected by measurement instrumentation monitors. When designing systems from scratch, graph-based visual methods such as Petri Nets prevail.

Modeling System Software. In the seventies, modeling techniques where first used to describe system software for mainframe computers (see e.g. [Rid72], [CY74] and [AB76]). The software was still small and not multi-threading capable. Today's general purpose operating systems are much larger in size and far more complex because of their concurrent parts.

Modeling Real-Time Systems. In the domain of real-time systems and real-time scheduling, modeling is a widely used methodology to express real-time prerequisites and to prove that they are met. In contrast to the field of general purpose operating systems, strictly limited assumptions and preconditions of hard real-time and embedded systems permit the generation of formal specifications.
In the literature – that is mainly of theoretical origin – real-time models are often simplified and cut down to general scheduling models rather than being wholistic representations. One famous example is the model of the AOCS real-time control system for the Olympus satellite [BW95].

Modeling Application Software. Nowadays, the design and implementation of all kinds of software can be alleviated by modeling some dedicated structural and behavioral aspects of it. The Unified Modeling Language UML [OMG05] is the agreed industry standard that has emerged during the last decade.

2.1 Preliminary Considerations

2.1.1 Transformational vs. Reactive Systems

Operating system software is designed to encapsulate the functionality of its underlying hardware. The interrupt handling facility is in continuous interaction with the hardware: it is supposed to react to each input as soon as it occurs. Wieringa [Wie03] lists those characteristics as the properties of a *reactive system*. It is highly state-dependent. The correctness of its output depends more on the state the system was in when the input occurred than on the input itself. We distinguish between such a system and a transformational (i.e. purely computational) one.
Formalisms that are suitable to model reactive systems differ greatly from those that are adequate for transformational ones as the latter do not have to handle any external stimuli.

2.1.2 Stochastic vs. Deterministic Modeling

The occurrence of external stimuli is without doubt of a stochastic nature. However, the models to be created describe the systems' deterministic reaction on those stimuli – their occurrence is rather a necessary fact than a degree of freedom for the models. In a nutshell: the deterministic reaction to (stochastic) events is modeled and evaluated, but not their indeterministic appearance itself. Subsequently, stochastic formalisms are not applicable for our purposes.

2.2 Formal Techniques

As already indicated, sophisticated formalisms, formal description as well as modeling techniques as well as languages have been invented and tools to support them have been developed in nearly four decades of research. All formal techniques are based on some formal language or specification for a precise description and communication of systems together with a corresponding mathematical meaning – the semantics [Mon03]. The underlying theory is potent and mature.
This section discusses which non-stochastic methods seem promising for modeling general purpose operating systems and lists a few characterizing examples.

2.2.1 Visual Formalisms

A variety of visual formalisms exist that can possibly serve as a basis for the necessary modeling capabilities. A formalism is considered to be visual when its main representation is in visual form. Nonetheless, visual formalisms also have an equivalent textual representation. Languages fulfilling this property are called visual or diagrammatic languages.

2.2.1.1 Graphs

Graphs are the oldest visual formalism known, invented by Euler in the 18th century [Eul72]. Although graphs are a very old-fashioned formalism with limited expressiveness and no manageability when it comes to larger models, they are the precursors of many modern techniques.

2.2.1.2 Automata

Automata model state transition systems [Koz97]. The simplest example are deterministic, finite automata (DFA) [Mea55] [Moo56]. This formalism can be used to define sets that are accepted by grammars or to visualize state transition systems. Nondeterministic finite automata (NFA) [RS59] introduced nondeterministic concepts, but were proven to be no more powerful than DFAs.

2.2.1.3 State Transition Nets

Transition nets were developed from graphs. Petri Nets [Pet62] and place/transition systems (P/T) [Rei87] are well-known and examined basic formalisms. There exist a variety of (non-stochastic) enhanced Petri Nets. Simple condition-event-systems [GLT80] or elementary net systems [RT86] can be seen as simplified Petri Nets with several limitations e.g. in their token structure. Coloured [*sic!*] Petri Nets [Jen91] and Timed Petri Nets (e.g. [Wan98]) are far more advanced concepts. Hierarchy can also be added to Petri Nets, as shown by Fehling [Feh93]. The different approaches can even be combined resulting in powerful high-level formalisms like HCPN (Hierarchical Coloured Petri Nets [HN04]).

2.2.1.4 Other High-Level Formalisms

The unified modeling language UML 2.0 [OMG07] contains several different diagrammatic languages such as statecharts, class diagrams and sequence charts. A variety of tools such as Rhapsody[1] support the application of UML from system specification to verification (see e.g. [STMW04]). Apart from UML, statecharts are a powerful standalone formalism [Har87] on a solid mathematic foundation and supported by a sophisticated tool suite called Statemate[2] [HP98].

2.2.2 Textual Formalisms – Description Languages

Textual formalisms – such as languages – from which visual representations can be directly derived and interactively used are also considered as a possible choice for an appropriate formalism. Pure languages such as Esterel or Z that do not come with some visual equivalent are not taken into account.
The Language Of Temporal Ordering Specification (LOTOS) invented by Turner [Tur87] is based on the algebras CCS and CSP (see Section 2.2.3) and is standardized by the International Organization for Standardization (ISO). A variety of tools support the usage of LOTOS.
The specification language Estelle is also ISO-standardized [ISO89]. Together with the commercial Estelle Development Toolset (EDT), it forms a powerful technique for specifying and verifying real-time systems as well as embedded software.
The Specification and Description Language (SDL) (e.g. [SH01]) is a formal description technique invented and standardized by the International Telecommunication Union ITU-T in their recommendation Z.100. It is mainly used to model hard real-time and communication systems [MT01]. These three languages can also be combined [ISO91].

2.2.3 Algebras

Beside the already mentioned visual and textual formalisms, there are also purely mathematical ones such as algebras. Usually, these serve as the underlying theory rather than as an adequate modeling technique. Two famous process algebras that facilitated many modeling techniques are Communication Sequential Processes (CSP) and Calculus of Communicating Systems (CCS). Hoare invented CSP [Hoa78], a textual but algebraic formalism that describes how to deal with concurrency, synchronization etc. in a formal but intuitive way. The CCS is a process algebra designed by Milner [Mil80] to specify concurrent systems and to reason about them [Mil82]. As mentioned above, although they are important milestones towards formal methods on their own, they cannot serve as an adequate modeling technique.

2.3 Comparison and Usability Studies

Expressiveness (column EXP in Table 2.1) is a crucial criterion for choosing the optimal formalism. It is essential to have the possibility to perform formal analysis and verification.

[1] http://www.telelogic.com
[2] ibidem

A key element for that is a formally defined semantics as well as a structured syntax (column FSS). Furthermore, the formalism must be adequate for modeling hierarchy (HIE), concurrency (CON) and recursion (REC). Moreover, it is crucial that models containing a considerable amount of hierarchy, concurrence and recursion are still manageable (MAN) with respect to their size and complexity. The visual representation of the model must be fully equivalent to the textual representation and provide interactive usability (VIS). This claim results from the fact that the models will be created by manually reverse engineering operating systems source code instead of automatically interpreting them.
The scale is defined as follows: - (not available), 0 (deficient), + (applicable), ++ (well suited).

Formalism	EXP	FSS	HIE	CON	REC	MAN	VIS
Graphs	0	+	-	-	-	0	+
Automata	0	+	-	-	-	0	+
Petri Nets (PN)	0	+	-	0	-	0	++
Colored PN (CPN)	+	++	-	0	0	0	++
Hierarchical CPN	++	++	+	0	0	+	++
Unified Modeling Language	++	0	++	++	+	+	++
Non-UML Statecharts	++	++	++	++	+	+	++
Spec./desc. languages	+	++	0	++	0	0	0
LOTOS	+	++	0	++	0	0	0
Estelle	+	++	0	++	0	0	0

Table 2.1: Comparison of available formalisms for modeling GPOS

Conclusion
Some formalisms do not fulfill the basic requirements mentioned above at all. HCPN, UML and the non-UML statecharts are the most promising candidates after a first brief evaluation of Table 2.1.
Recursion, concurrency and a large number of interfaces to the environment become serious issues when modeling large reactive systems such as OS. The majority of the existing formal methods briefly presented in the former sections generate huge, nearly unreadable models when applied to such systems or lack the possibility to do so at all. Beside that, practical features like standardization or tool support are not provided for all formalisms. These features allow for efficient reverse engineering and the evaluation of large systems. As concurrency and recursion are difficult to model in HCPN as well, this formalism is not suitable for the given application.
As Table 2.1 and the preceding discussion show, the most promising foundations for addressing the problems mentioned above are UML and the statechart formalism. Due to the fact that still today UML lacks a precise formal semantics, the preferable choice is conventional, non-UML statecharts.

2.4 Statecharts

Statecharts where invented during the eighties by Harel [Har87]. They provide modularity, hierarchy, orthogonality and broadcast communication as inherent characteristics [Har88]. Statecharts also offer formally defined basic semantics [HPSS87] (whose shortcomings will be discussed in Section 2.4.4). The adequateness of this formalism in modeling reactive systems is proven [HLN+90].
Statemate is a tool suite widely used for the design of reactive systems [HP98]. The related Statemate research resulted in a more sophisticated semantics [HN96] for statecharts. With the introduction of UML 2.0 [OMG07] and the inclusion of statecharts as the main behavioral description language, they also became an industry standard. In this thesis, only the notational style (i.e. the way of drawing state diagrams) of UML statecharts and their concept of submachines (cp. Section 4.2.3.5) are used, their syntax or semantic components (cp. Section 2.3) are omitted.

2.4.1 Introduction

This section explains the statechart formalism in an entirely informal but very intuitive way by discussing a working example. In Section 2.4.2, the formal foundations and definitions are given after gaining a first heuristic understanding of the nature of statecharts. As a working example, a toy model of a simple multimedia car audio and navigation system is used, see Figure 2.1.

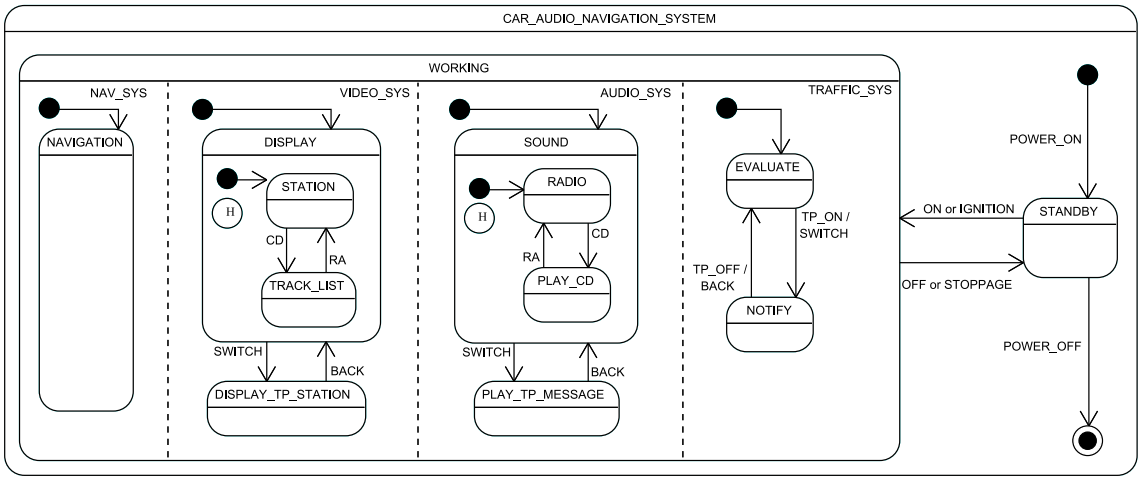


Figure 2.1: Working example: multimedia car audio navigation system, focus on traffic messages

There exist ten external inputs (*events*) to the statechart model represented by its *root state* CAR_AUDIO_NAVIGATION_SYSTEM, namely four user inputs (assumed to be buttons) ON, OFF, CD and RA as well as four sensor inputs IGNITION, STOPPAGE, POWER_ON and POWER_OFF. Finally, there are two events that symbolize the beginning and ending of a traffic program (TP), namely TP_ON and TP_OFF.
The system is switched on when the *event* ON or IGNITION occurs: then, the *transition* (arrow) reacting on such an event is taken. This changes the system's *state* from STANDBY

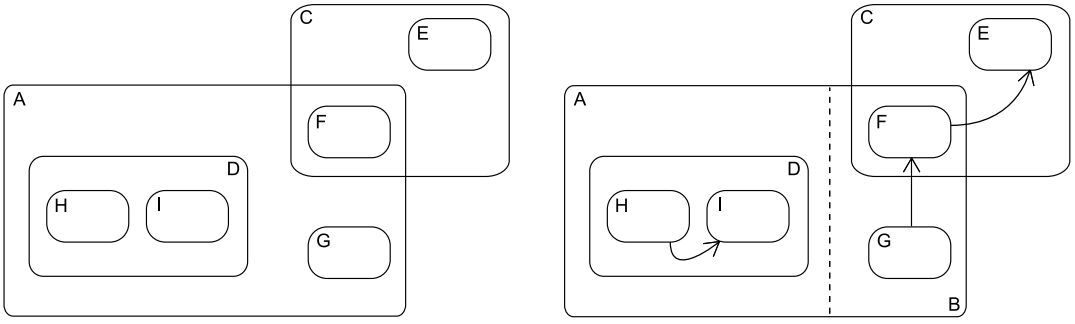
to WORKING and thus the four concurrent sections NAV_SYS, VIDEO_SYS, AUDIO_SYS and TRAFFIC_SYS are all entered simultaneously. WORKING is therefore called an *AND* state, the parallel substates are marked by the dashed lines. All other states are *XOR* states, i.e. the system can only be in one substate of such a state at a time. The navigational subsystem is not elaborated further[3], it just runs without any interaction until the system is switched off (event OFF or STOPPAGE occurs).
The transitions originating from the black filled circular *connectors* point to the default substate of each state. DISPLAY and SOUND are entered, and the system switches immediately to play radio music (RADIO) and display the station (STATION). When the user changes the source of music by pressing the CD button, this event is sent throughout the system causing it to switch to play the CD (PLAY_CD) and display the track list (TRACK_LIST).
The fourth parallel subsystem TRAFFIC_SYS evaluates the radio channel and listens for traffic messages. When a TP begins (event TP_ON), it changes its internal state from EVALUATE to NOTIFY and raises the internal event SWITCH. This raising is called an *action*. As a result of this internal event, the video system immediately takes the transition from DISPLAY to DISPLAY_TP_STATION and the audio system from SOUND to PLAY_TP_MESSAGE. When this happens, the current substate (RADIO or PLAY_CD) is left as well. This reaction crossed the boundaries of a concurrent component, the event was *broadcasted* all throughout the statechart. When the traffic subsystem senses the stimulus TP_OFF and generates the internal event BACK, the video and audio subsystem take their transition back to DISPLAY or SOUND respectively. This time, the substates of DISPLAY and SOUND are not the default ones, but the *history* (represented by circles with letter H) of these states is applied. This means that the most recently active substate of DISPLAY and SOUND respectively is chosen as currently active state.
It is obvious that states form a *hierarchy* due to their *insideness* with the deepest nesting level being four (e.g. RADIO, SOUND, WORKING, CAR_AUDIO_NAVIGATION_SYSTEM), this depth allows for simple abstraction and modularization.
Besides a mere demonstration of the statechart syntax, this working example already shows a very important paradigm of modeling systems by means of a visual language such as statecharts: different foci can easily be expressed by different levels of abstraction. In the example, the focus is on the handling of traffic messages alone whereas the entire behavior of the navigation system is blanked out.

2.4.2 Formal Definition

Statecharts are formally based on two different ideas [Har88] : On the one hand, they are predicated on Euler-Circles [Eul72] and the later Venn diagrams [Ven80]. These diagrams are a profound way to represent sets, collections and the structural relations between them. The intended semantics is uniformly interpreted in a set-theoretic manner.
On the other hand, statecharts are based on the concept of connecting entire sets of nodes rather than a single pair of nodes: a property originally provided by hypergraphs [Ber73], a graph extension. Hypergraphs are specifically designed to depict sets, their elements and special inter-set relations. The meaning of the edge relation can be interpreted freely depending on the field of application.

[3]This part of the system will become a distinct working example later in the thesis (cp. Chapter 5).

In order to exploit both ideas, higraphs [Har88] were invented. They are composed of slightly modified Euler-Circles that are enriched with the cartesian product of their sets. The latter are called blobs and can be nested and overlapping. Finally, the sets and n-tuples are connected by edges or hyperedges, see Figure 2.2(a). The concept of higraphs is widely used in computer science, e.g. for entity relationship diagrams [Che76] or semantical networks [Sha71].



(a) blobs and subblobs (b) edges, hyper-edges and the cartesian product

Figure 2.2: An example higraph with eight blobs

Definition 2.1 ***Higraph***
A Higraph is defined as a quadruple $\mathfrak{H} \stackrel{\text{def}}{=} (B, E, \sigma, \pi)$ *with* B *being the finite set of elements called blobs,* E *the set of edges,* $E \subseteq B \times B$*, the subblob function* σ *and a partitioning function* π*.*

Definition 2.1.1 ***Subblob Function*** σ
The subblob function $\sigma : B \to 2^B$ *yields all subblobs for each blob. The closure of* σ*, named* σ^+*, is recursively defined as:*

$$\sigma^+(x) \stackrel{\text{def}}{=} \bigcup_{i \geq 1} \sigma^i(x)$$

$$\sigma^{i+1}(x) \stackrel{\text{def}}{=} \bigcup_{\tilde{x} \in \sigma^i(x)} \sigma(\tilde{x})$$

$$\sigma^0(x) \stackrel{\text{def}}{=} \{x\}$$

In anticipation of the hierarchy function (cp. Definition 2.2.1) of statecharts, σ *has to be cycle-free, i.e.* $\forall\, x \in B : x \notin \sigma^+(x)$*.*

Definition 2.1.2 ***Partitioning Function*** π
The partitioning function $\pi : B \to 2^{B \times B}$ *determines the partitioning of blobs. It is defined as an equivalence relation, let* $[x]_\pi, x \in B$ *be the equivalence classes induced by* π*.*

Definition 2.1.3 ***Intersection of Subblobs***
For $x \in B$ *it holds that no two subblobs* $y, z \in \sigma(x)$ *can intersect unless they belong to the same orthogonal component (partition):*

$$\forall\, y, z \in (\sigma(x) \in B) : (\sigma^+(y) \cap \sigma^+(z) \neq \emptyset \Rightarrow [y]_\pi = [z]_\pi)$$

Statecharts are now a "higraph-based version of finite state machines and their transition diagrams" [Har88]. Each blob A to I in Figure 2.2 represents a certain set; this "unique contour convention" then allows for unambiguously identifying and labeling all sets. The nesting (e.g. blobs D and H) of blobs thus depicts set inclusion, overlapping of blobs (A and C) denotes set intersection. By adding a cartesian product – the dashed line in Figure 2.2(b) – to those sets, parallelism can be achieved. The resulting parts are called orthogonal components. Edges or hyperedges – the arrows in Figure 2.2(b) – represent system transitions while blobs depict system states.
Overlapping blobs within a single orthogonal component are allowed by the higraph definition. It turned out to be extremely difficult to define a proper semantics for this and to be not as beneficial as expected [HK92] (see also Section 3.5). Therefore, the following complete formal definition of statecharts prohibits overlapping blobs, i.e. states, per se. It is based on [HPSS87].

Definition 2.2 ***Statechart***
A Statechart is defined as a 12-tupel: $\mathfrak{S} \stackrel{\text{def}}{=} (S, \rho, \psi, H, \gamma, \delta, V, C, E, A, L, T)$ *with* $S \neq \emptyset$ *the finite set of states,* H *the finite set of history symbols and the following elements:*

Definition 2.2.1 ***Hierarchy Function*** ρ ***and Root State*** r
The hierarchy function $\rho : S \rightarrow 2^S$ *denotes all the direct substates or descendants of a state. A state* s *is called basic iff* $\rho(s) = \emptyset$*. For all* $x, y \in S$ *it holds that:* $\rho(x) = \rho(y) \Rightarrow\ x = y$*. The closure of* ρ *is recursively defined as:*

$$\begin{aligned}
\rho^+ &\stackrel{\text{def}}{=} \bigcup_{i \geq 1} \rho^i(s) \\
\rho^* &\stackrel{\text{def}}{=} \bigcup_{i \geq 0} \rho^i(s) \ \textit{where} \\
\rho^0(s) &\stackrel{\text{def}}{=} \{s\}\ ,\ \rho^1(s)\ =\ \rho(s)\ , \\
\rho^{i+1}(s) &\stackrel{\text{def}}{=} \bigcup_{\tilde{s} \in \rho(s)} \rho^i(\tilde{s})
\end{aligned}$$

Every statechart contains a state r *called root state:*

$$\exists\, r \in S :\ \left[\forall\, s \in S : [r \notin \rho(s)]\right]$$

The two extended functions $\rho^+(s)$ and $\rho^*(s)$ depict all substates of a state $s \in S$ down through the entire state hierarchy. Note that $\rho^+(s)$ does not include the very state s itself while $\rho^*(s)$ does.

Definition 2.2.2 ***Type Function*** ψ
The type of each state is determined by the type function $\psi : S \rightarrow \{AND, XOR\}$*. If* $\rho(s) \neq \emptyset$ *and* $\psi(s) = XOR$ *,* $\rho(s)$ *is an XOR decomposition of* s*, i.e. when the system is in a state* s*, it is in exactly one of the substates of* s*.*
If $\rho(s) \neq \emptyset$ *and* $\psi(s) = AND$*,* $\rho(s)$ *is an AND decomposition of* s*, i.e. when the system is in a state* s*, it is in all substates of* s *simultaneously.*

In the literature, the *XOR* type is often named *OR*. Due to the fact that the system is in exactly one substate only, this notation is inaccurate.

Definition 2.2.3 ***Historic State Function*** γ**,** ***Mapping Function*** ω
The historic state function $\gamma : H \to S$ *yields the assigned XOR state for a history symbol. For all* $h_1, h_2 \in H$ *it holds that* $\gamma(h_1) = \gamma(h_2) \Rightarrow\ h_1 = h_2$ *and*

$$\forall\, h \in H : \gamma(h) \subset \{s \in S \,|\, \psi(s) = XOR\}$$

The mapping function $\omega : H \cup S \to S$ *is a supporting function to avoid the need to distinguish between a history symbol and its associated XOR state:*

$$\omega(x) \stackrel{\text{def}}{=} \begin{cases} x & | \; x \in S \\ \gamma(x) & | \; x \in H \end{cases}$$

Each history symbol is assigned to an *XOR* state using the γ function. *AND* states cannot have a history symbol.

Definition 2.2.4 ***Default Function*** δ
The default function $\delta : S \to 2^{S \cup H}$ *yields a set of states and history symbols contained in a state.*

$$x \in \delta(s) \;\Rightarrow\; \begin{cases} x \in \rho^{+}(s) & | \; x \in S \\ \gamma(x) \in \rho^{*}(s) & | \; x \in H \end{cases}$$

The set $\delta(s)$ *is denoted as the default set for* s*.*

Definition 2.2.5 ***History Traversal Function*** τ
For a given sequence of system configurations (cp. Definition 2.5) $(X_0, \ldots, X_n)$ *and a history symbol* $h \in H$ *where* $\exists s \in S\ :\ \gamma(h) = s$*, we define the history traversal function* $\tau : H \times (2^S)^n \to S \cup 2^{S \cup H}$ *:*

$$\begin{aligned} I &= \{i \,|\, \rho^{*}(s) \cap X_i \neq \emptyset\} \\ \tau(h, (X_0, \ldots, X_n)) &\stackrel{\text{def}}{=} \begin{cases} \delta(s) & | \; I = \emptyset \\ s' \in \rho(s) & | \; else \\ \quad where\ \rho^{*}(s') \cap X_j \neq \emptyset,\ j = max(I) & \end{cases} \end{aligned}$$

for $i, j, n \in \mathbb{N}_0$

The history traversal function yields the most recently visited substate s' of a state s. If there was no visit of s yet, the default function δ is applied. Since $\delta(s)$ returns a set of states and history symbols rather than a single state, the range of τ has to be $S \cup 2^{S \cup H}$. Note that in the semantics, the default set will be defined by means of connectors (cp. Definition 2.7).

Definition 2.2.6 ***Expressions*** V
The set of primitive variables is V_p*. The set of expressions* V *is defined inductively as follows:*

$$\begin{aligned} k \in \mathbb{N}_0 \quad &\Rightarrow k \ \in\ V \\ v \in V_p \quad &\Rightarrow v \ \in\ V \\ v \in V \quad &\Rightarrow cr(v) \ \in\ V \\ v_1, v_2 \in V \quad &\Rightarrow (\ v_1 \ \langle + | - | * | / \rangle \ v_2\ ) \ \in\ V \end{aligned}$$

The expression $cr(v), v \in V$ refers to the current value of v within a system step (cp. Section 2.4.4).

Definition 2.2.7 ***Conditions*** C
The set of primitive conditions is C_p. The set of conditions C is defined inductively as follows:

$true, false$	$\in C$
$c \in C_p$	$\Rightarrow c \in C$
$s \in S$	$\Rightarrow in(s) \in C$
$e \in E$	$\Rightarrow ny(e) \in C$
$c \in C$	$\Rightarrow cr(c) \in C$
$v_1, v_2 \in V$	$\Rightarrow (v_1 \div v_2) \in C$, *where* $\div \in \{==, !=, <, >, \leq, \geq\}$
$c_1, c_2 \in C$	$\Rightarrow (\ c_1\ \langle \wedge \vert \vee \rangle\ c_2\ ) \in C, \neg c_1 \in C$

The condition $cr(c), c \in C$ is defined in the same way as above, $in(s)$ is true when the system is in state s while $ny(e)$ is true when an event e according to Definition 2.2.8 has not yet occurred during a system step.

Definition 2.2.8 ***Events*** E
The set of primitive events is E_p. The set of events E is defined inductively as follows:

atomic events:	
$\lambda \in E$	*null event*
$e \in E_p$	$\Rightarrow e \in E$
$c \in C$	$\Rightarrow tr(c), fs(c) \in E$
$v \in V$	$\Rightarrow ch(v) \in E$
$s \in S$	$\Rightarrow ex(s), en(s) \in E$
compound events:	
$e_1, e_2 \in E$	$\Rightarrow (\ e_1\ \langle \wedge \vert \vee \rangle\ e_2\ ) \in E, \neg e_1 \in E$
$e \in E, c \in C$	$\Rightarrow e[c] \in E$

The events $tr(c)$ and $fs(c)$ represent the boolean evaluation of a condition: $tr(c)$ is the event created when the condition is true, $fs(c)$ when it is false. The events $en(s)$, $ex(s)$ are raised when a state s is entered or exited, $ch(v)$ when an expression v changes its value.

Definition 2.2.9 ***Actions*** A
The set of actions A is defined inductively as follows:

atomic actions:	
$\mu \in A$	*null action*
$e \in E$	$\Rightarrow gen(e) \in A$
$c_1 \in C_p, c_2 \in C$	$\Rightarrow (c_1 := c_2) \in A$
$v_1 \in V_p, v_2 \in V$	$\Rightarrow (v_1 := v_2) \in A$
compound actions:	
$a_i \in A, (i \leq n) \in \mathbb{N}_0$	$\Rightarrow (a_0; ...; a_n) \in A$

An action can be regarded as an output in automata-theoretic terms. The action $gen(e)$ generates an (internal) event e that is sensed throughout the system. The sequence $(a_0; ...; a_n)$ denotes the concatenation of actions a_0 to a_n.

Definition 2.2.10 ***Labels*** L
The set of labels is a set of pairs $L \stackrel{\text{def}}{=} (E \times A)$. *A label* $l \in L$ *is called trivial iff* $l = (\lambda, \mu) \in L$.

Adopting the notation of finite state machines, e/a is written instead of (e, a). Furthermore, if it is clear from the context, e_1/e_2 is used for $e_1/gen(e_2)$ with $e_1, e_2 \in E$.

Definition 2.2.11 ***Transitions*** T
The set of transitions is a set of triples $T \subset (2^S \times L \times 2^{S \cup H})$. *In a transition* $t = (X, l, Y)$, X *is denoted as the source set of* t *(*$X = src(t)$*),* Y *as the target set of* t *(*$Y = tgt(t)$*).*

With the definition of transitions, the syntax of statecharts is complete.

2.4.3 Attributes, Properties and Notations

To clarify the syntax and to facilitate the later semantics definition, some additional specifications and notations based on [HPSS87] and [PS91] are now given.

Definition 2.3 ***Lowest Common Ancestor*** $\mathfrak{lca}$
For any set of states $X \subseteq S$, *the lowest common ancestor* $\mathfrak{lca}(X)$ *is defined as follows:*

$$\mathfrak{lca}(X) = x \in S \;\Leftrightarrow\; X \subseteq \rho^*(x) \text{ and } \forall\, s \in S : [X \subseteq \rho^*(s) \Rightarrow x \in \rho^*(s)]$$

The lowest common ancestor (LCA) of a set of states X is the topmost (with regard to the state hierarchy) superstate that contains all elements of X. For any statechart, there exists a unique LCA for every set of states $X \subseteq S$, being at least the root state.

Definition 2.4 ***Orthogonality***
Two states $x, y \in S$ *are orthogonal, denoted as* $x \perp y$, *if either* $x = y$ *or* $\psi(\mathfrak{lca}(x, y)) =$ *AND*. *A set of states* X *is an orthogonal set iff* $\forall\, x, y \in X : (x \perp y)$.
An orthogonal set X *is an orthogonal set relative to* $s \in S$ *if* $X \subset \rho^*(s)$. *It is called maximal orthogonal set relative to* s *if* $\forall\, y \in \rho^*(s)$, $y \notin X \Rightarrow X \cup \{y\}$ *is not orthogonal.*

A set of states being orthogonal means that they are in different orthogonal components of the statechart. The system can be in all those states at the same time (parallelism). Hence, a component of a statechart that is part of an *AND* state and thus has at least one orthogonal counterpart is called *concurrent component*.
From the definition it follows that for every $s \in S$, the set $\{s\}$ is a maximal orthogonal set relative to s.

Definition 2.5 ***Configurations***
A state configuration of $s \in S$ *is an orthogonal set* X *relative to* s *where all* $x \in X$ *are basic states (i.e. states without substates, see Definition 2.2.1).*
A maximal state configuration of $s \in S$ *is a maximal orthogonal set* X *relative to* s *where all* $x \in X$ *are basic.*
A system configuration is a maximal state configuration relative to the root state r.
The initial system configuration is a system configuration X_0 *where the system is in the root state only and no system steps have occurred yet.*

A system configuration contains all basic states that the system is concurrently in at one point in time.

Definition 2.6 ***Properties of Transitions***
The arena function $\mathfrak{arena} : T \to S$ *defines for any transition* $t \in T$ *the XOR state which contains both its source and target states* $src(t), tgt(t)$.

$$\mathfrak{arena}(t) \stackrel{\text{def}}{=} x \Leftrightarrow x = \mathfrak{lca}(\{src(t), tgt(t)\}) \text{ and } \psi(x) = XOR$$

Two transitions $t_1, t_2 \in T$ *are consistent if either* $t_1 = t_2$ *or* $\mathfrak{arena}(t_1) \perp \mathfrak{arena}(t_2)$. *Otherwise,* t_1 *and* t_2 *are in conflict. A set of transitions* $T_1 \subseteq T$ *is consistent if* $\forall\, t_1, t_2 \in T_1$: t_1 *consistent* t_2.

A set of consistent transitions can be simultaneously processed during one system step. The definition of system steps will be the crucial part of the statechart semantics.

2.4.4 Semantics

In the early years of the statechart formalism, Harel, Pnueli et al. published a formal definition of the statechart semantics [HPSS87]. This early approach was later criticized not only by various other researchers like Huizing [HG88] but also by Pnueli himself [PS91]. Von der Beeck [vdB94] provides an extensive list of deficiencies that where unveiled in the original semantics over time and lists over 20 statechart variants and their different semantical approaches. Harel later published a revised semantics [HN96].

2.4.4.1 Categories of Semantics

In the literature (e.g. [NN92]), three main distinct ways of giving formal semantics are described. There exist a few other approaches such as algebraic or action semantics [SK95] but these are not widely used.

1. *Operational Semantics*: The main goal of operational semantics is to define *how* a program, a language construct or, more generally, a syntactical construct is executed on a virtual machine rather than only defining its result. A virtual machine could be some abstract formalism like a term replacement system, a graph or Kripke structure (cp. Definition 6.8). Two approaches are possible when dealing with operational semantics:

   (a) *Structural operational semantics (SOS)*: This describes "small step semantics", i.e. a rule set that defines a step by step conversion to arrive at the underlying formalism.

   (b) *Natural semantics*: The contrary case to SOS depicts a kind of "big step semantics": axioms and rules are given to allow one to prove that a calculation terminates and to say what the result will be. That is, there is a big step directly from the start to the final result with no intermediate steps.

   Defining operational semantics is the natural way a compiler designer or a manufacturer for an interpreter of a language would choose.

2. *Denotational Semantics*: The behavior of a language can be defined by creating functions that represent the mapping of initial inputs to final results, i.e. a mathematical model is given. In contrast to the operational approach, this concentrates on the results rather than on the procedure itself. Denotational semantics always include compositionality of the syntactic categories and declarativity of the functions that define the meaning.
   Due to the fact that this method abstracts from any concrete implementation, it is a language designer's choice.

3. *Axiomatic Semantics*: A third way of defining behavior is to postulate pre- and post-conditions. This is usually done by means of a logical calculus. This is a very common approach to showing the equivalence of languages and to proving the total correctness of a language or of program sections.

Since operational semantics are most constructive in a mathematical sense and are very specific in defining the behavior of their components, they are the first choice for the purposes of enhancing a formalism and interpreting its meaning.

2.4.4.2 The Synchrony Hypothesis

During development of the synchronous programming language Esterel by Berry et al. [BG92] it became obvious that there exist major problems such as competing reactions or differing perceptions of the timescale within a system. One simple solution to these problems was to make an assumption about the system's reaction time: Each reaction is assumed to happen instantaneously and thus is atomic to its trigger.
This assumption, named *synchrony hypothesis*, was later adopted in different areas of application. A weaker version of the synchrony hypothesis is applied to very-large-scale-integration (VLSI) circuits: Any system reaction takes less than one clock cycle. In a way, this is also atomic with regard to the timing granularity.
The focus of this thesis is to model the part of an operating system software that reacts to external stimuli – interruptions. Although such a disruption can occur at any time in the continuous real-world time spectrum, the hardware reaction to that stimulus is always synchronized with clock cycles. When the CPU clock cycles are defined as a time scale (cp. Section 5.2.3.1), the synchrony hypothesis holds for all software models that always have a coarser grained granularity. In other words, no part of the SuI could possibly realize that some reaction is not atomic with regard to the real-world timescale.

2.4.4.3 Statechart Semantics

The original statechart semantics [HPSS87] applies the synchrony hypothesis. This operational approach (SOS) is based upon steps leading from one system configuration to another. In other words, there exists a next-step relation between a configuration and each of its legal (with respect to the semantics) configuration outcomes. Events can be generated as outputs of any transition and can trigger new transitions during the very same step. The intricate part is thus to find all transitions that are relevant and that can be taken simultaneously, i.e. that are not mutually exclusive and not conflicting. The basic idea was to introduce micro-steps (μ-step) and then defining a real system step as a

maximal sequence of μ-steps. Subsequently, the sequence of μ-steps represents the order of transitions and actions that make up a whole step. For the execution of each μ-step, a consistently executable subset of of enabled transitions is determined and taken. The evaluation of conditions and actions then is used as input for the next μ-step execution. The expression $cr(v)$ as well as the conditions $cr(c)$ and $ny(e)$ (cp. Definitions 2.2.6 and 2.2.7) particularly expose this internal handling to the outside world, i.e. they can be used to gain explicit control over the internal ordering of μ-steps because they are updated on a μ-step basis.
It is an interesting contradiction that on the one hand, all μ-steps should be executed transparently within an instant, i.e. not consuming any time, and on the other hand, the order of this infinitely short period can affect the overall result of the step considerably. Although the original semantics meets the requirements of synchrony, causality and local consistency, i.e. all μ-steps are pairwise consistent, the missing feature is global consistency (e.g. [HG88], [vdB94]).
A declarative statechart semantics, i.e. a denotational semantics that is not compositional, based on a fix-point equation is elaborated by Pnueli [PS91] confronting the problems of the original semantics. The presented fix-point equation is based on an *enabling function* $en : 2^T \rightarrow 2^T$ that determines for each taken set of transitions the maximal set of transitions that are enabled and can be taken as a consequence.
To show the differences between this declarative semantics and an operational approach, the missing global consistency is exploited. First, an operational step creation procedure such as the one in the basic semantics definition that is equivalent to the declarative semantics is created. The main limitation on the syntax is that only primitive events or negations, but no compound events can trigger actions. Since the enabling function is proven to be concave under these limitations, the equivalence between the step creation procedure and the enabling function can be shown. Thus, the syntactical constraints are set aside which removes the concavity property and thus establishing the equivalence between the two concepts. Under these circumstances, the procedure fails to create the correct steps while the declarative semantics does.
This demonstrates the basic deficiencies in the original operational semantics but does not offer any alternative since it is not completely elaborated in detail and since a declarative or even a full denotational semantics (such as in [HGdR88]) in general does not serve the purposes of this thesis.

Harel and Naamad [HN96] define the semantics used by the Statemate tool in an informal and highly operational way. Each single step leads from one system status to the next. A system status here is in fact a system configuration (cp. Definition 2.5) plus some additional information, e.g. about data items and activities of the Statemate model. A basic step algorithm is provided in pseudo-code.
The main difference between the presented approach and the basic semantics [HPSS87] is that changes occurring during a step (e.g. generated events) affect the following step only and never the current one. Events are valid for the duration of one step only. In other words, the system's memory is volatile. At any step, the subset of transitions that are taken must be maximal. In [HN96], this attribute of the semantics is called *greediness property.*
Due to the comprehensive nature of the Statemate tool and the fact that it embodies

code generation and simulation modules, the semantics provide more concepts than the original statechart semantics does, i.e. static reactions (actions that are carried out while the system is in a specific state), the timing means such as scheduling and timeouts and activity charts.

Since it is the most advanced SOS for statecharts, the Statemate semantics is used in the following. However, the additional features mentioned are not exploited in this work.

The most important semantical concepts for the further application of the Statemate semantics will now be defined:

Definition 2.7 ***Connector***
A connector is an entity that allows the splitting of transitions into several parts.

$$\begin{aligned} CON &= AND\_CON \cup XOR\_CON \cup TERM\_CON \cup DEF\_CON \cup H \\ AND\_CON &= FORK \cup JOIN \\ XOR\_CON &= COND \cup SELECT \cup JUNCT \end{aligned}$$

where $FORK$ is the set of all $fork$ connectors, $JOIN$ the set of join connectors, $COND$ of condition connectors, $SELECT$ and $JUNCT$ of select and junction ones. $TERM\_CON$ and DEF_CON are the sets of termination and default connectors.

Connectors can be seen as "whistle stops" for transitions that are split into parts (segments) and by that enriched with some particular meaning depending on the type of the connector. For example, a *select* connector allows for switching the target of a transition depending on some conditions.
Note that from now on the history symbols $\in H$ will be treated as connectors. The default connectors substantiate the default function δ: the target state of a transition segment (as defined next) that originates from a default connector is the default state of its superstate. In Figure 2.3, the transitions[4] t_2 and t_4 that originate from default connectors point to the default states of the two orthogonal components: C and D.

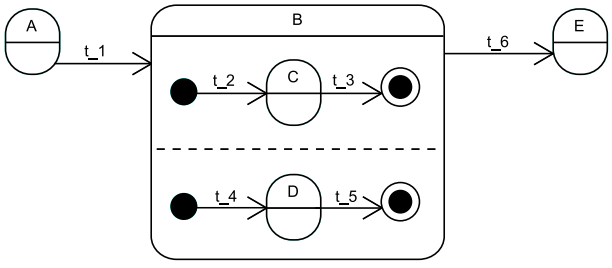


Figure 2.3: Different transition segments t_1 up to t_6 form different compound transitions

[4]For a better understanding, here the strings t_1 to t_6 denote the names of the transition segments rather than their labels. This – later recurring – intentional deviation from statechart standard is purely notational and does not impact syntax or semantics at all.

Definition 2.8 ***Default and Termination Connector Functions***
The default connector function $con_d : S \rightarrow DEF\_CON$ *yields the default connector assigned to a specific state:*

$$con_d(s) = c_d \Leftrightarrow c_d \in \rho(s), c_d \notin \rho^+(s)$$

The termination connector function $con_t : S \rightarrow TERM\_CON$ *analogously yields the termination connector of a state:*

$$con_t(s) = c_t \Leftrightarrow c_t \in \rho(s), c_t \notin \rho^+(s)$$

Definition 2.9 ***Full Transitions, Transition Segments***
A full transition $t \in T$ *leads directly from a set of states* $src(t)$ *to a set* $tgt(t)$ *without being split. If a transition is divided using connectors, the resulting parts are called transition segments. The set of transition segments* TS *is defined as follows:*

$$TS \subset (2^{CON \cup S} \times L \times 2^{CON \cup S})$$

In the example Figure 2.3, t_1, ... , t_6 are transition segments, the combination of e.g. t_1 — t_2 is a full transition with respect to Definition 2.9.

Definition 2.10 ***Compound Transitions***
A compound transition is a sequence of transition segments $ts_0 \mid \ldots \mid ts_n$, $ts_i \in TS$, $0 \leq i \leq n$, $n \in \mathbb{N}_0$. *The set of all compound transitions (CT) is* CT.
A basic compound transition (BCT) is a CT with the following properties (BCT being the set of basic compound transitions):

- *It is a maximal sequence of transition segments that can be taken within one single step as one full transition.*
- $\forall\, BCT \ni bct = (s, e/a, t) = ts_0 \mid \ldots \mid ts_n$ *with* $s \in 2^S, t \in 2^{S \cup H}, e \in E, a \in A$ *and* $0 \leq i \leq n$ *with* $ts_i = (s_{ts_i}, e_{ts_i}/a_{ts_i}, t_{ts_i})$*:*

$$e \stackrel{\text{def}}{=} \bigwedge_{i=0}^{n} e_{ts_i} \quad \textit{and} \quad a \stackrel{\text{def}}{=} (a_{ts_0}; \ldots; a_{ts_n})$$

An initial compound transition (ICT) has the following restrictions (with set ICT*):*
$\forall\, ICT \ni ict = ts_0 \mid \ldots \mid ts_n : src(ts_0) \subset S$ *and* $tgt(ts_n) \subset S \cup H \cup TERM\_CON$.
A continuation compound transition (CTC) has the following restrictions (with set CCT*):*
$\forall\, CCT \ni ict = ts_0 \mid \ldots \mid ts_n : src(ts_0) \subset H \cup DEF\_CON$ *and* $tgt(ts_n) \subset S \cup H \cup TERM\_CON$.

A basic compound transition is taken if all events of the TS labels occur and the BCT triggers all actions defined by these labels. An initial CT always has its origin in a set of states, a continuation CT in a set of default or history connectors. Both lead to either states, history or termination connectors. In Figure 2.3, the combination of e.g. t_1 — t_2 is a basic compound transition, the transition segments t_1, t_3, t_5 and t_6 are initial compound transitions, t_2 and t_4 continuation compound transitions.

Definition 2.11 ***Full Compound Transitions***
A full CT (FCT) is a combination of one ICT followed by one or more CCTs that lead the system to a full system configuration. The following restrictions apply:

- *The sequence of these compound transitions has to be cycle-free.*
- *All source states are mutually orthogonal.*
- *All target states are mutually orthogonal.*

Figure 2.3 also illustrates the differences between FTs and FCTs: in order to reach a valid system configuration, it would be insufficient to take the full transition t_1 — t_2 because a system has to be in all its orthogonal components at the same time. Subsequently, only t_1 — t_2 — t_4 makes a full compound transition. The same holds for t_3 — t_6, this FT is insufficient, thus t_3 — t_5 — t_6 is the full compound transition.
The chosen semantics is predicated on full compound transitions leading from one full system configuration to another.

Chapter 3

Engineering Statechart Formalism

Although the statechart formalism has been identified as the most suitable technique for modeling operating systems, this approach still has shortcomings. The number, multiplicity and complexity of external interfaces and the multi-layer design of an operating system model results in an exploding number of inter-level transitions.
The reason for that is that usually, statecharts are used to describe the behavior of systems that have a bounded, small number of interfaces to the outside world on the one hand and a very detailed and complex "inner workings" on the other. In GPOS, we also have the reverse case: a high ratio of external interfaces in relation to internal behavioral states. The fact that the models of multi-processor systems additionally contain considerable concurrency and the exhaustive use of recursion exacerbates the transition explosion – the model becomes nearly unreadable and thus barely understandable. This not only limits the potential of the model – the goal of visualizing the complicated activities inside the system and gaining a better understanding can barely be achieved – but also impedes formal analysis and verification (cp. Section 6.3) of the model.
The solution presented in this chapter is to define a new set of standardized, use case tailored and UML-compliant statechart elements. These have a basic syntax including modeling rules and a rigorous semantics as described later. This set of constituents together with the conventional statechart elements make up the Engineering Statechart Formalism. Being an extension of statecharts, the resulting method does not lack any formal correctness or expressiveness. Furthermore, it supplements the reverse engineering process of general purpose operating systems by providing simple elements for intuitively modeling OS-inherent paradigms such as recursion, nested function calls and control flow branching.
The technique has been developed iteratively while modeling a number of real-world operating systems. This evaluation and accreditation process culminated in a formalism that can be regarded as formally sufficient while still usable. The following *ESF* constituents will now be defined:

- to reduce the number of explicitly given transitions and labels in the model: path events, split states and combine states
- to reduce the overall number of transitions, particularly inter-level transitions: event busses
- to ease the modeling of recursion: long-term history connectors

- to deal with parallel, grouped systems behavior: cartesian transition sets

Syntax and semantics of these extensions are presented in the following sections. All extensions are based upon the statechart definition as in 2.4.2 and the Statemate semantics as discussed in Section 2.4.4. The given operational semantical transformation rules define how to sequentially translate *ESF* constituents into conventional statechart elements. Later in Section 6.3, these rules will be used to define a step relation as described in Section 2.4.4. A rule has the following format:

$$\textbf{RULE NAME}\ \frac{\textit{ESF}\text{ element(s) to be transformed}}{\text{resulting }\textit{ESF}\text{ or statechart constituent(s)}}\ \Downarrow$$

A simple running example (Figures 3.3, 3.5, 3.7, 3.8 and 3.9) will explain and demonstrate the usage of the *ESF* elements. The example is a very small and comprehensive toy model of an operating system part that handles interruptions, executes a process and runs on two CPUs in parallel.

3.1 Preliminaries of *ESF* Syntax and Semantics

3.1.1 Sequential Transitions

The *ESF* explicitly allows for modeling a situation in which a system simply steps through a sequence of states without any external events triggering this walking-through. Figure 3.1 depicts such a situation. The crucial claim here is that the system has to process the sequence of states STATE_1 to STATE_3 in three distinct system steps. When the transition t_1 contains a trivial label, the system would traverse to STATE_2 in the same system step it reaches STATE_1. We call this the "fast-forward problem".

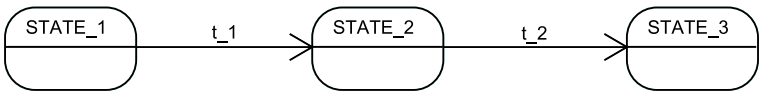


Figure 3.1: Sequential transitions t_1 and t_2

To confront the "fast-forward problem", a special kind of transition called *sequential transition* is defined.

Definition 3.1 ***Sequential Transition***
A transition $(s_1, e/a, s_2) \in T$ *with* $s_1, s_2 \in S$, $e \in E$ *and* $a \in A$ *is called a sequential transition iff* $e = en(s_1)$, s_1 *basic and* $a = \mu$. *In the case where* s_1 *is not a basic state, let* $ts_{last} = (s_{last}, e_{last}/a_{last}, con_t(s_1))$ *with* $s_{last} \in \rho(s_1)$ *be the transition segment that leads to the termination connector embodied in* s_1. *A transition* $(s_1, e/a, s_2)$ *where* $s_1, s_2 \in S$, $e \in E$ *and* $a \in A$, s_1 *not basic is called sequential iff* $e_{last} = en(s_{last}), a_{last} = term(s_{last})$, $e = term(s_{last})$ *where* $term(s_{last}) \in E$ *and* $a = \mu$. *A sequential transition is denoted* t_{seq}.

Due to the nature of the chosen semantics, the special event $en(s_1)$ that is raised when the system entered state s_1 is realized one system step later. This solves the problem in a very easy and elegant way.

In the more complex case that the originating state is not a basic state, the second part of the definition ensures that the substates within the originating state are first processed until the termination connector is reached. Any other behavior would reflect branching rather than sequential execution.
Note that this mechanism would also allow the introduction of timing to *ESF* models by simply redefining the condition that allows stepping further. However, this option is not elaborated further in this thesis.

3.1.2 Enhanced Data Structures for Statecharts

Some of the *ESF* constituents make use of data lists. List functions have to be defined to facilitate handling of these lists.

Definition 3.2 ***Lists and List Functions***
A list is an n-tuple $(x_0, \ldots, x_{n-1})$, $n \in \mathbb{N}$, *of elements of a set* X. *Three list functions are defined:*

$$\begin{aligned} &push: && X \times X^n \to X^{n+1} \\ &push(e, (x_0, \ldots, x_{n-1})) && = (e, x_0, \ldots, x_{n-1}) \\ \\ &tail: && X^n \to X^{n-1}, n \geq 1 \\ &tail(x_0, \ldots, x_{n-1}) && = (x_1, \ldots, x_{n-1}) \\ \\ &get: && X^n \times \{0, \ldots, n-1\} \to X \\ &get((x_0, \ldots, x_{n-1}), i) && = x_i \end{aligned}$$

The set of extended actions is consisting of all actions in A and the list operations defined above:

Definition 3.3 ***Extended Actions*** $\tilde{A}$
The set of extended actions consists of the statechart actions A *and the defined list functions:*

$$\tilde{A} = A \cup \{push(e, l), tail(l), get(l, n)\}$$

where $e \in X, l \in X^n, n \in \mathbb{N}_0$

3.2 Path Events and Split/Combine Pseudo-States

When modeling an operating system with statecharts, all external events can – due to the (code) locality principle – be categorized as $k \in \mathbb{N}$ fixed, finite event classes called branch categories (see Figure 4.8 in Section 4.2 for a universal partition for GPOS).

Definition 3.4 ***Branch Categories, Category Sets and Identifiers***
CAT *is the set of all branch category sets:* $CAT = \{CAT_0, \ldots, CAT_{k-1}\}$ *with* $\forall\, a, b \in \{0, \ldots, k-1\}, a \neq b : CAT_a \cap CAT_b = \emptyset$.

For each branch category κ, $0 \leq \kappa \leq k-1$, *the branch category set* $CAT_\kappa = \{i_{\kappa_0}, \ldots, i_{\kappa_{m-1}}\}$, $m \in \mathbb{N}$ *contains all possible values which the branch identifier* i_κ *of this category* κ *can take.*
Note that for each category κ, $i_\kappa \in CAT_\kappa$ *and* $i_\kappa \notin CAT_0 \cup \ldots CAT_{\kappa-1} \cup CAT_{\kappa+1} \cup \cdots \cup CAT_{k-1}$.

A branch category can be seen as a way-point in a location plan (e.g. a turnoff) and the branch identifiers are then instructions such as "turn left". A path (location plan) thus is an event with k indices (way-points), one for each branch category set defined. Each index κ represents the branch identifier i_κ of branch category CAT_{i_κ}, $0 \leq \kappa \leq k-1$. That is the information where to "turn".

Because of the code locality principle, events of one category are mainly handled at one dedicated state at a specific hierarchy level of the model. This fact is crucial for gaining any simplification.

Definition 3.5 ***Path Events*** $\overrightarrow{E}$
The set of path events $\overrightarrow{E}$ *is defined as a k-fold indexed set of events. Each index represents the branch identifier* i_κ *of branch category* CAT_{i_κ}, $0 \leq \kappa \leq k-1$.

$$e \in E : e_{i_0 \ldots i_{k-1}} \in \overrightarrow{E} \text{ where } k \in \mathbb{N}_0$$

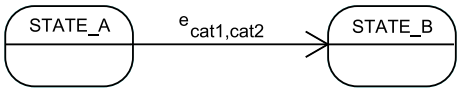


Figure 3.2: Path event

A path event is only written to that transition reacting on the "original" event $e \in E$. This event implicitly contains the information for $i_0, \ldots i_{k-1}$. The path event resulting from a signaling SCSI device $e_{INT,DEV,SCSI}$ with $k = 3$ is thus indexed with $i_0 = INT$ (an interrupt), $i_1 = DEV$ (caused by a peripheral device) and $i_2 = SCSI$ (source device is a SCSI device).

Definition 3.6 ***Path Conditions***
A path condition is a list of branch identifiers $(i_j, \ldots, i_{k-1})$ *where* $0 \leq j \leq k-1$. *PC is the set of path conditions.*

A path condition is initially set when a path event occurs.

Definition 3.7 ***Path Event Initialization Function*** $\mathfrak{pc}$
The path event initialization function $\mathfrak{pc} : \overrightarrow{E} \rightarrow PC$ *initially sets a path condition. The result of* $\mathfrak{pc}$ *is called initial path condition.*

The initial path condition for the above example is $\mathfrak{pc}(e_{INT,DEV,SCSI}) = (INT, DEV, SCSI) \in PC$.
Path conditions are defined within the topmost *XOR* superstate only. Within an *AND* state, i.e. when orthogonal components are used, each of these components has its own

path condition indexed pc_ω with ω being the identifier of the component the path condition relates to. Note that the index can be omitted when the component the path condition belongs to is unambiguous or irrelevant.

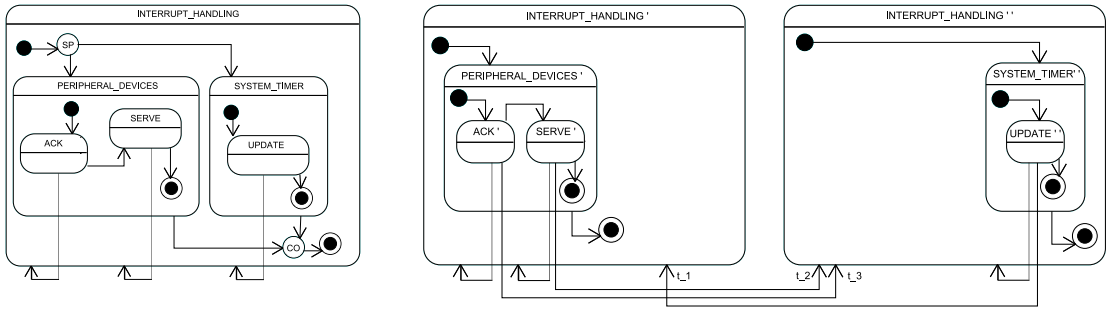
Path events are useful in allowing for branching according to the given categories. To achieve this, it is necessary to define means of splitting and combining paths. Each split processes a specific index of a path event and branches according to its value (which is like carrying out the given turn at the specific way-point). By use of such an extension, alternative paths can be drawn as part of their one parent state – instead of drawing disjoint states in conventional statecharts. The alternative paths are then merged again by a so-called combine on the same hierarchy level as the associated split.
The existing *XOR* connectors (see Definition 2.7) condition, junction and selection seem to be an ideal way of realizing such a mechanism at first glance. Unfortunately, they are not capable of managing flexible data structures such as lists. Hence, this mechanism cannot be used when it is exposed to recursion (cp. Section 3.4); recursion depths > 1 cannot be modeled. From this consideration it follows that a new type of constituent must be defined: the pseudo-state concept enriches the concept of *XOR* connectors with flexible data structure handling. In analogy to connectors, they do not need any system steps to be traversed. This is guaranteed by exploiting the "fast-forward problem" to our advantage. For ease of the later semantical transformation rules, pseudo-states are not part of the set of connectors at all.
Split and combine pseudo-states now manage correct branching and merging of the paths determined by path conditions.

Definition 3.8 ***Pseudo-States*** PS ***and Extended States*** $\tilde{S}$
SP and CO are the sets of named split and combine pseudo-states.

$$PS \stackrel{\text{def}}{=} SP \cup CO$$
$$\tilde{S} \stackrel{\text{def}}{=} S \cup PS$$

Each split and combine state pair (s_{sp}, s_{co}) is assigned to a category κ, $0 \leq \kappa \leq k-1$ with the branch category set CAT_κ[1]. It thus splits for branch identifier $i_\kappa \in CAT_\kappa$, i.e. for the value of condition i_κ set by the last path event occurred. Depending on the current value of i_κ, a corresponding outgoing transition is performed. From all transitions leading from a specific split state to normal system states, only one is ever carried out so that there is a specific system path to be taken.
To each split state, a specific combine state is assigned which unites all branches, i.e. all paths originating from its split state.
In our running example, we now look at the part of the system that carries out some dedicated action triggered by external stimuli. In Figure 3.3(a), it is represented by the root state INTERRUPT_HANDLING and its descendants. This state contains a branch (SP, CO) that corresponds to the second branch identifier (i_1 = DEV) of the path event $e_{INT,DEV,SCSI}$. Note that processing the first branch identifier is supposed to happen in a higher or earlier state not shown in this figure, the one according to the third is to happen later (see also Figure 3.7(a)).

[1]Note that the opposite is not necessarily true.



(a) with path events and split/combine

(b) without path events and split/combine

Figure 3.3: Comparison of path events and split/combine mechanism to a conventional statechart presentation

The shortcomings of the conventional statechart approach become clear with an example: Let's assume the system is in state SERVE due to a former processed event and a new event occurs implying transition to state UPDATE. Since no alternative paths within the same superstate can be traversed, the superstate INTERRUPT_HANDLING has to be cloned with disjoint descendants as shown in Figure 3.3(b). Thus, all possible transition permutations between the cloned states have to be modeled explicitly (t_1, t_2 and t_3). It is easy to see that for models with a larger number of alternative paths, this quickly leads to unmanageable models.

To give a transformation rule for path events as well as for split and combine states, extensions for E, L and T have to be first defined:

Definition 3.9 ***Extended Events*** $\tilde{E}$ ***and Extended Labels*** $\tilde{L}$
The set of extended events $\tilde{E}$ *contains the inductively defined statechart events and the path events.*

$$\tilde{E} \stackrel{\text{def}}{=} E \cup \overrightarrow{E}$$

The set of extended labels $\tilde{L}$ *is defined as*

$$\tilde{L} \stackrel{\text{def}}{=} \tilde{E} \times \tilde{A}$$

Definition 3.10 ***Full Extended Transitions*** $\hat{T}$***, Simplified Ext. Transitions*** $\tilde{T}$
The set of full extended transitions $\hat{T}$ *is analogous to the standard definition of* T *as in Definition 2.2.11 but uses the extensions on* S*,* L *and* H*:*

$$\hat{T} \subset (2^{\tilde{S}} \times \tilde{L} \times 2^{\tilde{S} \cup \tilde{H}})$$

with the set of extended history symbols $\tilde{H}$ *as in Definition 3.16.*
The set $\tilde{T}$ *of simplified extended transitions does not have sets of states as source and target, but only one state each:*

$$\tilde{T} \subset (\tilde{S} \times \tilde{L} \times (\tilde{S} \cup \tilde{H}))$$

As for the semantics, it holds:

$$\textbf{STRANS}\ \frac{\tilde{t} = (s_1, l, s_2) \in \tilde{T}}{\hat{t} = (\rho^*(s_1), l, \rho^*(\omega(s_2))) \in \hat{T}} \Downarrow \tag{3.1}$$

where $s_1, s_2 \in (\tilde{S} \cup \tilde{H}), l \in \tilde{L}$.

The definition of simplified extended transitions eases the transition syntax: only the topmost state – instead of a full set of nested states – is referenced as source or target respectively.

Now, the semantic transformation rule for a path event can be given:

$$\textbf{PATH EVENT}\ \frac{t = (s_1, e_{i_0,\ldots,i_{k-1}}/a, s_2) \in \tilde{T}}{\tilde{t} = (s_1, e/(a; a_{cs}), s_2) \in \tilde{T}} \Downarrow \tag{3.2}$$

where $s_1, s_2 \in \tilde{S}, a \in \tilde{A}, e \in E$ and

$$\tilde{A} \ni a_{cs} = \Big(push(i_0, cond_{CAT_0}); \ldots; push(i_{k-1}, cond_{CAT_{k-1}})\Big),$$

$cond_{CAT_0}, \ldots, cond_{CAT_{k-1}}$ are lists of statechart conditions (according to Definitions 2.2.7 and 3.2). Additionally, the initial path condition is set as soon as the path event occurs: $pc = \mathfrak{pc}(e_{i_0,\ldots,i_{k-1}})$.

The definition of the semantics implies that an *ESF* transition acting on a path event $e_{i_0,\ldots,i_{k-1}}$ implicitly reacts to its corresponding event $e \in E$ and does not only carry out action a but also an action a_{cs} setting the conditions by pushing them on the condition lists for each branch category.

To further specify the transitions between system states and pseudo-states, two important properties have to be defined: The degree of a split or combine state, i.e. the number of its incoming or its outgoing transitions and then the dimension of a path condition, i.e. the current number $k - j$ (cp. Definition 3.6) of its indices.

Definition 3.11 ***Dimension and Degree Functions*** $\mathfrak{dim}(e)$ ***and*** $\mathfrak{deg}(s)$

$$\begin{aligned}
&\mathfrak{dim} : PC \to \mathbb{N}_0, && \mathfrak{dim}(pc), pc \in PC \stackrel{\text{def}}{=} && k - j,\ 0 \le j \le k-1 \\
&\mathfrak{deg}^+ : \tilde{S} \to \mathbb{N}_0, && \mathfrak{deg}^+(s), s \in \tilde{S} \stackrel{\text{def}}{=} && |\ \{\ t = (in, l, out)\ |\ out = s\ \}\ | \\
&\mathfrak{deg}^- : \tilde{S} \to \mathbb{N}_0, && \mathfrak{deg}^-(s), s \in \tilde{S} \stackrel{\text{def}}{=} && |\ \{\ t = (in, l, out)\ |\ in = s\ \}\ |
\end{aligned}$$

where $l \in L, t \in \tilde{T}$ *and* $in, out \in \tilde{S}$. $\mathfrak{deg}^-(s)$ *denotes the incoming degree of a state,* $\mathfrak{deg}^+(s)$ *is called outgoing degree.*

The following constraints are postulated for the formalism. Those are also the modeling boundaries:

1. Exactly one transition leads to each split state and one originates from a combine state.

$$\forall\, sp \in SP : \mathfrak{deg}^{+}(sp) = 1 \tag{3.3}$$
$$\forall\, co \in CO : \mathfrak{deg}^{-}(co) = 1 \tag{3.4}$$

2. For each split state $sp \in SP$, exactly one (quantifier $\overset{1}{\exists}$) combine state $co \in CO$ is assigned (pairing).

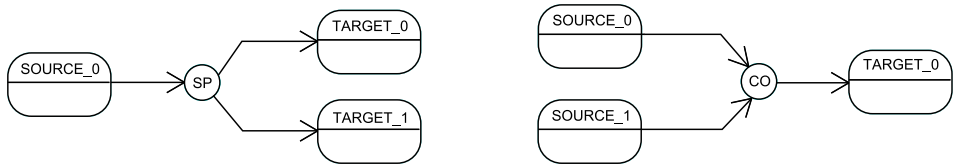
$$\forall\, sp \in SP : [\overset{1}{\exists}\, co \in CO : co \in \rho(\mathfrak{lca}(\{sp, co\}))] \tag{3.5}$$
$$\forall\, co \in CO : [\overset{1}{\exists}\, sp \in SP : sp \in \rho(\mathfrak{lca}(\{sp, co\}))] \tag{3.6}$$

3. For all assigned pairs ($sp \in SP, co \in CO$), the number of outgoing transitions of sp is equal to the number of incoming transitions of co, i.e. all paths originating from a split state are united by its corresponding combine state..

$$\forall\, (sp, co) \in SP \times CO :$$
$$[co \in \rho(\mathfrak{lca}(\{sp, co\})) \wedge sp \in \rho(\mathfrak{lca}(\{sp, co\})) \Rightarrow \mathfrak{deg}^{-}(sp) = \mathfrak{deg}^{+}(co)] \tag{3.7}$$

4. All transitions originating from a split or combine state have to react to the null event λ.

$$\forall\, sp \in SP : \qquad [\forall\, t = (sp, e/a, s) \in \tilde{T}, s \in \tilde{S}, a \in \tilde{A} : e = \lambda] \tag{3.8}$$
$$\forall\, co \in CO : \qquad [t = (co, \lambda/a, s) \in \tilde{T}, s \in \tilde{S}, a \in \tilde{A}] \tag{3.9}$$



(a) split state SP $\in SP$ with $\mathfrak{deg}^{+}(sp) = 1$, $\mathfrak{deg}^{-}(sp) = 2$

(b) combine state CO $\in CO$ with $\mathfrak{deg}^{+}(co) = 2$, $\mathfrak{deg}^{-}(co) = 1$

Figure 3.4: Split and combine pseudo-states and their $\mathfrak{deg}$ values in the *ESF*

Furthermore, they are two functions dealing with the assignment of branch category sets and branch identifiers to pseudo-states or system states:

Definition 3.12 ***Branch Category Assignment Function*** $\mathfrak{cat}$
The branch category assignment function $\mathfrak{cat}$ *assigns a branch category set to each pseudo-state:*

$$\mathfrak{cat} : PS \rightarrow CAT$$

The function $\mathfrak{cat}$ yields the branch category (set) for which a pseudo-state is "responsible".

Definition 3.13 ***Branch Identifier Assignment Function*** $\mathfrak{bid}$
The branch identifier assignment function $\mathfrak{bid}$ *assigns a specific value for the branch identifier* $i_t \in CAT_t$ *to a state:*

$$\mathfrak{bid} : \tilde{S} \rightarrow \underset{t}{CAT},\ 0 \leq t \leq k-1$$

By default, this assignment is carried out by the matching of identifiers (operator $\overset{\text{id}}{=}$*):*

$$\mathfrak{bid}(s) = i_{t_b} \Leftrightarrow i_{t_b} \overset{\text{id}}{=} s$$

The function $\mathfrak{bid}$ is used to determine for any (pseudo- or system) state $s \in \tilde{S}$ that is the target of a transition originating directly from a split state the condition under which the transition can occur. That means that it determines the value the branch identifier i_κ of the branch category set CAT_κ assigned to the precedent split state has to take to activate the transition leading to state s.

We now have all the necessary definitions to give the semantics for the pseudo states:

$$\textbf{SPLIT STATE}\ \frac{t_i = (s_1, e_i/a_i, sp) \in \tilde{T}; sp \in SP; t_o = (sp, \lambda/a_o, s_2) \in \tilde{T}}{\tilde{t} = (s_1, e_i[c]/(a_i; a_o), s_2) \in \tilde{T}} \Downarrow \tag{3.10}$$

where $s_1, s_2 \in \tilde{S}$ and $a_i, a_o \in \tilde{A}, e_i \in E, c = get(cond_{\mathfrak{cat}(sp)} \overset{\text{id}}{=} \mathfrak{bid}(s_2))$.
At each split state, the path condition is modified:

$$pc = \left(get(cond_{CAT_{\mathfrak{cat}(sp)+1}}), \ldots, get(cond_{CAT_{k-1}})\right)$$

This leads to an adjustment in the dimension: $\mathfrak{dim}(pc)$ is decremented. The new path condition does not contain the branch identifier for the previously taken branch any more. Note that this definition of the path condition is based only on the path condition lists, not on the path condition that was valid before the split. This is very important when it comes to interrupting unfinished processes that are later resumed using long term history (see Section 3.4): it has to be possible at any time to specify the currently active path condition even if the system jumps into any state within the system using history connectors that skip previous default and split states.

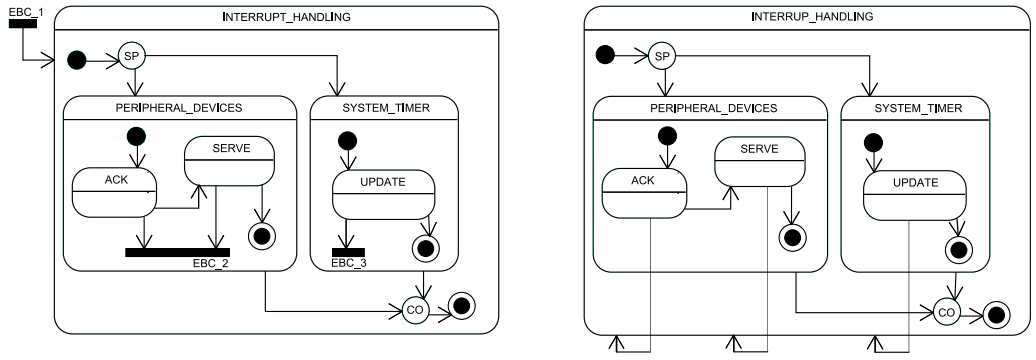
$$\textbf{COMBINE STATE}\ \frac{t_i = (s_1, e_i/a_i, co) \in \tilde{T}; co \in CO; t_o = (co, \lambda/a_o, s_2) \in \tilde{T}}{\tilde{t} = (s_1, e_i/(a_i; a_o; tail(cond_{CAT_{\mathfrak{cat}(co)}})), s_2) \in \tilde{T}} \Downarrow \tag{3.11}$$

where $s_1, s_2 \in \tilde{S}$ and $a_i, a_o \in \tilde{A}, e_i \in E$.
The topmost value of the condition list of $CAT_{\mathfrak{cat}(sp)}$ (which is by definition equal to $CAT_{\mathfrak{cat}(co)}$ for the assigned co) is thus excluded from the path condition at the split state but not actually removed from the list until the respective combine state is reached and the action executed.

3.3 Event Bus

The event bus constituent implements the same paradigm that we presented before with the split and combine mechanism: a shift from explicitly modeled to implicitly given.

The introduction of both path events and split and combine states trims the numbers of inter-level transitions considerably. To additionally reduce the number of explicitly given transitions in the models, a routing mechanism has to be established. Such a simplified way of drawing transitions is provided by a new element named an *event bus*. In a way, it "teleports" transitions to their actual destination states.
There are two possible variations of this element: The first would be a heavy-weight event bus (HWEB) that is a routing mechanism taking a transition labeled with a path event $e_{i_j \ldots i_{k-1}}$ and heading to an event bus pseudo-state (EBS). The HWEB mechanism then continues this transition to that destination state which contains the split state $sp \in SP$ branching for branch category j as a child state. For this, it uses a target function which assigns a target state to each correctly indexed path event.
One of the drawbacks of this approach is that when it comes to cartesian transition sets (cp. Section 3.5), the definition of this mapping towards a valid target would be difficult and ambiguous. Furthermore, the extension of pseudo-states with the EBS complicates the transition rule definitions considerably. Finally, only transitions labeled with a path event could be processed by the HWEB mechanism; those with conventional events cannot.
On the other hand, the light-weight event bus (LWEB) defines a simplified way of drawing explicitly given transitions. Contrary to the HWEB case, the transition leading to an LWEB can also be labeled with a standard event $e \in E$. The LWEB is fully transparent, i.e. the target of the transition taken is the explicitly given target system state. Regarding its semantics and adaptability, it has strong advantages compared with the HWEB. Therefore in the following, only the LWEB (or simply event bus) will be specified and used. An event bus is depicted in the style of ground symbols used in CAD systems for electronic circuits. The graphic entities (black bars in Figure 3.5) are called event bus connectors (EBCs). The set of event bus connectors is named EBC.



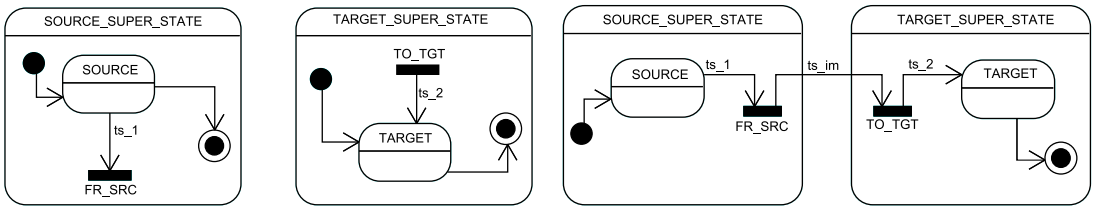
(a) with event bus (b) without event bus

Figure 3.5: Comparison of the event bus facility with a conventional statechart presentation

A completely defined transition $t \in \tilde{T} = (\mathsf{SOURCE}, e/a, \mathsf{TARGET})$ (see figure Figure 3.6) with $e \in \tilde{E}$ has to be given to use the LWEB. The entire transition t is a full transition (Definition 2.9). Since EBCs are connectors, we define:

Definition 3.14 ***Extended Connectors*** $\widetilde{CON}$
$\widetilde{CON}$ *is the set of extended connectors with* $\widetilde{CON} = EBC \cup CON$.

In Figure 3.6(a), there exist two dedicated event bus connectors FR_SRC, TO_TGT $\in EBC$ for t. They break the full transition t down into two transition segments ts_1 and ts_2 $\in TS$. The LWEB now implicity defines a further transition segment ts_im to complete the full transition t as shown in Figure 3.6(b).



(a) event bus connectors FR_SRC and TO_TGT for $t \in T = (\mathsf{SOURCE}, label, \mathsf{TARGET})$

(b) full transition completed by implicitly given segment ts_im. The full transition is also a FCT.

Figure 3.6: Compound transition constructed by event bus connectors

Formally, the set of transition segments is defined as follows:

Definition 3.15 ***Extended Transition Segments*** $\widetilde{TS}$

$$\widetilde{TS} \subset \left((\tilde{S} \cup \widetilde{CON}) \times \tilde{L} \times (\tilde{S} \cup \tilde{H} \cup CON)\right)$$

To limit the semantic side effects, it is claimed that all transition segments leading from an EBC to a target state contain only a trivial label:

$$\forall\, t \in \tilde{T} : t = (ebc, \lambda/\mu, s_1),\ ebc \in EBC, s_1 \in \tilde{S}$$

Using the EBC routing mechanism to implicitly draw a transition must not under any circumstances alter the events, conditions and actions associated with the transition. From that, the semantics of the LWEB can now be derived.

The transformation rule contains two steps:

$$\textbf{EB1}\ \frac{\begin{array}{c} t_{ini} = (s_1, e_{ini}/a_{ini}, s_{ebc\_s}) \in \tilde{T}; s_{ebc\_s} \in EBC; \\ s_{ebc\_t} \in EBC; t_{cont} = (s_{ebc\_t}, \lambda/\mu, s_1) \in \tilde{T} \end{array}}{\begin{array}{c} \tilde{t}_{ini} = (s_1, e_{ini}/a_{ini}, s_{ebc\_s}) \in \tilde{T}; \tilde{t}_{impl} = (s_{ebc\_s}, \lambda/\mu, s_{ebc\_t}) \in \tilde{T}; \\ \tilde{t}_{cont} = (s_{ebc\_t}, \lambda/\mu, s_1) \in \tilde{T} \end{array}} \Downarrow \tag{3.12}$$

$$\textbf{EB2}\ \frac{\begin{array}{c} \tilde{t}_{ini} = (s_1, e_{ini}/a_{ini}, s_{ebc\_s}) \in \tilde{T}; \tilde{t}_{impl} = (s_{ebc\_s}, \lambda/\mu, s_{ebc\_t}) \in \tilde{T}; \\ \tilde{t}_{cont} = (s_{ebc\_t}, \lambda/\mu, s_1) \in \tilde{T} \end{array}}{\tilde{t} = (s_1, e_{ini}/a_{ini}, s_2) \in \tilde{T}} \Downarrow \tag{3.13}$$

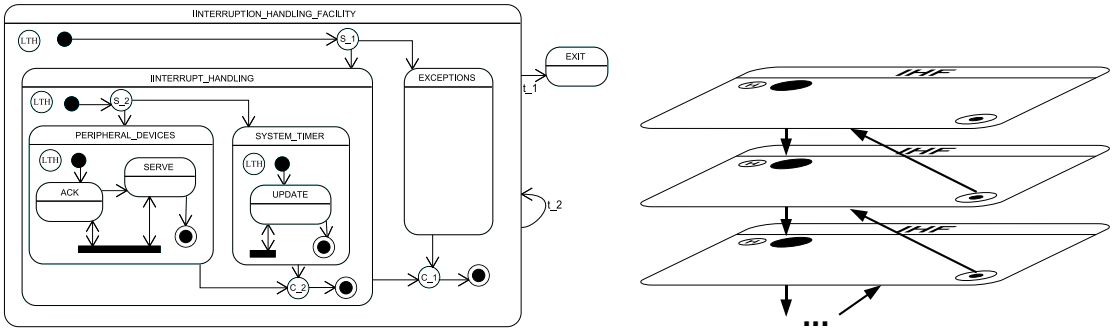
where $e_{ini} \in \tilde{E}$, $a_{ini} \in \tilde{A}$, $s_1, s_2 \in \tilde{S}$.
EB1 defines the enhancement with a further continuation TS between the EB connectors

and *EB2* the translation of this completed connector structure into a full transition. Note that according to *EB1*, step *EB2* would not be necessary if the EBC were replaced by a standard connector, e.g. a condition connector with a true-condition or a junction connector. Furthermore, in the case of the two connectors (FR_SRC and TO_TGT) being identical, the full transition is a trivial special case of the transition rule *EB1*.

3.4 Long-Term History Connectors

The mere reduction of the number of explicitly defined transitions is not sufficient to create a modeling technique for large and complex real-world systems. A satisfying way to represent recursion is also required to ease modeling efforts of systems consisting of numerous recursive parts. To achieve this, a modification of the history connector named long-term history connector (LTH) is created: it facilitates the tracking of a number of previously accessed substates of a specific state.

The conventional (flat or deep) history connectors cannot provide this: they do not track the overall sequence of previously traversed states but only the last visited one. Especially when applying the mechanisms of path events and event-busses, this is not at all sufficient. Whereas deep history connectors provide structural, i.e. hierarchical depth, the long-term history connector now offers temporal depth, i.e. tracking of a longer past. Both concepts are orthogonal.



(a) with long-term history (b) without long-term history

Figure 3.7: Comparison of long-term history with a conventional statechart presentation

As shown in Figure 3.7(b), it would be necessary to clone the recursive part of the model – here the state INTERRUPTION_HANDLING_FACILITY – where each clone uses conventional history. This is not only inefficient, but also impedes variable recursion depths.

Now, an extended set of history symbols $\tilde{H}$ is defined. It contains H as well as the elements of a third type of history symbols, the long-term history symbols (elements of the set LTH). In the original definition of H, there is no function to determine the type of a history symbol (deep or flat). Consequently, a history type function $\mathfrak{bit} : \tilde{H} \rightarrow \{H, LTH\}$ has to be defined. It denotes whether a history connector is an ordinary (either deep or flat) or a long term one. Both sets are disjoint.

Definition 3.16 ***Extended History Symbols*** $\tilde{H}$***, History Type Function*** $\mathfrak{hit}$
LTH is defined as the set of long-term history connectors (or symbols). The set of extended history symbols is given as $\tilde{H} \stackrel{\text{def}}{=} H \cup LTH$ *with* $H \cap LTH = \emptyset$.
The history type function $\mathfrak{hit} : \tilde{H} \to \{H, LTH\}$ *denotes the type of a history symbol:*

$$\mathfrak{hit}(h) \stackrel{\text{def}}{=} \begin{cases} H & | \; h \in H \\ LTH & | \; h \in LTH \end{cases}$$

To allow for recursion, two premises must hold:

1. The recursion depth has to be bounded by a natural number k. We refer to that property as k-boundedness.
2. The length of a path, i.e. the number of states processed in a single recursion step, has to be bounded by a natural number l. We refer to that property as l-boundedness.

These boundedness properties guarantee that a recursion is always terminated. With that precondition, a history function $\tilde{\tau}$ for long-term history connectors can be defined in analogy to Definition 2.2.5. It yields an m-tuple of substates of the very state s containing the long-term history connector: the last m direct substates of s that were accessed in the past. The value m is increased when entering a state s containing a long-term history connector and decreased when leaving s.

Definition 3.17 ***History Function*** $\tilde{\tau}$ ***for Long-Term History Connectors***
For a given sequence of system configurations $X = (X_0, \ldots, X_n)$ *and a history symbol* $h \in \tilde{H}$, $\exists\, s \in S \; : \; \gamma(h) = s$ *and* $\mathfrak{hit}(h) = LTH$, *the history function* $\tilde{\tau} : H \times (2^S)^n \to S^m$ *yields the sequence of the last* m *accessed substates of a state* s *with* $m \leq n$ *and* $m, n \in \mathbb{N}_0$.
It is defined using S', *the set of all substates visited in sequence* X*:*

$$\begin{aligned} \tilde{I} &= \{i | \rho^*(s) \cap X_i \neq \emptyset\} \\ S' &= \{s'_i | i \in \tilde{I}, s'_i \in \rho(s), \rho^*(s'_i) \cap X_i \neq \emptyset\} \end{aligned}$$

$$\tilde{\tau}(h, X) \stackrel{\text{def}}{=} \begin{cases} \delta(s) & | \textit{if } \tilde{I} = \emptyset \\ (s_{i_j}, \ldots, s_{i_{j+m-1}}), s_{i_k} \in S', & \\ j \leq k \leq j + m - 1, \forall\, s_{i_k}, s_{i_{k+1}} : i_k > i_{k+1} & | \textit{ otherwise} \end{cases}$$

The list $(s_{i_j}, \ldots, s_{i_{j+m-1}})$ contains the m most recently visited substates of s with indices $i_j, \ldots, i_{j+m-1}$ representing the m biggest elements of $\tilde{I}$ in descending order. Formally: $i_j = max(\tilde{I}), i_{j+n} = max(\tilde{I} \setminus \{i_j, \ldots, i_{j+n-1}\})$ where $1 \leq n \leq m$.

The maximum length of sequence S' is limited by the k-boundedness property of *ESF* in the case of a recursive ascent and by the general finiteness of statecharts otherwise. The integer m is always bound to a specific state s. In the following, m_s is the number of tracked history states for a state s.
By means of this modified history function, the transformation rules for the LTH guarantee

that the value m_s for each state s is handled correctly within the model. Thus, the transformation rules for handling the number of tracked states are defined as follows:

$$\textbf{LTH IN}\ \frac{t = (s_1, e/a, s) \in \tilde{T} \text{ where } \exists\, lth \in LTH : \gamma(lth) = s}{\tilde{t} = (s_1, e/(a; m_s : m_s + 1), s)} \Downarrow \tag{3.14}$$

$$\textbf{LTH OUT}\ \frac{t = (s, e/a, s_2) \in \tilde{T} \text{ where } \exists\, lth \in LTH : \gamma(lth) = s}{\tilde{t} = (s, e/(a; m_s : m_s - 1), s_2)} \Downarrow \tag{3.15}$$

where $s_1, s_2 \in S,\ a \in \tilde{A},\ m_s \in V,\ e \in E$.
Recursion can now be modeled using long-term history connectors: Each state containing an LTH keeps track of its m_s previously visited substates. To make recursion applicable, it has to be postulated that there are so-called recursion scopes (RS), i.e. states with nested substates, where all non-basic states contain a long-term history connector. With that prerequisite, it is possible to leave a state at any time, process another path (recursive descent) and later return to the state(s) the system was in before (recursive ascent).
To terminate such a recursion, the uppermost state of a recursion scope, called the recursion delimiter state, must be known. When it reaches its exit state, it must be checked whether there is still any path awaiting completion, i.e. if the long-term history still contains one or more states. If this is the case, the traversal of the most recently accessed substate has to be resumed. Otherwise, the state can be left since the recursive ascent is finished. In Figure 3.7(a), the state INTERRUPTION_HANDLING_FACILITY is the recursion delimiter state of the drawn scope.

Definition 3.18 ***Recursion Delimiter States*** *RDS*
RDS is the set of recursion delimiter states with

$$\begin{aligned} RDS = \quad & \{s \in S \,|\, \exists h \in \tilde{H}, s^* \in S : [(\gamma(h) = s \wedge \text{hit}(h) = LTH) \wedge (\rho(s^*) = s \wedge \\ & (\forall\, h^* \in \tilde{H} : \gamma(h^*) = s^* \Rightarrow \text{hit}(h^*) = H))]\} \end{aligned}$$

For states belonging to the set of recursion delimiter states RDS, a transformation rule additional to *LTH OUT* is defined:

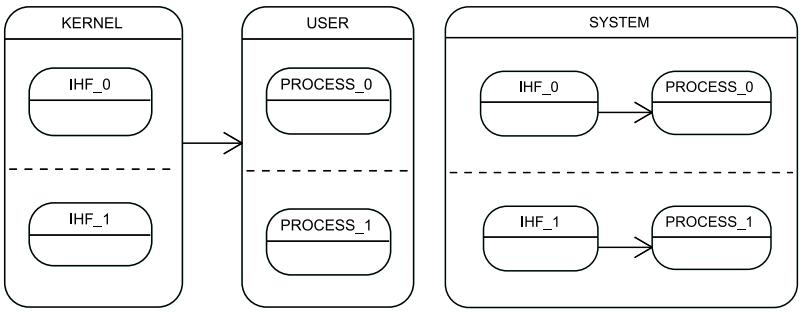
$$\textbf{LTH OUT RDS}\ \frac{t = (s, e/a, s_1) \text{ with } s \in RDS}{\tilde{t}_1 = (s, e[m_s == 0]/a, s_1); \tilde{t}_2 = (s, e[m_s > 0]/a, s)} \Downarrow \tag{3.16}$$

This rule specifies that any transition originating from a recursion delimiter state is taken only if the corresponding m_s is zero (transition t_1 in Figure 3.7(a)). Otherwise (t_2 in Figure 3.7(a)), the RDS is entered anew to proceed with the next "suspended" state.

3.5 Cartesian Transition Set

There is further optimization potential when considering n-time parallel systems. The uppermost hierarchy layer must here be described by an n-time *AND* superstate. When some of the substates of these *AND* states have to be logically grouped because of common functionality or transition groups, this is not possible with conventional statecharts. The existing orthogonal approaches are insufficient: either a fully grouped approach where it

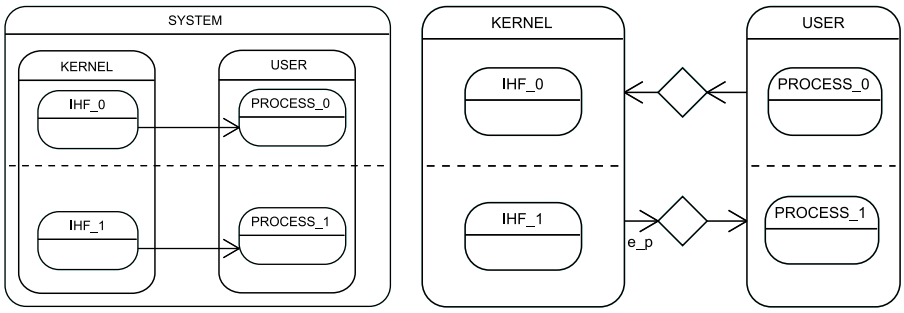
is impossible to be in a substate of one *AND* state and in a substate of another *AND* state at a time (Figure 3.8(a)), or a parallel approach where states and transitions are to be duplicated and grouping is completely lost (Figure 3.8(b)).



(a) groups (b) parallel

Figure 3.8: Comparison of approaches grouping vs. parallelism

Figure 3.9(a) shows a possible solution to this problem by allowing states to overlap. Harel and Kahana [HK92] show clearly that general overlapping in statecharts leads to much more complex semantics and creates a number of new problems. The most severe one is that after creating fully or partially overlapping states, some parts of the statechart semantics become unclear and fuzzy. For example, it is ambiguous whether the system is in all overlapping superstates, or only in a subset of them or what it means being in parts of a state only. Furthermore, it is shown that enriching the statechart formalism with overlapping does not gain any expressiveness that could not be gained in other ways.



(a) grouped overlapped (b) with CTS - parallel grouped

Figure 3.9: Comparison of approaches to deal with parallelism

Fortunately, general overlapping is not necessary to confront the problem: a mechanism operating on path events analogous to the split-combine mechanism is now constructed to address the tension between parallelism and grouping. The main idea is to provide a connector (called cartesian transition set connector or CTSC) connecting two *AND* states (Figure 3.9(b)). Now, the substate of such a state can be addressed directly as the target of a transition occurring on a path event.
The target state is then an implicitly given *AND* state constructed of the substates of the source state that are *not* left and the specified *new* substate. In our example (Figure

3.9(b)) with two parallel CPUs, the given path event e_p facilitates the transition to an implicit, merged *AND* state with substate PROCESS_1 for CPU 1 together with the unaffected substate IHF_0 of CPU 0.

Following this principle, the set of CTS connectors is defined as follows:

Definition 3.19 ***Cartesian Transition Set Connectors*** $CTSC$**,** ***CTS Assignment Function*** $\mathfrak{ctf}$
The set of cartesian transition set connectors is named $CTSC$*. The set of connectors is redefined as follows:*

$$\tilde{CON} \stackrel{\text{def}}{=} CTSC \cup EBC \cup CON$$

The CTS assignment function $\mathfrak{ctf} : CTSC \rightarrow 2^T$ *yields the transitions associated with a CTS connector.*

Each CTSC has a direction, i.e. it leads via transition segments from a set $\bar{S} \subset S$ to another set $\hat{S} \subset S$, and we say that *the CTSC leads from* $\bar{s} \in \bar{S}$ *to* $\hat{s} \in \hat{S}$. For *AND* states $s_{src}, s_{tgt} \in S$ and $\nu, \mu, n_{src}, n_{tgt} \in \mathbb{N}$, the following holds:

$$SRC = \{s_{src_\nu} |\, 0 < \nu \leq n_{src}\} \subset S \quad \text{where } \rho(s_{src}) = SRC, \psi(s_{src}) = AND \tag{3.17}$$

$$TGT = \{s_{tgt_\mu} |\, 0 < \mu \leq n_{tgt}\} \subset S \quad \text{where } \rho(s_{tgt}) = TGT, \psi(s_{tgt}) = AND \tag{3.18}$$

The following modeling rules are given for CTSC:

1. *AND* superstates which – or whose substates – are connected using a CTSC must have the same number of parallel substates: $n_{src} = n_{tgt}$

2. A transition segment leading to a CTSC has to be labeled with a path event.

$$ts = (s_1, e_{i_0 \ldots i_{k-1}}/a, c) \in \widetilde{TS} \tag{3.19}$$

   where $c \in CTSC, e_{i_0 \ldots i_{k-1}} \in \overrightarrow{E}, s_1 \in S$.

3. A transition segment leading from a CTSC to a state has to be labeled with the trivial label λ/μ.

$$ts = (c, \lambda/\mu, s_2) \in \widetilde{TS} \tag{3.20}$$

   where $c \in CTSC$, $s_2 \in S$.

4. The first index of the path event on a TS to a CTSC has to specify the target substate (of the *AND* state) to which the CTSC leads.

$$\forall\, ts_1, ts_2 \in \widetilde{TS} \; : [ts_1 = (s_1, e_{i_0 \ldots i_{k-1}}/a, c) \wedge ts_2 = (c, \lambda/\mu, s_2) \Rightarrow i_0 \stackrel{\text{id}}{=} s_2] \tag{3.21}$$

5. For each CTSC leading from an *AND* state s_1 to another *AND* state s_2, there has to be a corresponding CTSC directed from s_2 to s_1.

$$\forall\, s_1, s_2 \in S, t_1 \in T, c_1 \in CTSC : \Big[t_1 = (s_1, e_{i_0 \ldots i_{k-1}}/a, s_2) \wedge t_1 \in \mathfrak{ctf}(c_1) \Rightarrow$$
$$\big[\exists\, t_2 \in T, c_2 \in CTSC : [t_2 = (s_2, e_{i_0 \ldots i_{k-1}}/a, s_1) \wedge t_2 \in \mathfrak{ctf}(c_2)]\big]\Big] \tag{3.22}$$

   where $e_{i_0 \ldots i_{k-1}} \in \overrightarrow{E},\ a \in \tilde{A}$.

Note that by means of the CTSC mechanism as defined, only one substate can be addressed per path event. This is intentional and could be changed by allowing more than one category to be processed by a CTS connector.

The transformation rule is given as follows:

$$\textbf{CTSC}\ \frac{\begin{array}{c} t = (s_s \in \rho^*(s_{src}), e_{i_0 \ldots i_{k-1}}/a, s_t \in \rho^*(s_{tgt})); \\ c \in CTSC : t \in \mathfrak{ctf}(c); s_{tgt} \in S \end{array}}{\begin{array}{c} \tilde{s}_{tgt}; \rho(\tilde{s}_{tgt}) = \{s_{src_0}, \ldots s_{src_{\nu-1}}, s_{tgt_\nu}, s_{src_{\nu+1}}, \ldots, s_{src_n}\}; \\ \tilde{t} = (s_s, e_{i_0 \ldots i_{k-1}}/a, s_t) : s_t \in \rho^*(\tilde{s}_{tgt}) \end{array}} \Downarrow \tag{3.23}$$

where $s_{tgt_\nu} \overset{\text{id}}{=} i_0$, *AND* states $s_{src}, s_{tgt}, \tilde{s}_{tgt} \in S$ with $\rho(s_{src}) = \{s_{src_0}, \ldots s_{src_n}\}$, $s_{tgt_\nu} \in \rho(s_{tgt})$, $a \in \tilde{A}$, $e_{i_0 \ldots i_{k-1}} \in \overrightarrow{E}$, $s_s, s_t \in S$, $t, \tilde{t} \in T$.
Because of this rule, it is not necessary to implicitly create the full cartesian product of imaginable new *AND* states: only those new *AND* states are created that are reachable due to a matching transition marked with the respective path event.

Analogous to the split state semantics, the path condition is modified when passing a CTSC as well since the CTSC removes the first condition:

$$pc = \big(get(cond_{CAT_1}), \ldots, get(cond_{CAT_{\kappa-1}})\big)$$

This also decrements the value of $\mathfrak{dim}(pc)$.

3.6 Conclusion

By definition of the *ESF* constituents, all shortcomings and missing features of Harel's statecharts have been addressed. Path events and path conditions allow for implicit specification of the part of the model where certain events are handled. Split and combine pseudo-states provide the formal foundation for the processing of path conditions in a hierarchical model providing a branching mechanism. The event bus mechanism reduces the number of transitions in the models. The problems occurring when recursive behavior is modeled is solved by the definition of long-term history connectors. Finally, the modeling of concurrent paths is simplified considerably by way of cartesian transition set connectors. The syntax and prerequisites of these new *ESF* elements has been given and transformation rules have been defined that allow for the step-by-step translation of *ESF* constituents into conventional statechart elements. These also provide the semantical basis necessary for the later formal verification (cp. Section 6.3).

Chapter 4

IHF Models

While the developed *ESF* suits all OS modeling targets, this thesis addresses specific questions about soft real-time as outlined in Section 1.2. For that purpose, we shift our focus now towards the interrupt handling facility. The IHF is the part of an operating system that represents nondeterminism. Furthermore, it is the most reactive part of a complete OS. As elaborated later in chapter 5, the architectural and qualitative properties of the interrupt handling facility greatly influence a system's ability to meet (soft) real-time requirements.

4.1 Hardware Platform

The first step is to discuss possible hardware platforms that can be used as a basis for the investigation of different operating systems. Any interruption is handled first by the microprocessor and its peripheral circuits.

Modern microprocessors differ widely according to their fields of application, Flik [FL01] gives a detailed introduction to microprocessors.
General purpose systems such as personal computers usually rely on complex instruction set computing (CISC) processors. Their clock rate is between 1 and 4 GHz. Instructions are pipelined and executed out of order. Multiple arithmetical logical units (ALU) and sophisticated pipelines allow for super-scalar execution. Data and address bus widths are usually either 32 or 64 bit.
The most common CISC architecture is the Intel Architecture (IA), starting with the 8086 CPU in 1978. The CPU family most commonly used today is IA32. Its representatives are mainly Intel Pentium CPUs, most recently the Pentium 4 (2000 - 2006). The IA32E – Intel Pentium 4 Extreme Edition (2005-2007) – is the first real 64 bit architecture emerging from the Pentium family. The Intel Core Duo and Intel Core Solo Processors (2006 - 2007) as well as the Intel Core 2 Processor family (2006-2008) are the currently used CPUs.
High end servers and workstations usually use reduced instruction set computing (RISC) processors. The reduced instruction set is implemented very efficiently and fast, the slower clock-rates between 1 and 3 GHz are compensated for by fewer cycles per special instruction. Whereas the CISC CPUs of any vendor are nearly all similar to the Intel Architecture, different vendors of RISC processors implement entirely different features in their

CPUs. The current SPARC processors such as UltraSPARC IIIcu, UltraSPARC IV and UltraSPARC IV+ are also used in workstations, whereas the T1 Niagara processors are optimized for server side throughput computing. Other examples are the 32 bit MIPS CPUs (MIPS32 (4k)) such as R3000 and the modern 64 bit MIPS64 (5k) such as the R6000A.

Intel Architecture
The Intel Architecture (x86) features a three-way super-scalar design equipped with a 12-stage pipeline [Int08a]. Furthermore, a 4-way micro-operations cache as well as a 8-way level 2 cache are on die. Starting with the Intel Pentium Pro, the IA is capable of dynamic out-of-order-execution and intelligent branch prediction. Since the Intel Pentium 4 processor, the Intel NetBurst Architecture that allows for hyper pipelining and advanced dynamic execution has been implemented. Intel Virtualization Technology was then also introduced. Dual Core capability was added with the Extreme Edition. Thermal management and dynamic power coordination are the latest accomplishments of Intel research.
Since the release of the Extreme Edition 840 (2005), several cores can be located on every physical carrier (package) – such a system is called multi-core. However, every core (i.e. an entire CPU) possesses up to two logical CPUs, or more precisely architectural states (each with its own set of registers and APIC, see Section 4.1.1) sharing busses and the ALU – such a system is called hyper-threaded. Today both paradigms are used: current Intel processors are hyper-threaded multi-core CPUs.
Disruptions of the control flow are subdivided into two basic categories: exceptions and interrupts [Int08b]. Exceptions cannot be masked under any circumstances whereas interrupts can be blanked out according to the operating system. Exceptions can be raised due to program errors detected by the CPU itself or due to machine check results. Interrupts can be generated by external hardware or issued by means of software instructions.

SPARC V9 Architecture
Sun Microsystems currently manufactures the UltraSPARC IV+(2006) and the UltraSPARC T1 (2007) CPU [SUN05]. Both implement the SPARC V9 Architecture [WG00]. The UltraSPARC IV+ features a four-way super-scalar design equipped with a 14-stage pipeline. Its instruction set is compatible with the visual instruction set (VIS) Version II. Its virtual address space has a width of 64 bits, the physical one is 43 bits (8192 GB of memory can be accessed). The CPU embodies full 64 bit arithmetical and logical operations. The specifications are inherited mainly from the UltraSPARC IV CPU [SUN04b] and the UltraSPARC III cu [SUN04a].
The UltraSPARC T1 CPU can execute up to 32 threads in parallel. Note that not all necessary components like the floating point unit are duplicated – which would limit the fields of application for this CPU considerably. All SPARC V9 processors are capable of symmetrical multiprocessoring. Besides these features, all CPUs have architectural support for external monitoring by means of watch-dog cards such as RSC by Sun Microsystems or XSCF by Fujitsu Siemens Computers.
Because of these features, the handling of interruptions within a SPARC V9 processor is complex. Internal conditions that lead to an altered control flow are called exceptions, external stimuli (e.g. by other CPUs or I/O devices) are interrupts. The system's reaction to either of them is defined by system traps. A trap is treated like any asynchronous

procedure call. If the CPU has reached the maximal nesting depth for interruptions, the system switches first to an "alert state" that can only be left by immediately starting the recursive ascent. If another trap occurs, the system switches to an "error state" where only watchdog resets or hardware resets are valid inputs. This mechanism of failsafe handlers allows the external environment to interact at any time and in whatever condition the system is.

MIPS
MIPS64 (2005) is the latest set of 64 bit CPUs fabricated by MIPS. They feature a four-way super-scalar design equipped with a multiple stage pipeline [MIP05a]. The MIPS64 5Kc CPU possesses six pipeline stages, the MIPS64 20Kc seven. The instruction set implements the MIPS instruction set architecture (ISA) version V [MIP05b]. The so-called dual-issue super-scalar micro-architecture is capable of executing pairs of instructions together (SIMD), its pre-fetch unit fetches four instructions per cycle. A MIPS CPU has a 40-bit virtual address space and a 36-bit physical address space (so it addresses up to 64 GB of physical memory).
All interruption handling [MIP05c] is carried out by the first co-processor that is integrated into the core of a MIPS CPU. The MIPS64 CPUs distinguish between interrupts and exceptions as the Intel Architecture and SPARC V9 family do. While the first version of the architecture supported merely six hardware interrupts, the second version allows for three modi: a compatibility mode that is identical to version 1, a vectored interrupt mode that allows the mapping of dedicated handlers to interrupt vectors and finally a mode that delegates all prioritizing and handling of interrupts to an external interrupt controller circuit. Regardless of which modus is chosen, a handler routine is not interruptible and thus its length is restricted to 32 machine instructions [MIP05c]. Any handling beyond that has to be realized by means of routine calls and is thus not reentrant at all – preemption cannot be prohibited.

Table 4.1 sums up these IHF properties and the adequacy for real-time usage of the platform. Furthermore, practical criteria such as usage and costs of the platform are assessed as well. Finally, the availability of operating systems for each architecture is considered.

Hardware	IHF Properties	Adequacy for Real-Time	Usage, Costs	OS Available
SPARC	complex, powerful	most eligible	common, very costly	some
MIPS	basic, limited	concept-dependent	uncommon, costly	few
IA32	comprehensive, flexible	adequate	widespread, reasonable	numerous

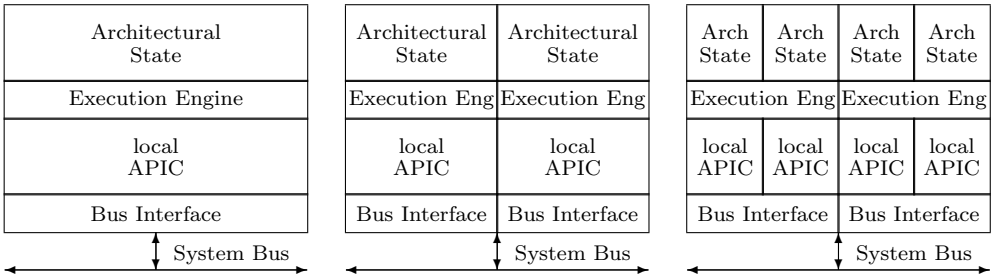
Table 4.1: Hardware comparison

Obviously, after immediately ruling out the MIPS platform, the choice between Intel (CISC) and SPARC (RISC) is not only practical but also conceptual. The Intel Architec-

ture is the platform chosen for this thesis: due to the flexibility of the IHF, the system's real-time capabilities depend on the supervisor software rather than on the hardware.

4.1.1 Intel Architecture Specific Details

In this thesis the focus is on multi-processor architectures rather than on single CPU systems. As mentioned above, a system's symmetrical multiprocessor (SMP) [Int97] capability can be achieved in different ways. In order to have n-fold SMP capability, either n single core CPUs (see Figure 4.1(a)) , $\frac{n}{2}$ dual core (Figure 4.1(b)) or hyper-threading[1] CPUs or even $\frac{n}{4}$ dual core CPUs capable of hyper-threading (Figure 4.1(c)) can be used. Regardless of this usage, any architectural state (AS) of a single or dual core processor as well as any AS of a hyper-threading CPU has its own local advanced programmable interrupt controller [Int08b] as depicted in Figure 4.1.



(a) Intel Processor (b) Intel Dual Core Processor (c) Intel Dual Core Processor with Hyper-threading

Figure 4.1: Overview of Intel single core, hyper-threading and dual core design

Any CPU implementing the specifications of the Intel Architecture provides four different current privilege levels (`CPL`) for the currently running program.

Interrupts and Exceptions

The two distinct types of interruptions that are defined by the Intel Architecture [Int08b] can be further categorized. The asynchronous interrupts that are issued by external hardware circuits via an IOAPIC and sensed by the local APIC (by means of the `LINT[1:0]` pins) are all maskable. The external interrupt that is wired to the `NMI` pin is called the non-maskable interrupt (NMR), it can *under no circumstances* be masked. Exceptions cannot be masked at all. There exist three different kinds of exceptions:

- Fault: if the issue causing an exception can be corrected (e.g. a `page fault`) and it is possible to continue the disrupted code an exception is called a fault. The return address of a fault is usually the memory address of the faulty instruction.

[1]Since an architectural state does not have as many independent CPU components as an entire core [Int08a], the two options dual-core or hyper-threading CPU are not equivalent from a performance perspective. Resource conflicts are more likely in hyper-threading CPUs. This was one of the major problems when Sun Microsystems exploited the hyper-threading concept by enabling the T1 CPUs to simultaneously run 32 threads but only provided one single FPU: Depending on the application, the entire CPU stalled due to conflicts.

- Trap: A simple way to transport debugging information (e.g. a `breakpoint`) to the supervisor software. The return address points to the memory location directly after the instruction that was trapped.
- Abort: If a severe error occurs, the program that was terminated cannot be continued, a handler is called instead. Example: `double fault`.

Every interrupt has a unique identifying 8-bit number, called the *interrupt vector*. Vectors 0 and $3-14$,$16-19$ are the 17 predefined exceptions (for details see [Int08b] section 5.3.1), vectors 1, 15 and $20-31$ are unused – they are reserved for further use by Intel. The vectors $32-255$ are the user-defined interrupts that can be assigned to any external device. The missing vector 2 is assigned to the NMI. Each one of the 256 existing interrupts can be generated by means of the instruction `int n`. If an interrupt is issued by software, it cannot be masked, regardless of whether its vector is maskable or not.
The interrupt enable flag `IF` that is part of the `EFLAGS` register is used to mask interrupts. If the flag is set, the processor services hardware interrupts; otherwise, they are masked. The set interrupt flag instruction `sti`, and respectively the clear interrupt flag instruction `cti` are used to modify flag `IF` directly. The ability to execute these instructions depends on the current privilege level of the program or task attempting to do so.

In order to localize a handler routine address that is associated with an interrupt vector, a CPU internal table called interrupt descriptor table (IDT) exists. It has to be populated while the CPU operates in privileged mode (`CPL` is 0) by the operating system, usually during the boot phase of the system. The IDT can reside anywhere in the system's address space; it is referenced by the `idtr` register. Depending on the type of exception (or interrupt) the IDT may contain three different kinds of descriptors: task-gate descriptors, interrupt-gate descriptors or finally trap-gate descriptors. The main difference between these descriptors is that the interrupt-gate disables all maskable interrupts by default, whereas the trap-gate descriptor does not.
Each interrupt that is delivered to the CPU has its own priority called the interrupt priority [Int08b]. It is determined as follows:

$$priority = \lfloor vector\,/16 \rfloor$$

Since the vector numbers 0 - 31 are reserved or predefined, the user defined interrupts have the priorities 2 up to 15. A group of 16 interrupt lines sharing the same priority is called an interrupt priority class or priority level. A task priority register (TPR) is part of any local APIC (see next paragraph for details). It keeps a priority threshold defined by the supervisor software. Only interrupts with a higher priority can be handled by the CPU. If it is set to 0, all interrupts are served, if it is set to 15, only the non-maskable ones are handled. An additional register called the processor priority register (PPR) is automatically populated by the CPU; it can only be read by the operating system. This register represents the current priority at which the CPU is currently executing. Its value is calculated as follows:

```
if TPR[7:4] >= ISVR[7:4]
  then
    PPR[7:0] <- ISVR[7:0]
```

```
else
  PPR[7:4] <- ISVR[7:4]
  PPR[3:0] <- 0x00
```

`ISVR` represents the vector number of the highest priority bit of any service routine registered.

Advanced Programmable Interrupt Controller and Symmetric Multiprocessor Architecture

In recent years, the programmable interrupt controller (PIC) mainly used for Intel Architecture compliant systems was Intel 8259A. Usually, two of the controller chips – that handled eight interrupt lines each – where used in a cascade to have 15 IRQ lines available [Int88]. With multiprocessor systems spreading, the advanced programmable interrupt controller (APIC) became more common. When dealing with an APIC architecture, it is crucial to distinguish between the local APIC that is embodied in every CPU core and the IOAPIC that replaces the old-fashioned PIC [Int08b] (see also Figure 4.2). For the latter, controller circuits such as Intel 82489DX[2] or the Intel 2093AA [Int96] are mainly used.
This IOAPIC provides multi-processor interrupt management and implements static as well as dynamic distribution of symmetric interrupts across all processors of a system. If there exist multiple I/O subsystems, each one possesses its own set of interrupts.

- **Static distribution mode**. The IOAPIC delivers interrupts depending on their origins to specific local APICs. This assignment is stored within the internal Interrupt Redirection Table (IRT) of the IOAPIC. The interrupt is delivered to either a single CPU (unicast), a subset of CPUs (multicast) or all CPUs (broadcast).

- **Dynamic distribution mode**. As mentioned above, each local APIC has a programmable Task Priority Register (TPR) that yields the priority of the currently running process. The interrupt is delivered to the CPU running with the lowest priority. If two or more CPUs share the lowest priority, arbitration – based on a second priority that is reset every time an interrupt is delivered – is used.

As defined by Intel in 1997 [Int97], "the MP specification's model of multiprocessor systems incorporates a tightly-coupled, shared-memory architecture with a distributed interprocessor and I/O interrupt capability." Any SMP system that follows [Int97] is entirely symmetric, i.e, all CPUs are functionally identical, have the same status and are able to communicate with every other processor. In a nutshell: There exist neither hierarchy nor master-slave relationships among them.

Yet, Figure 4.2 distinguishes between a bootstrap processor (BSP) and the other application processors (AP). The BSP is responsible for initially booting the operating system while the application processors are activated later during the boot process itself.
By means of the local APIC, every CPU can send inter-processor interrupts (IPI) to every other CPU. When a CPU intends to send an IPI to another, only the targets identifier and interrupt vector is to be stored in the local interrupt command register (ICR). The interrupt is then automatically generated via the interrupt controller communication bus.

[2]Intel abandoned this series in 1999

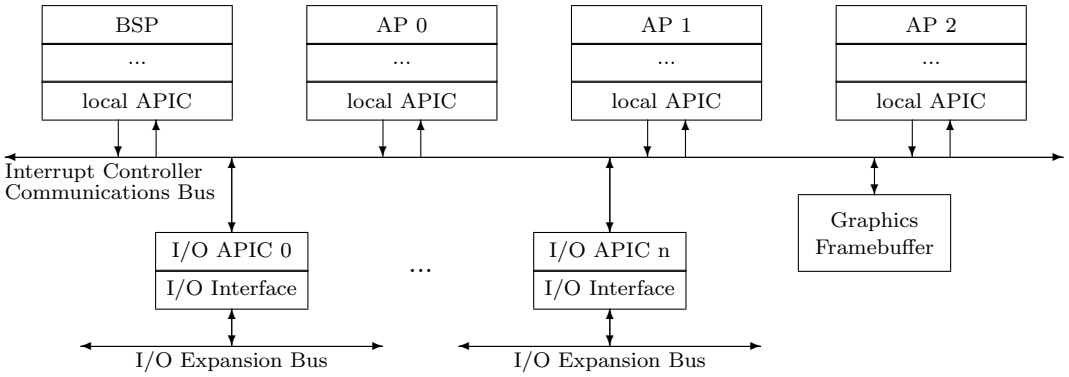


Figure 4.2: APIC configuration in an Intel SMP version 1.4 compliant system

Timing Facilities

Before defining the concrete machine setup that will then serve as the basis for all operating system models, another facette of the Intel Architecture must be considered: the existing timing facilities. A timing facility is a means that allows to create an operating system's internal representation of time.

- **Real Time Clock (RTC)**. The first chips that provided timing capabilities to home computer systems were Motorola MC146818 and compatible CMOS RAMs. Today, RTC devices are integrated into the I/O controller hub (ICH), the main part of the chipset.
- **Programmable Interval Timer (PIT)**. Different CMOS circuits, such as the Intel 8253 and 8254 programmable interval timer [Int94], were for a long time the most commonly used way to periodically fire timer interrupts and thus to create a system's internal representation of time.
- **High Precision Event Timers (HPET)**. The former Multimedia Timer [Int04] is a high resolution timing facility manufactured to synchronize multimedia streams. Many well-known hardware vendors follow the HPET standard that was mainly developed by Intel, Microsoft, Compaq, Toshiba and Phoenix.
- **Advanced Programmable Interrupt Controller (APIC)**. If there is no external [Int96] circuit available for generating timer interruptions, the APIC can be programmed to issue interrupts with the theoretical frequency of the systems frontside bus (FSB).
- **Advanced Configuration and Power Interface (ACPI)**. Every operating system uses the ACPI [TIM+00] system to keep track of the time the hardware is unused. Thus, the ACPI power management facilities also possess timing capabilities where interrupts can be issued on a one-shot basis only and not periodically.
- **Time Stamp Counter (TSC)**. This special CPU register is incremented every cycle. It can be accessed by means of the `rdtsc` assembly language instruction

[Int08b]. It is not capable of signaling any elapsed time at all, therefore it is mainly used to calibrate the timing facility used rather than being used as one itself.

Table 4.1.1 taken from [Koe02] lists all the crucial characteristics of the above mentioned hardware clocks and timers. Their capability to issue "one-shot" interrupts as well as to periodically deliver interrupts to the CPU is the most important feature with respect to the IHF models. For later discussion about soft real-time requirements (cp. Section 5.2.2), their resolutions (theoretical, real and the range of their periodicity) are elaborated as well.

	RTC	PIT	HPET	APIC	ACPI	TSC
Interrupt	yes	yes	yes	yes	yes	no
Periodic	yes	yes	yes	yes	no	no
Range	2Hz-8kHz	$\leq$ 1,19 MHz	$\leq$ 10 MHz	$\leq$ bus frequency	$\leq$ 14,3 MHz	n/a
Theoret. Resol.	$2^{1..13}$ Hz Steps	$\sim 10^{-2}$Hz	$\sim 10^{-7}$Hz steps	1 FSB-Cycle	$\sim 10^{-3}$Hz steps	1 Cycle
Real Resol.	n/a	1 Hz	n/a	20 FSB-Cycles	n/a	38 Cycles

Table 4.2: Characteristics of the different timing facilities

4.1.2 Specific Machine Setup

All models and later investigations are based upon a specific machine setup. This setup is defined by the parameters of the TwinUx project.

The TwinUx Idea
Scheidig [SS03] invented the TwinUx-Architecture inspired by the paradigm of graph-based execution of multimedia data [MP96] within operating systems. By then, only the experimental operating system Scout [Mos97] implemented this paradigm. See Figure 5.6 on page 102 for a simple example of a multimedia scenario graph.

In order to exploit this idea further, a second paradigm was introduced by Scheidig [Sch06]: A TwinUx system has to separate interactive processing from the execution of path-based multimedia content. To realize this, an additional entity named the coordinator was conceived [Sch03] to manage the two parts and to de-conflict the handling of shared hardware resources. The video output device for example is one of those shared resources. The overall architecture [Sim04] is depicted in Figure 4.3.
The multimedia subsystem is supposed to be a minimized operating system that does not include any interactive components or spooling. It incorporates all required multimedia functionality and is driven by a basic scheduling mechanism. It therefore contains the component extension (see Section 5.2.1) that decouples the multimedia functionality from the operating system layer. The interactive part is based on any general purpose operating system that is in place to handle all interactions with the outside world (the user) and to

communicate with pre-defined interfaces of the component extensions via the coordinator software.

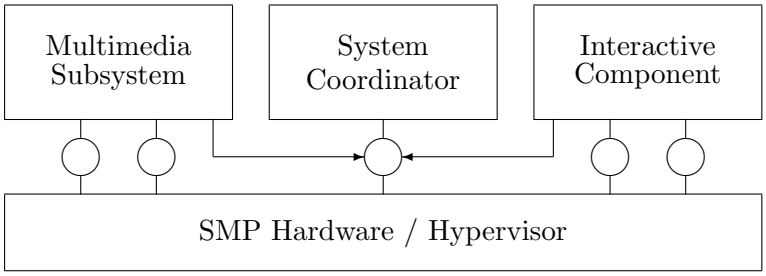


Figure 4.3: TwinUx architecture - circles depict abstract resources

The need for three processors directly derives from the TwinUx Architecture. Given that common symmetrical multiprocessor systems can only handle 2^n CPUs, four CPUs are considered. Since this thesis deals with the interrupt handling facility rather than with overall performance evaluation, a single Intel Core Duo Processor with hyper-threading capabilities could be regarded as sufficient. However, Figure 4.1 shows that architectural states do not each possess their own bus interfaces. It is therefore reasonable to disregard hyper-threading and to consider only multi-core CPUs.

- The setup contains four Intel Pentium 4 cores, i.e. either two Core Duo Processors or four conventional Intel Pentium 4 CPUs.
- The overall system is equipped with one IOAPIC that is connected to the Peripheral Component Interconnect (PCI) bus.
- An external timing facility which is either the PIT or the HPET is used to generate the periodic timer interruptions (uses one interrupt line).
- An additional real-time clock device exists in order to provide one-shot timer functionality (uses one interrupt line). This is the reasonable choice from the above discussion on timing facilities.
- The advanced configuration and power interface regulates idle times and device power. It also carries out thermal management (uses one interrupt line).
- A fixed set of peripheral devices is defined. Those devices are assumed to allocate an IRQ channel, i.e. one IRQ line in the interrupt controller each. IRQ sharing or the assignment of multiple channels per device are not considered.

  1. single IDE or SCSI hard drive (HDD)
  2. MPEG decoder card (MPEG)
  3. human interface device (HID) like a mouse that is connected via universal serial bus (USB)
  4. sound card (SND)
  5. network interface card (NIC)
  6. keyboard (KB)

  Each of the peripheral devices generates I/O interrupts using one interrupt line each.

4.1.3 Load Scenarios

In a system running with the previously defined peripheral devices, workload for the interrupt handling facility is created. Each device can issue interrupts, which are modeled as events – or more specifically path events (see Section 4.2.2.3) – at any point in time. An interrupt load scenario (or for short: load scenario) is a finite sequence of disruptive events, or formally:

Definition 4.1 ***Load Scenario (ESF)***
A load scenario is a finite sequence of path events $\overrightarrow{E}^n$ *together with the distances (in system steps) between their occurrences:* $scen \in \overrightarrow{E}^n \times \mathbb{N}_0^n$*, n is called the length of scen. The function length(scen) yields n.*
It holds that $\forall \vec{e}_\nu, \vec{e}_{\nu+1} : x_\nu \in \mathbb{N}_0$ *is the corresponding distance between these path events* $(0 \leq \nu < n)$*.*

Note that the previously used term *at any point in time* is disregarded by the formal definition: instead, system steps are used to describe the distance between two events. Later in the thesis (cp. Section 5.4), this somewhat unusual definition will prove helpful.

4.1.4 Load Profiles

While a load scenario is a concrete situation, an interruption load profile (or for short: load profile) defines the abstract ratio of device usage (load). Load profiles depend on the kind of usage of the system. For different categories of usage, e.g. interactive work, multimedia playback or scientific computing, different dedicated load profiles can be defined. The formal definition is based on path events once more:

Definition 4.2 ***Load Profile***
For each subset of path events $E_{disrupt} \subseteq \overrightarrow{E}$ *that represents disruptions, a load profile is defined. Let* $\mathfrak{occ} : \mathbb{R}^+ \times \overrightarrow{E} \rightarrow \mathbb{N}_0$ *be a function that yields the number of occurrences of a path event within a certain period of time. With* $e_0, \ldots, e_{n-1} \in E_{disrupt}$*, the load profile is the ratio:*

$$\mathfrak{occ}_\infty(e_0) : \mathfrak{occ}_\infty(e_1) : \cdots : \mathfrak{occ}_\infty(e_{n-1})$$

4.2 Modeling Approach

Based on the preliminary discussions about the underlying hardware platform presented so far, the approach for creating the IHF models is now given.
The modeling methodology can be described as a top-down approach. Starting from a very abstract level, the architecture of an operating system is described by hiding all details in black boxes. If source code is available for this particular SuI, more detailed sub-models can be created in order to fill the black boxes so that they become white boxes in terms of software engineering [JW96]. This especially applies to the IHF which is the crucial white box in this thesis: as the interactive part of an OS, it greatly influences the overall real-time capabilities of the system (cp. Section 5.2 for a detailed discussion).

All principles and paradigms will be explained in the following sections starting with a top-down definition of building blocks of an entire OS. A guide of how to model specific implementation details of any operating system such as interruptions, priorities or synchronization is then elaborated. A basic nomenclature facilitates easy comparison and conveyance of any model.

4.2.1 Top Down View

To create the IHF white box model, it is indispensable to reverse engineer the operating system's source code. Unfortunately, the modeling of an IHF is the most complex part of the whole operating system from a reverse engineering point of view. Any source code that implements the interrupt handling facility contains considerable amounts of low-level assembly language instructions – which is the most difficult language to reverse engineer – for two main reasons:

1. In the compiled operating system kernel, the part that represents the IHF is executed more often than other parts (every clock tick, every system call as well as any interruption traverses this section of the OS). Only assembly language code can be sufficiently optimized and tuned with regard to high performance.

2. The hardware-dependent parts of the IHF need to be coded by means of the native instruction set of the target hardware platform. Usually, there exists no high-level language equivalent for instructions such as `iret` that returns from an interrupt handler or `cli` that disables all local interrupts (cp. Section 4.1.1).

Yet the fact that all interrupt handling starts at a well-defined point (which is derivable from the IDT entries) eases the reverse engineering efforts a little. Besides the challenging task of reverse engineering, modeling the IHF as part of the complete OS that is looked at from a top down perspective requires some further consideration.

As already mentioned, when modeling an entire OS or a dedicated part of it, it is useful and necessary to provide different levels of abstraction according to the focus and purpose of the model. This usually results in a layered (different levels of abstraction) and modularized (different black or white boxes) model. It is easy to see that the contrary, i.e. giving one flat model for the internal states of an OS with constant level of detail is not helpful.
Different granularities of the model are required when focusing on different parts of the system. For example, a system state depicting the first steps of handling an interruption may represent only a few machine instructions, whereas a state describing the update of system statistics may represent hundreds of high-level language statements and thus a large number of machine instructions. Those differences should be encapsulated by modularity or separated by hierarchy. In our case, only the IHF is modeled with high granularity.

Although the paradigm of partiality in modeling is applied, investigating one isolated aspect of an OS is not possible. This is so because of the concurrent and reactive nature of the IHF as a subsystem. Unpredictable side effects may cause other paths outside this particular subsystem to be taken. Therefore, all possible causes for those effects have to be taken into account carefully.

Because of this, a predefined stub for a model of the IHF is necessary. In any operating system, the following subsystems exist which form the building blocks of an OS model [SGG01]:

1. **User space processes**[3]
   For some decades, pseudo-parallelism as a paradigm has been mostly exploited for all GPOS [Han73]. Processes, tasks and threads [IEE01] implement this principle in nearly all modern operating systems.
   Hence, the set of all processes and their logical states is an important facet for all models as well. In general, any process can be active, i.e. running, blocked or ready, i.e. waiting. Most operating systems have implemented more than those three very basic states.

2. **Kernel control paths (KCP) / Intermission kernel control paths (IKCP)**
   The main part of any IHF model is the set of kernel control paths (KCPs). A KCP depicts the control and data flow within the operating system kernel. In general, it is the description of any action carried out while the CPU is in privileged mode. It does not necessarily need to be triggered by a disruption of any other control flow.
   A special case are the intermission kernel control paths (IKCPs). They represent the paths through the kernel that were triggered by an intermission of the currently active tasks, i.e. by an interruption of any kind. In the case of multiple processors, the IKCP building block can exist several times in a model.

3. **Multi-processor and interrupt platform**
   If the sophisticated mechanisms of the Intel Architecture for interrupt distribution and handling as described in Section 4.1.1 are used, they need to be represented in the model. This block represents the arbitration and distribution of interrupts within the Intel IA32 hardware platform. It incorporates the modes of operation of the interrupt controller circuits as well as the numbers, types and properties of interrupt sources.

4. **Synchronization primitives**
   Especially in multi-processor systems, critical regions have to be protected from concurrent access of simultaneously executed code. The operating system primitives implementing the protection have to be defined or at least modeled by an interface or any other means.

5. **Memory management interface**
   When modeling system calls or kernel control paths, memory handling has to be modeled by means of a common interface. When modeling IKCPs, the management of segmentation and paging can be crucial for the accuracy of the model as well:

   - User space processes could raise exceptions when accessing a swapped memory page.
   - Any IKCP could execute segment switches or cause pages to be swapped.

[3] Note that the subset of processes located in kernel space memory is modeled separately.

- In some rare cases, privilege levels within the operating system are implemented by means of memory segmentation.

In general, concepts and implementation of memory management are not part of the thesis. Thus, the according models contain abstract interfaces for the memory management at most.

The *top level model* of an OS gives a clear overview of architectural types and design paradigms the system is based on. It unites and therefore contains all more detailed building blocks (subsystems) in a hierarchical manner.
Depending on the implementation and the complexity of an operating system and its IHF, it can become necessary to elaborate all of these building blocks. For the systems investigated in this thesis, it will prove feasible to abstract most of them in the models. Note that creating a wholistic model for the entire operating system would contradict the fact that there is no uniform implementation paradigm throughout the kernel.

4.2.2 Modeling Implementation Patterns

Having defined the formalism, discussed the hardware platform and explored some preliminary issues about the models in general, it is necessary to provide a more detailed guideline on how to translate certain source code elements into *ESF* models. Four characteristic operating system patterns will be elaborated:

1. Sequential control flow including function calls
2. Disruptions
3. Synchronization
4. Conditional branching

No interrupt handling facility follows the object-orient paradigm at the lowest level of implementation. Though the more high level interrupt service routines can be implemented in an object-oriented way, no object-oriented patterns are reflected in the models as these routines are treated as black boxes in this thesis.

4.2.2.1 Modeling Sequential Control Flow

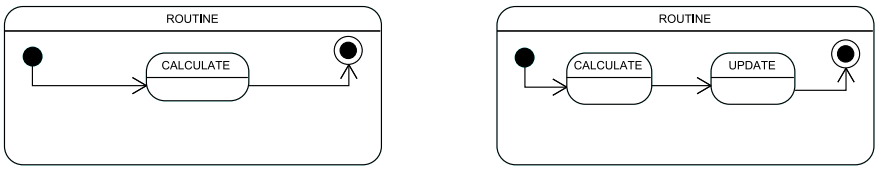
The easiest part of any building block to model is a sequential control flow that does not branch and cannot be interrupted. It may contain function calls to subroutines. Each subroutine call should be modeled by adding an additional level of hierarchy. The following pseudo-code depicts the simplest case:

```
void routine (void *param) {
    disable_interrupts();
    calculate(param);
    enable_interrupts();
}
```

A routine that is not interruptible calls a subroutine to carry out some calculations. Figure 4.4(a) shows the according statechart model. If there is more than one function call (or any other computational entity), a sequential transition is used – Figure 4.4(b) shows this

case. The transition t with $src(t) = \mathsf{CALCULATE}$ and $src(t) = \mathsf{UPDATE}$ is a sequential transition according to Definition 3.1.

```
void routine (void *param, void *data) {
    disable_interrupts();
    calculate(param);
    update(data);
    enable_interrupts();
}
```



(a) a simple routine consisting of one atomic part

(b) a routine that contains two dedicated parts

Figure 4.4: Non-interruptible routines

Note that due to the considerations in the previous sections, namely different granularity for different code regions, there might be other criteria to either group a region of code to one state or to model it preserving its peculiarities. Nevertheless, this decision must be made carefully because it can change the significance of some of the later indicators (cp. Section 5.4).

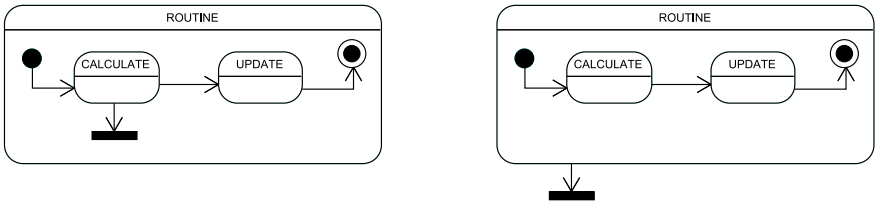
4.2.2.2 Modeling Disruptions

If an elementary part of code is interruptible, such as the `calculate(param)` function call in the following pseudo-code, interruptibility has to be modeled on the basic state level as depicted in Figure 4.5(a). By doing so, it is guaranteed that the finest granularity possible is achieved. The transition t with $src(t) = \mathsf{CALCULATE}$ heading towards the event bus represents interruptibility on the level providing the finest granularity.

```
void routine (void *param, void *data) {
    calculate(param);
    disable_interrupts();
    update(data);
    enable_interrupts();
}
```

Whenever a collection or any other coarse grained part of the code is interruptible, this fact needs to be modeled on the appropriate non-basic state level. The function `routine` now is assumed to be interruptible at any point in time, thus the transition (Figure 4.5(b)) depicting this fact originates from the state ROUTINE rather than being duplicated to originate from both CALCULATE and UPDATE. The following pseudo-code represents such a situation.

```
void routine (void *param, void *data) {
    calculate(param);
    update(data);
}
```



(a) one part of the routine is interruptible

(b) a routine that is entirely interruptible

Figure 4.5: Interruptible routines

Note that the fact of whether a region of code is interruptible or not is a sufficient condition for grouping code to a single model state. It is a fact that due to grouping of code, the level of abstraction is not constant throughout a model. Since the proximity of the model to the source code is not a requirement in this thesis, these variations are no issue. However, when generating code from the *ESF* models would be considered, this varying level of abstraction would need to be confronted.

4.2.2.3 Modeling Synchronization

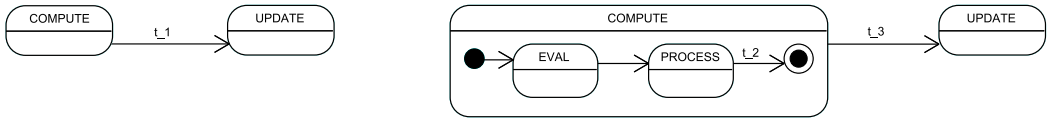
A sequential control flow may enter a critical region. Access to critical regions must be treated carefully (see Section 4.3). When concurrent access is synchronized, the impact on the control flow must be incorporated in the models although the primitives themselves are not elaborated any further. The following pseudo-code gives an example of a critical region synchronized by a spinlock.

```
void routine (void *param, void *data) {
    spinlock_t the_lock;
    calculate(param);
    spin_lock(the_lock);    // obtain lock or "spin around"
    update(data);           // critical region, data is accessed elsewhere
    spin_unlock(the_lock); // finished critical region, release the lock
}
```

The instruction `spin_lock` is used as implemented e.g. in the Linux kernel sources (see e.g. [CRKH05]). When creating *ESF* models, sequential transitions enhanced with conditions can be used to depict these circumstances in a very easy and intuitive way.

Definition 4.3 ***Synchronized Sequential Transition***
A transition $(s_1, e/a, s_2) \in T$ *where* $s_1, s_2 \in S$, $e \in E$ *and* $a \in A$ *is called a synchronized sequential transition iff* $e = \lambda[c]$, s_1 *basic and* $a = \mu$, $c \in C$ *is called the synchronization condition. A synchronized sequential transition is denoted* t_{sync}.

In Figure 4.6(a), the transition t_1 is such a synchronized sequential transition. When the synchronization condition becomes true, the system steps further. The fast forward problem does not affect this at all.
When dealing with synchronization on a non-basic state level, this approach does not reflect the intended semantics any longer. Let us for a moment consider transition t_3 from Figure 4.6(b) being labeled with $\lambda[c]/\mu$, c being the synchronization condition as defined. It is clear that in this case, the state COMPUTE would be left regardless of whether



(a) synchronization applied to basic states

(b) synchronization cannot be applied to superstates

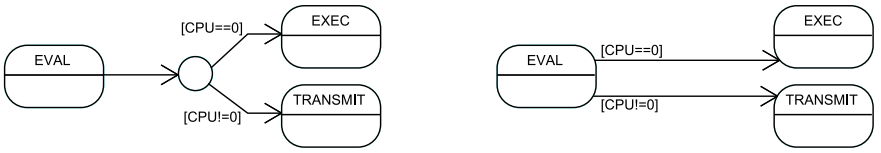
Figure 4.6: Synchronization by means of modified sequential transitions

it completed (PROCESS might never be entered). This is not the intended meaning of synchronization. If we apply the approach of raising a special event $term(s_{last}) \in E$ when taking t_3, t_3 could be defined as $term(s_{last})[c]/\mu$. Unfortunately, this would require the finishing of COMPUTE and the fulfilling of the synchronization condition in the very same system step due to the statechart semantics.
From this it follows that synchronization must be modeled on a basic state level only. In the given example, t_2 must be a synchronized sequential transition, but not t_3.

4.2.2.4 Modeling Conditional Branching

The statechart formalism offers two nearly identical ways of modeling conditional branches. The following pseudo-code can thus be represented by either Figure 4.7(a) or Figure 4.7(b).

```
eval(cpu);
if (cpu == 0)         // Bootstrap CPU ?
    exec(cpu);        // Execute locally
else
    transmit(++cpu);  // Otherwise delegate
```



(a) branching by means of a connector

(b) branching with different transitions

Figure 4.7: Possible ways to model branching

When dealing with sequential transitions that represent the branches, the first version (4.7(a)) is the one to be used when modeling operating systems code with the *ESF*. The first transition segment leading from EVAL to the connector is to be a sequential one, the following transition segments carry the branch condition $l = \lambda[cpu == 0]/\mu$).

When dealing with branches that might have several optional values – as the number of CPUs in the given example can be any number – the use of an expression $[val_1 \doteqdot val_2]$ is unavoidable. However, when considering a binary choice, a simple condition that can be *true* or *false* only is the preferable way to model a branch.

IHF Impact on the Modeling Paradigm

Although the presented implementation patterns are widely used throughout the IHF subsystem, best practices of software engineering are rarely applied to the kernel code in this particular area. As shown later in Section 4.4, the code is highly optimized for performance reasons and thus often implemented in assembly language. These optimizations are not necessarily reflected in the later *ESF* models. However, this fact greatly influenced the choice of the modeling technique.

4.2.3 Nomenclature and Conventions

For better readability and an easier interchange of operating system models, a general nomenclature is defined. This nomenclature consists of the definitions of a set of universal branch categories for path events that all models must have in common, the introduction of transition tables and the use of the UML 2.0 submachine mechanism.

4.2.3.1 Events

Events in any operating system can be divided into three categories: disruptive, scheduling and signaling events. For now, scheduling and signaling events are put aside as we focus on disruptive events (DEVT). These are defined within the models as a set $DEVT \subseteq \overrightarrow{E}$, i.e. they are represented by path events.

Classification of Disruptive Events Each system model implements a distinct tree of disruptive events. This tree categorizes dedicated disruptive events based on the complete path from root to leaf. Each of these events is represented by a path event, each level of the tree depicts a branch category with the nodes at this level being the specific branch identifiers of this category.

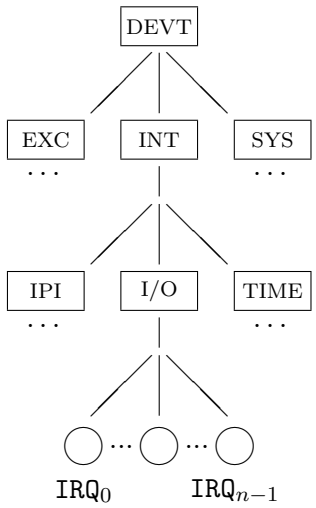


Figure 4.8: Disruptive event tree

Common to all trees for different systems is the distinction between interrupts, exceptions and syscalls $\{INT, EXC, SYS\}$ as inferred from the Intel architecture (first hierarchy level

of the tree in Figure 4.8). The sources for interrupts processed can be subdivided into three classes: interrupts delivered by I/O devices (I/O), inter-processor interrupts (IPI) and timer interrupts (TIME). The lower levels are different for each concrete system.
The branch categories are, as already mentioned, defined on the basis of this disruptive event tree. As we deal with a hardware platform with parallel CPUs, the first branch category is always the number of the CPU that is addressed by the event. The further categories are then specified starting with the first hierarchy layer of the tree. Hence, for an interrupt issued by the peripheral device allocated to IRQ channel 2 that is then delivered to the first CPU, the corresponding path event would be $e_{0,INT,I/O,2}$. The specific machine setup as provided in Section 4.1.2 defines the concrete I/O interrupts of its devices somewhere below the I/O hierarchy level of the tree, depicted as circles (leaves).

4.2.3.2 States

States in the operating system models are representations of certain activities that are carried out. They represent what the system is doing at the moment, i.e. executing a function of some kind. Since this activity is regarded as atomic, the details and sequences of the execution are of no interest. The focus is on what a system does from an abstract point of view, not on how it does it. Consequently, states can model more than one atomic point in time.

4.2.3.3 Transitions

For transitions, an abbreviated label (or identifier) is used to avoid the cluttering of figures and diagrams. This identifier refers to the detailed description of the transition that is provided separately.

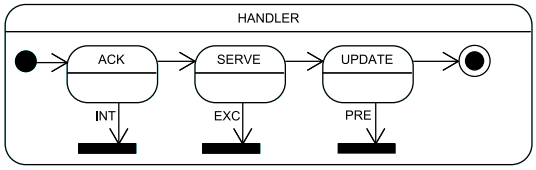


Figure 4.9: Example statechart with three transitions associated with abbreviated labels

Figure 4.9 is a simple statechart using identifiers for labeling transitions instead of correct statechart labels. Table 4.3 provides the detailed information about the three sample transitions.

ID	Source	Target	Event/Action	Description
INT	ACK	IKCP_0	$DEVT_{0,INT,I/O,\iota}/\mu$, ι = device	Any device
EXC	SERVE	EXC_HD	$DEVT_{0,EXC,\epsilon}/\mu$, $\epsilon \in \{8, 9, 18\}$	Any abort
PRE	UPDATE	SCHED	$DEVT_{0,INT,IPI,0}/\mu$	Preemption

Table 4.3: Example transition table for the statechart in Figure 4.9

Note that Table 4.3 also illustrates the property of the event bus that the real target of a transition leading towards an event bus connector must be given: in the first row of the table, the target of the transition is the state IKCP_0 (which is not part of this example).

The example shows two different cases:

1. In the third row, there is only one path event $DEVT_{0,INT,IPI,0}$. Subsequently, $DEVT_{0,INT,IPI,0}/\mu$ directly corresponds to the statechart label of the transition in question.

2. The first and second rows show otherwise. A set of multiple path conditions (cp. Definition 3.6, e.g. $DEVT_{0,EXC,\epsilon}$, $\epsilon \in \{8, 9, 18\}$) is given. In this case, the following holds for the statechart label:

$$\forall\, pc_\nu \in \{pc_0, \ldots, pc_{n-1}\} \subseteq PC : l = \bigvee_{\nu \in \{0 \ldots n-1\}} e_{pc_\nu}/\mu$$

with n being the number of different disruptive events covered by the respective set. The actual (implicit) transition label is constructed from a disjunction of all corresponding path events.

Sequential Transitions
In the visual representations of the OS models, sequential transitions are not to be labeled. By this, some additional clearness of the representation is gained. In Figure 4.9, the transitions t_1 with $src(t_1) = \mathsf{ACK}$ and $tgt(t_1) = \mathsf{SERVE}$ as well t_2 with $src(t_2) = \mathsf{SERVE}$ and $tgt(t_2) = \mathsf{UPDATE}$ are such sequential transitions:

Furthermore, sequential transitions are not to cross hierarchical levels, i.e. they must not be used as inter-level transitions. From their meaning itself it is obvious that this usage does not reflect the intended meaning. As a matter of course, all other types of transitions are allowed to be inter-level transitions.

4.2.3.4 Usage of Event Busses in Combination with CTSC

When the event bus connector is used in combination with cartesian transition set connectors, *ESF* does not at all restrict the ways of combining the two constituents. Figure 4.10 illustrates a maximal chain.

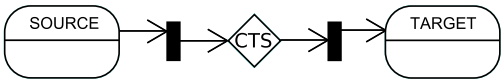
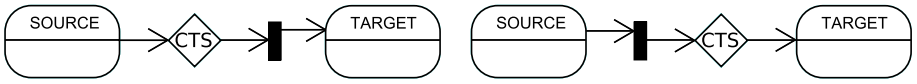


Figure 4.10: Usage of CTSC together with event-bus connectors

When creating IHF models, it makes sense to limit this usage to one of the two possibilities given in Figure 4.11. This limitation does not impede the expressiveness of the model in any way, but rather facilitates more consistent modeling. Which variant is used does not matter at all, as long as it is used consistently throughout the entire model.

4.2.3.5 UML 2.0 Submachines

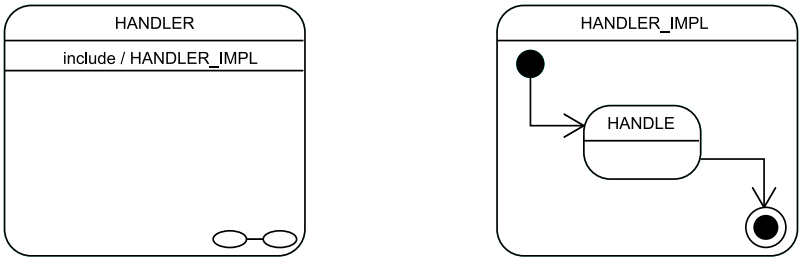
All black or white boxes of the model (except for the top level model) can be defined as UML 2.0 submachine states [RJB05]. If any building block contains more detailed information about the SuI, further submachine states can be defined additionally.



(a) Event bus followed by CTSC (b) CTSC followed by event bus

Figure 4.11: Possible limitations of the usage

Such an include mechanism was always part of the chosen Statemate semantics [HN96]. In the Statemate tool, includes were depicted as @NAME . Its consequences are of a notational nature only; it has no semantical implications at all.



(a) a UML 2.0 submachine state (b) the internal representation

Figure 4.12: UML 2.0 submachine mechanism

4.3 Operating Systems

Before modeling some concrete operating systems, the choice of which to pick must be narrowed down. There exists a variety of different systems dedicated to different purposes and characterized by numerous criteria. The facts that the Intel Architecture was chosen as the underlying hardware platform and that the operating system must be SMP capable already rule out a considerable number of OS. Some operating systems matching the given requirements will now be discussed further.

4.3.1 Miscellaneous Properties

To begin with, some miscellaneous properties are assessed in order to develop a first heuristics about operating systems that might be suitable for modeling.
The greatest difference between different operating systems lies in the kernel architecture. In operating systems theory, one distinguishes mainly between three types of kernel architectures.

1. **Monolithic Kernel Systems**. A monolithic kernel contains all functionalities of the operating system. All kernel libraries are statically linked into one kernel binary. Hence the range of functionality is determined the moment the system is compiled. Monolithic systems are on the one hand inflexible and not modifiable but on the other hand, the tight coupling allows for very efficient optimization [Kon96]. In general, such systems are used for performance reasons.

2. **Modular Kernel Systems**. A modular kernel allows for loading additional functionality during runtime. The majority of the kernel libraries are thus linked dynamically. Kernel modules such as device drivers, file systems or even schedulers can be added at any point in time. This approach addresses the shortcomings of monolithic systems. Unfortunately, malicious modules can affect the overall safety and stability of such operating systems [SS96].

3. **Micro-Kernel Systems**. A micro-kernel system (or μ-kernel) only contains the functionalities of process management, a rudimentary interruption handling and inter-process communication means. Every other function is implemented as a "normal" process. Every device driver and internal service (such as scheduling or memory management) is outsourced to processes. The main challenge is to implement such a loosely coupled system in an efficient way [Lie96b].

Since the boundaries between these categories overlap, in reality, mainly hybrid systems (monolithic/modular and modular/μ-kernel) exist. Hybrid approaches address partially the performance problems of early μ-kernel implementations [HHLS97].

L4 and Pistachio

In 1995, the German GMD developed a μ-kernel called L4 [Lie96a] that was the basis for numerous later operating systems such as L4Linux[4] and L4Ka::Pistachio[5], short Pistachio. The original μ-kernel implementation L4/x86 does not support preemption (for the definition of preemption as a general concept see Section 4.3.2), hence nested interrupts cannot be processed in the order of their occurrences but rather in the order of their priorities [AH99]. As a consequence, priority inversions occurred. The Omega0 project [LH00] as well as the Fiasco kernel [Hoh98] address this issue.
Any synchronization within an L4 kernel is based on messages, the transmission of messages is always synchronous and unbuffered. A mechanism of grouping processes called "Clans and Chiefs" [Lie92] facilitates multi-cast transmission, especially amongst server processes. The only scheduling strategy implemented is time slice policy-based on one parameter, the maximal controlled priority.
The Pistachio micro-kernel operating system is a new implementation of the L4 API Version four. It was designed to overcome the shortcomings of L4 while preserving its functional specifications. The focus of the re-implementation was on the design of an efficient inter-process communication (IPC) facility.

Mach

In the years 1985-1993, the micro-kernel system Mach[6] [BBB97] was developed by researchers at the Carnegie-Mellon University, supervised by the US Department of Defense. There are still numerous operating systems based on this kernel, MacOS X[7], GNU/Hurd[8] and MkLinux[9] are three of the most popular examples [SG91]. Mach was launched as a

[4] http://os.inf.tu-dresden.de/L4/LinuxOnL4/
[5] http://l4ka.org/projects/pistachio/
[6] http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html
[7] http://www.apple.com/macosx
[8] http://www.gnu.org
[9] http://www.mklinux.org

BSD variant, but later development turned away from that path. A Mach system (Version 3) [Leo91] contains comprehensive handling of physical devices, a highly efficient process and thread management and full kernel preemption. It can handle nested interrupts as delivered by the underlying hardware platform. The internal synchronization is based on messages. The inter-process communication is a mixture of a modified System-V standard and POSIX (see Section 4.3.2) means. The Mach scheduling policy is based on simple priorities: no optional strategies are implemented [RJO+89].

Linux
Started in 1991 by Linus Torwalds, the Linux kernel[10] is an open source project. A worldwide community performs the development, as well as different companies such as Redhat, Novell or IBM. The general purpose operating system Linux [Rus99] is used in desktop-, server- and embedded systems.
The current kernel versions 2.6.x allow for optional kernel preemption, nested interrupts and the use of SMP hardware.
Linux contains numerous primitives for internal synchronization of interleaving control flows and IPC. At compile time, it is decided which specification to follow for IPC: System-V standard or POSIX. Linux offers amongst others a conventional time sharing policy, a round robin policy and a simple first-in-first-out-based strategy [BC01].

OpenBSD
The OpenBSD[11] Kernel Version 4 was released in 2006. OpenBSD is an open source variant of the Berkley Software Distribution (BSD) Unix [Luc03]. The monolithic kernel possesses full SMP capabilities. It does not allow for full kernel preemption but handles nested interrupts and uses software interrupts in conjunction with system priority levels. OpenBSD is not fully POSIX-compliant, however it implements numerous POSIX primitives for the IPC. Since the kernel is monolithic and hence no external modules are provided, the internal kernel synchronization still only relies on a few primitives. OpenBSD uses a dynamic priority-based algorithm to schedule user processes [Erk05].

OpenSolaris
When Sun Microsystems decided to create an Intel x86 version of their SPARC-based general purpose operating system Solaris (or SunOS) an open source project called OpenSolaris[12] was founded later. OpenSolaris is based on the modular kernel of the commercial Solaris 10 operating system [MM06].
All System-V and POSIX IPC primitives are implemented, OpenSolaris handles multiple CPUs, nested interruptions and supports full kernel preemption. A variety of different scheduling policies is integrated into the Solaris scheduler: from traditional timeshare to real-time, from fixed priority scheduler to fair share scheduling.

[10] http://www.kernel.org
[11] http://www.openbsd.org/
[12] http://opensolaris.org

QNX
QNX Software Systems develops and maintains the commercial hard real-time (see Section 5.1.2) operating system QNX Neutrino[13] [QNX06]. In order to achieve a transparent development process, the QNX sources were published in 2007. QNX Neutrino follows the μ-kernel approach. QNX offers fully controllable kernel preemption, fine-granular locking of interrupts and the processing of nested interrupts.
Synchronization means differ between those used to control POSIX threads, internal kernel synchronization and conventional IPC. The IPC implementation is comprehensive and fulfills both System-V and POSIX standards. A real-time scheduler is an inherent part of the system, other schedulers or strategies cannot be implemented.

Table 4.4 provides an overview of the discussed general criteria.

	SMP Capabilities	Kernel Architecture	Real-Time Capabilities	Languages	Standards
L4	none	μ-kernel	GPOS	ASM, C, C++	none
Pistachio	SMP (limited)	μ-kernel	GPOS	ASM, C, C++	none
Mach	full MP	μ-kernel	GPOS	ASM, C	none
Linux	full SMP	monolithic, modular	GPOS	ASM, C	System-V, POSIX (limited)
openBSD	full SMP	monolithic	GPOS	ASM, C	POSIX (very limited)
OpenSolaris	full SMP	modular	real-time	ASM, C	POSIX, System-V
QNX	full SMP	μ-kernel	hard real-time	ASM, C	POSIX

Table 4.4: Comparison of operating systems, general criteria

4.3.2 Detailed Criteria

Now, more detailed criteria are given to further classify the subset of operating systems. The focus of the following considerations is on real-time aspects rather than general properties.

Nested Interrupts
The Intel Architecture supports nested interruptions. Therefore, an operating system must be capable of disrupting any execution of a previously started interrupt handling. The processed kernel control path (KCP) is left (recursive descent) and the new path is

[13] http://www.qnx.com

entered. This property is called nested interrupts and evidently, this is crucial for achieving the goals of this thesis.

Kernel Preemption
When a kernel of an OS can be suspended by other kernel threads or any process residing in the same address space, this system property is called kernel preemption. A prerequisite for kernel preemption is that all interruptible functions are reentrant, i.e. they can be resumed any time without losing data integrity.
When dealing with μ-kernel systems, this property and the ability to handle nested interrupts overlap because any interrupt service routine implemented for such a system is a decoupled server process.

Synchronization
Since the later considerations do not include the synchronization primitives but only the consequences of those (see Section 5.4.6), these primitives are not elaborated in depth. However, in order to select dedicated systems from the previous short list, it is necessary to look at kernel-level synchronization and IPC primitives. In single-CPU systems, kernel synchronization can be avoided by simply prohibiting kernel preemption. Unfortunately, this trivial solution does not work with multi-processor systems. In these, all critical regions must be protected from concurrent access by one of the following primitives:

1. Per-CPU Variables
2. Atomic Operations
3. Compiler Optimization Barriers
4. Conventional Memory Barriers
5. Locks/Spinlocks
6. Mutexes
7. Semaphores
8. Condition Variables
9. Interrupt Disabling
10. Messages

Inter-Process Communication
In any current operating system, regardless of whether the underlying hardware platform is a multi-core one or not, the set of user space processes is exposed to (pseudo) parallelism. Therefore, it is vital to provide a set of communication primitives (and thus means of synchronization) for user space processes:

1. Semaphores
2. Messages (System-V)
3. Message Queues (POSIX)
4. Shared Memory
5. Signals

6. Pipes
7. FIFO Pipes
8. Sockets

There exist two separate standard definitions for IPC implementations:

1. System-V IPC: The vendor of the Unix System III, namely AT&T, defines semaphores (1), messages (2) and shared memory (4) as IPC primitives.
2. The POSIX IPC specification [IEE01] defines messages queues (3), shared memory (4), signals (5), pipes (6), FIFO pipes (7) and sockets (8) as means of inter-process communication.

Table 4.5 provides an overview on the operating systems previously presented showing the detailed criteria.

	Nested Interrupts	Preemption	Kernel Synchronization	IPC
L4	no	no	(10)	(2)
Pistachio	full	yes	(10)	(2)
Mach	full	full	(6) (7)	(1) (2) (3) (4) (6) (8)
Linux	full	optional	(1) (2) (3) (5) (7) (9) (10)	all
OpenBSD	full	limited	(5) (6) (7) 8) (9) (10)	(1) (4) (5) (6) (8)
OpenSolaris	full	optional	(5) (6) (7)	all
QNX	full	full	(2) (4) (5) (6) (7) (8) (10)	(2) (3) (4) (5) (6) (7)

Table 4.5: Comparison of operating systems, detailed criteria

From this table it is obvious that choosing the pure L4 kernel would not serve the goals of this thesis since it does not have nested interrupt capabilities. Hence, Pistachio is chosen as a μ-kernel system because it is more modern and its architecture better structured than Mach.

From the general purpose operating systems, one modular and one monolithic kernel is chosen. OpenBSD is a reasonable choice for a complete statically linked kernel. Either OpenSolaris or Linux could be modeled as modular GPOS. Since the Linux community is much larger than the OpenSolaris community, there exists more documentation and support for the reverse engineering process for Linux.

Although the main focus of this thesis is on soft real-time systems, it would have been desirable to have a detailed model of a hard real-time system such as QNX. Unfortunately, the sources were published too late to include such a model in this thesis (cp. Section 8.2).

4.4 Models

Now, the concrete models of Linux, OpenBSD and Pistachio will be presented.

4.4.1 Modeling Process

This section describes the technical process of creating the IHF models. Since there exists no tool suite for engineering operating systems at all, a variety of tools has been surveyed to build up some adequate work chain.

- **OpenGrok**[14] is a web-based source code browser and search engine developed by Sun Microsystems to ease the community-based development of OpenSolaris. It creates hyper-linked code references from any C/C++ source package.

- **The LXR project**[15] by Gleditsch and Gjermshus is an older source code browsing system used to handle and document the Linux kernel sources. It is web-based like OpenGrok and allows for various search modi as well as comparison of different versions and hardware architecture ports of the same source code.

- **The Red Hat SourceNavigator**[16] is a stand-alone source browser that allows for quick and detailed tracking of logical paths throughout source code. Unfortunately, this software was discontinued several years ago and the latest available version still has severe performance problems.

- **Statemate**[17] is the commercial development tool for statechart design. It is sold and maintained by the Telelogic company. This research and thesis was partially supported by the Telelogic University Research Programme.

- **ArgoUML**[18] is a free of charge and open source UML modeling tool that supports all standard UML 1.4 diagram types, including statecharts. It is capable of exporting diagrams into the Meta Object Facility (MOF) 2.0/XMI Mapping Specification [OMG05] format. This allows statecharts to be saved in an XML-based standardized manner. Furthermore, statechart diagrams can be exported to the scalable vector graphics (SVG) format, a W3C recommended standard [W3C03] for scalable images and graphics. The SVG format is also used for visualizing and evaluating operating system traces at the chair for operating systems research at Saarland University [Koh07]. The **Inkscape**[19] graphics software is used as an editor and converter for the SVG format.

Applying the tools exposed a number of shortcomings: Beside many others, Statemate was not able to exchange the models with any other software while ArgoUML was not able to handle the resulting model sizes.

[14] http://www.opensolaris.org/os/project/opengrok/
[15] http://lxr.linux.no/
[16] http://sourcenav.sourceforge.net/
[17] http://www.telelogic.com/
[18] http://argouml.tigris.org/
[19] http://inkscape.org/

Implementation details of operating systems impede not only the use of the source browsing tools but also complicate the manual reverse engineering process considerably.

- Assembly language parts are highly efficient but greatly complicate a deep understanding of the systems' internals. Usually, only the machine dependent or performance critical parts of an OS should be implemented this way.

```
 /* Load the potential sixth argument from user stack.
  * Careful about security.*/
    cmpl __PAGE_OFFSET-3,%ebp
    jae syscall_fault
1:  movl (%ebp),%ebp
```

  Often, assembly language code relies on some prerequisites, e.g. the call stack created by the used compiler. Such code (the example is from the Linux 2.6 kernel) is extremely hard to trace.

- Poor coding: When, for example, an encapsulation of underlying functionality is not done properly, this causes more difficulties as the following example from the Pistachio operating system easily shows:

```
/* deactivate APIC timer */
local_apic.mask(local_apic_t<APIC_MAPPINGS>::lvt_timer);

/* activate APIC timer:*/
write_reg(APIC_LVT_TIMER,(((periodic ? 1 : 0) << 17)) | irq);
```

  The function that deactivates the APIC timer is encapsulated in a proper and intuitive way, its companion function is not. Such engineering shortcomings frequently appear in a lot of real-world systems.

- Compiler macros and defines are not sensed by common code browser systems since they do not follow the same syntactical rules for declarations as variables. However, the C compiler gcc of the GNU Compiler Collection[20] can be used to only preprocess the source files without compiling them by setting the option `gcc -E`. While the resulting sources do not contain any definitions, they are hardly readable. This work around therefore is only to be used with caution.

The reverse engineering process for modeling the IHF fortunately has a common starting point: the interrupt descriptor table (IDT). From there, all source code parts that make up the IHF can be reached by performing a linear search of the code. This strategy is crucial for successful creation of the models. In order to analyze the content of the IDT, the initialization phase of the system must be tracked and understood. This is the first major part of work in reverse engineering any of the chosen SuIs.

4.4.2 SuIs

There are three systems chosen from the preliminary choice conducted in Section 4.3. The IHF model for the Linux 2.6 kernel (cp. Section 4.4.3) was the first model to be created

[20] http://gcc.gnu.org/

by the author. This model then served as a guideline for creating others. Gogolok [Gog07] modeled the OpenBSD 4.0 interrupt handling facility while Wieder [Wie07] chose the experimental μ-kernel system Pistachio.
Table 4.6 lists the sizes of the three IHF models and compares them to an estimated size of non-*ESF* models. These estimates are based on fictitiously unwinding the *ESF* constituents to conventional statechart elements based on the *ESF* transformation rules (cp. Section 3).

Model	Trans. (ESF)	States (ESF)	Trans. (Non-ESF)	States (Non-ESF)
Linux	990	602	$\sim$ 13 000	$\sim$ 8 000
OpenBSD	1062	612	$\sim$ 18 000	$\sim$ 12 000
Pistachio	801	550	$\sim$ 11 000	$\sim$ 5 000

Table 4.6: Model sizes of the three SuIs – *ESF* compared to estimated sizes in statecharts

4.4.3 Linux

4.4.3.1 Specific Parameters

Linux can be configured and compiled in many ways. In order to deal with an unambiguously defined system, a few parameters have to be set. The following extract from the global configuration file shows the settings used for this thesis:

```
CONFIG_X86_64=y                 #Intel Pentium 4 CPU 64 bit
CONFIG_SMP=y                    #Intel SMP specification
CONFIG_PREEMPT_NONE=y           #Preemption is deactivated for now
CONFIG_NR_CPUS=4                #Probe for max. 4 CPUs, no hotplug
CONFIG_HPET_TIMER=y             #Use the high precision event timer
CONFIG_GENERIC_HARDIRQS=y       #Allow for generic arbitration of IRQs
CONFIG_GENERIC_PENDING_IRQ=y    #Enable pending IRQ lines
```

4.4.3.2 Further Classification of Events in a Linux Model

All interrupts caused by the I/O devices $\iota = \{HDD, MPEG, HID, SND, NIC, KB\}$ are used in the Linux model exactly as depicted in the event tree Figure 4.8: $DEVT_{x,INT,I/O,\iota}$ with x being the CPU number which is processed by the CTSC. The agglomeration $DEVT_{x,INT,I/O,\iota}$ is possible because the device drivers – which are different of course – are all invoked in the same way. The interrupts between two ore more CPUs build up the class $DEVT_{x,INT,IPI,\theta}$ as defined in the event tree. In Linux 2.6, only three different IPIs $\theta = \{0, 1, 2\}$ are covered, the implementation is very basic. Deviating from the default event tree, two distinct sources for timer interrupts are embodied in the Linux models: local timer interrupts $DEVT_{x,INT,LOC\_TIMER}$ and global timer interrupts $DEVT_{x,INT,GLOB\_TIMER}$ handled by one dedicated CPU.

In the model, all software generated exceptions but the double fault ($DEVT_{x,EXC,DF}$) are treated equally. They are covered by the disruptive events $DEVT_{x,EXC,HD,\eta}$ with $\eta \in \{0, 3 \dots 13, 16 \dots 19\}$ which correspond to the set of the 16 exceptions defined by the IA32 (see Section 4.1.1). The double fault exception is handled in a different way.

The selected hardware platform comes with the special instructions `syscall` and `sysret` for efficient handling of system calls (see [Int08a] for more details). Older Linux kernels[21] realize all system calls by means of the dedicated interrupt vector `int 0x80` and a C function `system_call()` as registered handler for the interrupt. This function uses the exception number passed to branch to the corresponding routine.
To achieve downwards compatibility, an abstraction layer called `vsyscall` was implemented in the kernel. In this thesis, the `syscall - sysret` mechanism is used and the abstraction layer is thus disregarded. Note that the `execve()` system call is under every circumstance handled by means of an interruption vector since all registers have to be saved properly for this call anyway.
All system calls are denoted as given with $DEVT_{x,SYSCALL,\sigma}$ with $\sigma \in [0 - 288]$. Linux implements 289 system calls (Linux 2.6.11). The kernel macro `NR_syscalls` limits that number at compile time. However, additional system calls might exist. The kernel version of the component extension (Linux/CE) [Mül08] for example adds 13 additional system calls to the kernel.

4.4.3.3 Top Level Model of the Linux Kernel

The top level model of Linux is given in Figure 4.13. The two top level Cartesian Transition Set Connectors CU and CK connect the two main groups of the Linux operating system: KERNEL_LEVEL and USER_LEVEL.

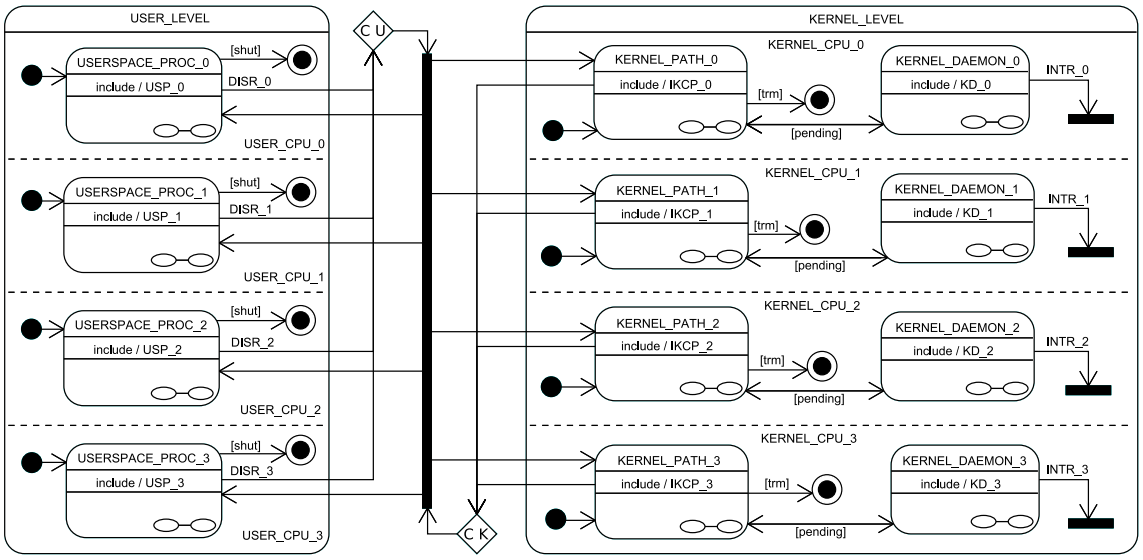


Figure 4.13: Linux 2.6 top level model

The rest of the model is straightforward: when a disruption $DEVT_{x,\{INT,EXC,SYS\}}$ occurs

[21] and modern kernel versions compiled for old CPUs

on CPU x, the user space process is abandoned and the system starts the corresponding kernel control path. When soft interrupts are pending, i.e. [pending] is true, the kernel daemon is launched. Linux can process interrupts (INTR_0, ... INTR_3) while executing the kernel daemon. The two artificial guards [shut] and [trm] are used to ensure that the corresponding states are not terminated unintentionally.
Table 4.7 lists all top level transitions. For typesetting reasons, the null action μ is not listed in this transition table.

ID	Source	Target	Event/Action	Description
DISR_0	USP_0	IKCP_0	$DEVT_{0,\{INT,EXC,SYS\}}$	An interruption of the process on CPU 0
DISR_1	USP_1	IKCP_1	$DEVT_{1,\{INT,EXC,SYS\}}$	An interruption of the process on CPU 1
DISR_2	USP_2	IKCP_2	$DEVT_{2,\{INT,EXC,SYS\}}$	An interruption of the process on CPU 2
DISR_3	USP_3	IKCP_3	$DEVT_{3,\{INT,EXC,SYS\}}$	An interruption of the process on CPU 3
INTR_0	KD_0	IKCP_0	$DEVT_{0,\{INT\}}$	An interrupt occurring on CPU 0
INTR_1	KD_1	IKCP_1	$DEVT_{1,\{INT\}}$	An interrupt occurring on CPU 1
INTR_2	KD_2	IKCP_2	$DEVT_{2,\{INT\}}$	An interrupt occurring on CPU 2
INTR_3	KD_3	IKCP_3	$DEVT_{3,\{INT\}}$	An interrupt occurring on CPU 3

Table 4.7: Transitions in the Linux top level model

4.4.3.4 The Interrupt Handling Facility Model

In the Linux top level model, a dedicated interrupt kernel control path is given by UML 2.0 submachine include for every CPU. Within each such include, an artificial grouping state IKCP is invented to bind the handling of exceptions, system calls and exceptions.
The split pseudo-state SP1 reacts to the second branch identifier of a delivered path event which is, according to Section 4.2.3.1 one of $\{INT, EXC, SYSCALL\}$. Exceptions are then processed further within the substate EXC, the processing of interruptions is modeled within INT and syscall servicing is elaborated further in the state SYSCALL.
Already on this very high level of modeling, the interruption which is possible in principle for an entire set of substates down through the hierarchy is given. Table 4.8 gives all complete labels and transitions of the Linux IKCP model.

4.4.3.4.1 Exceptions - EXC

The first state CPU_HANDLING in the Linux exception handling substate EXC represents the initial IA32(e) hardware handling. This handling comprises the following steps: The interrupt vector (in this case one of the numbers 0,3-14 or 16-19) is determined and the

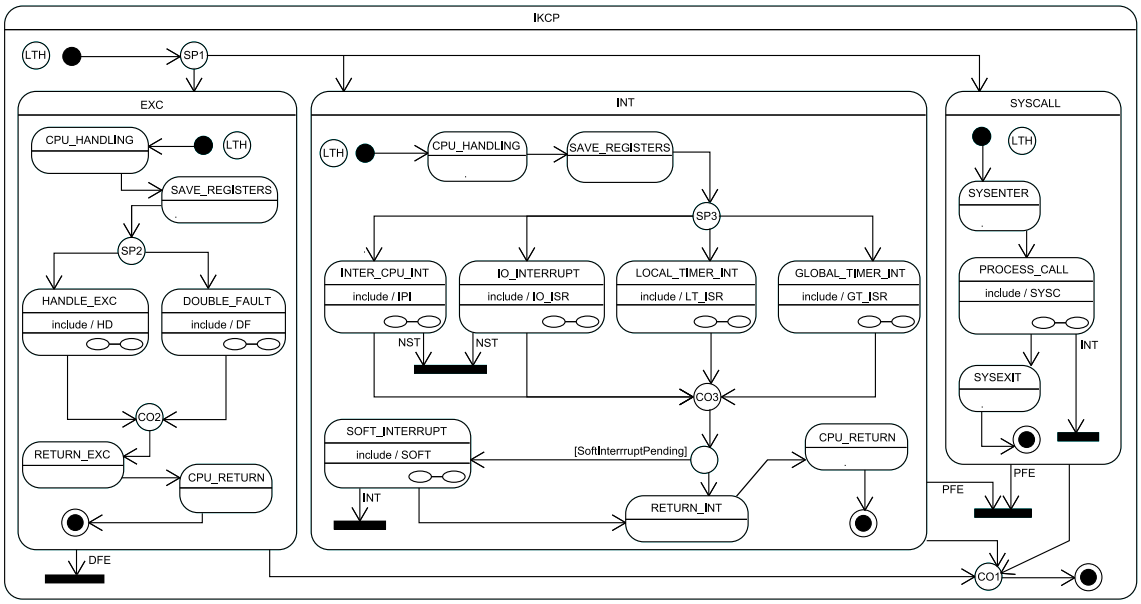


Figure 4.14: Linux 2.6 intermission kernel control path model

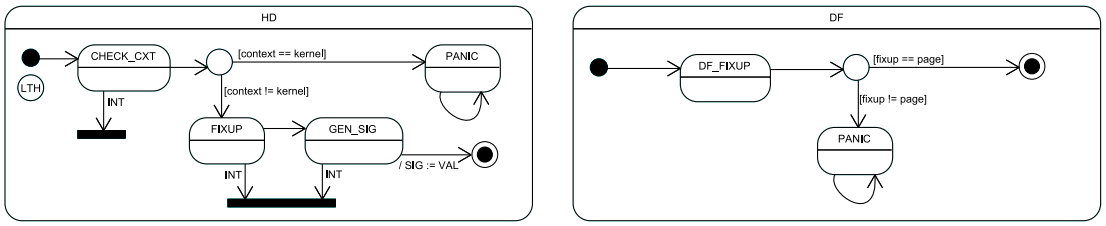
corresponding entry from the IDT is read. The new values of the registers `ss` and `esp` are loaded with respect to the exception which occurred and previous values are saved. Then, `cs` and `eip` are replaced with new values as well. Finally the hardware error code (if it exists) is pushed on the stack.
The first instructions executed by the operating system Linux itself when handling an exception are given in the state SAVE_REGISTERS. This code is implemented as an assembly language macro called `errorentry` which saves the number of the error in question, the general purpose registers (GPR) and branches to the registered handler. Two other dedicated macros `zeroentry` and `paranoidentry` are used by some special interrupt or exception handlers that need special care when saving the register contents.
The handling of the error itself is done by high-level C functions. For conventional exceptions, the corresponding state in the model is HANDLE_EXC, or more specifically the included state HD. Any exception handling routine must check whether the calling context was the kernel- or user space. Furthermore, according to the error, a fixup is processed if applicable and finally a signal is generated. The whole handler routine is interruptible. It is depicted in Figure 4.15(a). If the exception handler was called from kernel mode context, the kernel obviously suffers from errant code, a debug message, the "kernel oops" is printed and the system is terminated, i.e. it stays ultimately in state PANIC. The only legal exception which can be initially raised in kernel mode is the page fault exception. The double fault exception necessarily occurs in kernel mode as a result of an erroneous condition while handling an exception. The first activity taken by the (concrete) Linux system while in (abstract) state CHECK_CXT is to enable local interrupts, hence any exception handler in Linux can be disrupted by hardware interrupts.

The state DOUBLE_FAULT, detailed by its submachine state DF, represent exactly those unusual cases (see Figure 4.15(b)). When a double fault is raised, the kernel restores the registers `eip` and `esp` using the per-CPU task state segment (TSS) in previous ($<$ 2.6.18)

ID	Source	Target	Event/Action	Description
DFE	EXC	IKCP	$DEVT_{x,EXC,DF}/\mu$, x CPU number	Exception handler raising an exception - double fault
NST	IPI	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	IPI handler disrupted by interrupt - nesting
NST	IO	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	I/O handler disrupted by interrupt - nesting
INT	SOFT	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	SoftIRQ disrupted by interrupt
PFE	INT	IKCP	$DEVT_{x,EXC,HD,14}/\mu$, x CPU number	Any interrupt handler raises a page fault
PFE	SYSCALL	IKCP	$DEVT_{x,EXC,HD,14}/\mu$, x CPU number	System call routine raises a page fault
INT	SYSC	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	System call routine disrupted by any interrupt

Table 4.8: Transition table for $DEVT$s in the Linux IKCP

Linux versions. In newer versions, the double fault handler has its own double fault TSS (offset 31). Any double fault handler is initiated by the macro `paranoidentry`. In most cases it just prints debug-information and terminates the system. Nonetheless, if a fix is possible – for example when a faulty system call parameter raised the fault – the corresponding fix is processed. The main difference between the double fault handler and the other exception handlers is that this code is not interruptible under any circumstances.



(a) conventional exception handler in Linux (b) double fault handler in Linux

Figure 4.15: Exception handlers in Linux

In the model, the distinction between the two different exception handler types is made by means of the pseudo-states SP2 and CO2. The state RETURN_EXC represents a very complicated part of the source code that is executed when the systems returns from any interruption handling. If preemption is omitted, a check for the need to invoke the scheduler is performed. The decision to execute in the next step depends on whether a user process or a previously interrupted kernel path is to be continued. Finally, this step is then done in state CPU_RETURN. The instruction `iret` forces the processor to cease interrupt processing and return from the current context to the old one that was left.
Table 4.9 shows all transitions that can possibly be taken in submachine HD. Note that as an alternative to providing three distinct transitions, a grouping state as done for example in the OpenBSD models (see Figure 4.29) could be used.

ID	Source	Target	Event/Action	Description
INT	CHECK_CXT	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	EXC handler disrupted by an interrupt
INT	FIXUP	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	EXC handler disrupted by an interrupt
INT	GEN_SIG	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	EXC handler disrupted by an interrupt

Table 4.9: Labeled transitions for $DEVT$s in the exception handlers in Linux

4.4.3.4.2 Interrupts - INT

After the initial hardware handling state CPU_HANDLING which is the very same one used to handle exceptions, the contents of all relevant GPRs are saved in the next part of the interrupt handling facility. These saved GPRs are treated as a structure `pt_regs` called `*regs` in the subsequent processing. The corresponding state SAVE_REGISTERS is not interruptible since all local interrupt lines are still disabled.

The different classes of interrupts – inter-processor interrupts, I/O devices, and local as well as global timer interrupts – are modeled in different substates of INT. The split pseudo-state SP3 branches to the appropriate substate, one of IPI, IO_ISR, LT_ISR, GT_ISR in the model. Note that in this case, the names of the states reachable from the split state SP3 do not match the branch identifiers. The deviating values of the branch identifier assignment function $\mathfrak{bid}$ are therefore defined as:

$$\mathfrak{bid}(IO) \stackrel{\text{def}}{=} \text{IO\_ISR} \ , \ \mathfrak{bid}(LOC\_TIMER) \stackrel{\text{def}}{=} \text{LT\_ISR} \ , \ \mathfrak{bid}(GLOB\_TIMER) \stackrel{\text{def}}{=} \text{GT\_ISR}$$

4.4.3.4.2.1 Inter-Processor Interrupts - IPI

IPIs are implemented very basically in Linux, hence the submachine for state IPI is quite small and simple. Figure 4.16 shows the steps taken by the Linux kernel. First, due to performance reasons, only a reduced set of GPRs is saved to the stack in state SAVE_GPR. One of the interrupt vectors 251, 252 or 253 is pushed to the stack. Afterwards, the according C function is called – `smp_reschedule` (depicted by state DO_RESCHED), `smp_call_function` (depicted by state DO_CALL_FUN) or `smp_invalidate_tlb` (depicted by state DO_INV_TLB).
Since the choice of which IPI is to be handled is encoded as a path event, the pair of pseudo-states SP4 and CO4 models the branches. Similar to Table 4.9, Table 4.10 lists the detailed transitions of the inter-processor interrupt handling submachine.

4.4.3.4.2.2 I/O interrupts

Any interrupt (see Figure 4.17) issued by peripheral devices is first processed by the state DO_INT. This still critical (in terms of urgency) part of the IHF contains the following steps:

1. The function `do_IRQ()` first executes the macro `irq_enter` which increases the number of nested interrupts in the `preempt_counter` - field, see Section 4.4.3.5.

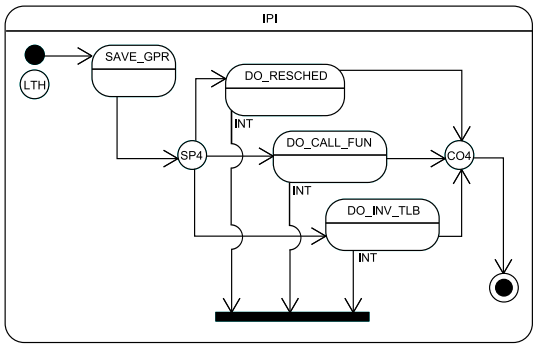


Figure 4.16: Linux 2.6 inter-processor interrupt handler

ID	Source	Target	Event/Action	Description
INT	DO_RESCHED	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	Rescheduling is disrupted by any interrupt
INT	DO_CALL_FUN	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	Remote function call is disrupted by any interrupt
INT	DO_INV_TLB	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	Invalidating the TLB disrupted by any interrupt

Table 4.10: Labeled transitions for $DEVT$s in the Linux IPI handling submachine

2. If necessary, the stack is switched to the hard IRQ stack. If the current interrupt is a nested one, this step is not needed, since the correct stack is already in use.

3. The `__do_IRQ()` function is called. It acknowledges the IRQ using the registered low level routines and marks the IRQ as pending.

4. If no other CPU is already executing the registered service routines, the function `handle_IRQ_event()` loops through all registered handler routines for the particular interrupt.

The remaining handling of interrupts raised by the external I/O devices and arbitrated via the IOAPIC is also modeled in the submachine IO_ISR as depicted by Figure 4.17. Table 4.11 shows the transitions.

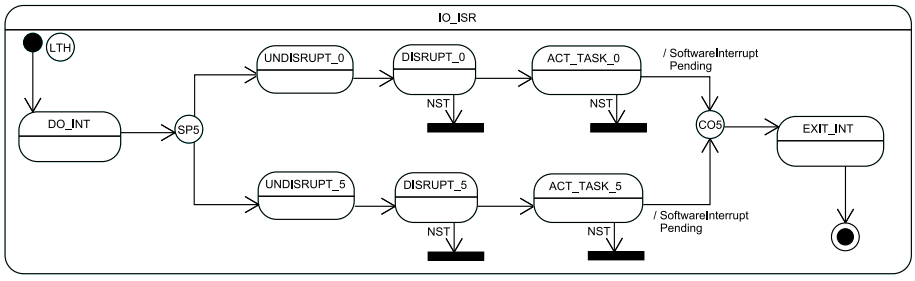


Figure 4.17: Linux 2.6 I/O interrupt handling ISR (simplified model, ISRs 1..4 analogous)

Without loss of generality we assume at this pint that all I/O interrupt service routines have the same basic structure:

1. Some critical parts – communication with the according hardware device may be one of these – have to be executed with interrupts being disabled, as processed in states UNDISRUPT_0 ... UNDISRUPT_5 of the model. If this is the first part of the handler, the flag `SA_INTERRUPT` is used to prevent the function `handle_IRQ_event` from enabling local interrupts at the beginning. This flag is part of the entry stored in the IDT and thus it is set during the initialization phase.

2. Some parts of a called ISR may run without any restrictions regarding interruptibility. These parts are represented by the states DISRUPT_0 ... DISRUPT_5.

3. Many uncritical and deferrable tasks may be swapped out to a tasklet, see Section 4.4.3.4.2.5. In states ACT_TASK_0 ... ACT_TASK_5, these tasklets are set active for execution.

The state EXIT_INT basically calls the assembly language macro `irq_exit` to decrease the counter for nested interrupts which is the companion operation to `irq_enter`.

ID	Source	Target	Event/Action	Description
NST	DISRUPT_n	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	Uncritical parts of an ISR are disrupted by any interrupt
NST	ACT_TASK_n	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	Tasklet activation is disrupted by any interrupt

Table 4.11: Labeled transitions for $DEVT$s in the submachine for device-triggered I/O interrupt handling; $n \in \{0..5\}$

4.4.3.4.2.3 Local timer interrupts

Timer interruptions issued by the the local APIC are processed in submachine LT_ISR. The initial (DO_INT) and final (EXIT_INT) handling do not differ from the methods described for I/O devices above. The following extract from the Linux kernel explains the whole process and shows that the acknowledgement is sent immediately. The reason for the immediate acknowledgement is that the runtime of the overall routine can be quite long.

```
void smp_apic_timer_interrupt(struct pt_regs *regs) {
 ack_APIC_irq();
 irq_enter();
 smp_local_timer_interrupt(regs);
 irq_exit();
}
```

Figure 4.18 shows the model of the entire handling routine `smp_local_timer_interrupt`. As mentioned above, the whole service routine was registered with the flag set `SA_INTERRUPT` and thus started with local interrupts disabled. In a nutshell, it performs the following steps:

1. The invocation of `profile_tick`, depicted by state PROF_TICK, realizes the profiler[22].

[22]Only if this facility is enabled during the boot-process.

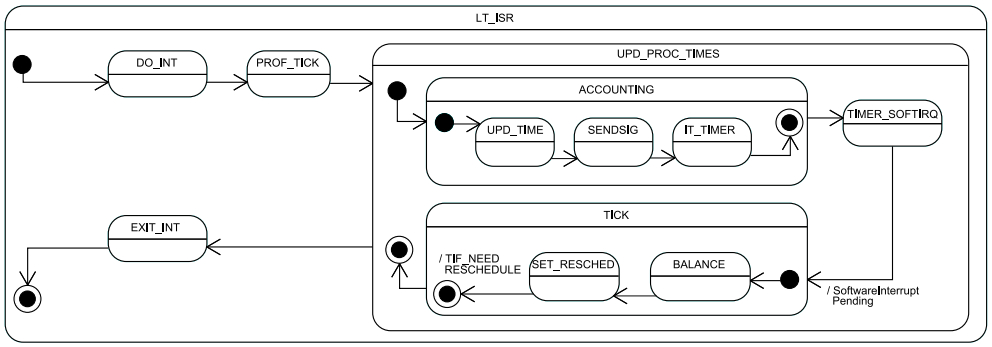


Figure 4.18: Linux 2.6 handler routine for the local timer interrupt

2. The state UPD_PROC_TIMES represents the function `update_process_times` which contains the following steps:

   (a) In state ACCOUNTING, the first activity performed is updating the time for the current process, UPD_TIME, and if its time quantum elapsed to send the according signals `SIGXCPU` and `SIGKILL` where necessary are send to the process in question (SENDSIG). If any interval timer is registered, it is checked separately from the conventional timers in state IT_TIMER.

   (b) The system timers in Linux are implemented as deferrable functions, i.e. as software interrupts. A software IRQ that corresponds to a system timer is marked as runnable in state TIMER_SOFTIRQ.

   (c) Afterwards, in TICK which corresponds to the kernel function `scheduler_tick`, the CPU load between scheduling domains is re-balanced (BALANCE) and the scheduler is marked to execute (lazy invocation) (SET_RESCHED).

4.4.3.4.2.4 Global Timer Interrupts

The global timer interrupt raised by the external hardware circuits like the PIT or HPET (see Sections 4.1.1 and 4.1.2) are handled in submachine GT_ISR which is included by the submachine state GLOBAL_TIMER_INT (see Figure 4.14).

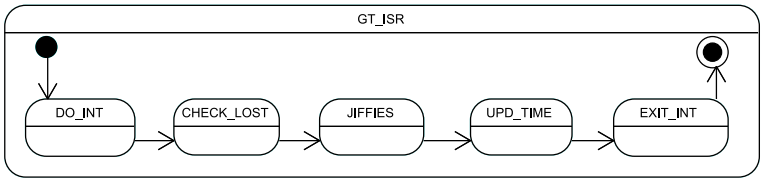


Figure 4.19: Linux 2.6 handling routine for the global timer interrupt

In multi-processor systems, the corresponding handling routine `timer_interrupt` is very simple as most issues related to timekeeping are CPU-specific and thus conducted by local routines. The few actions taken by the handler are shown in Figure 4.19. First, a hardware-dependent part checks whether timer interrupts have been lost since the last tick (state CHECK_LOST), then the variable `jiffies64` – the tick counter – is incremented and finally the `update_time` function represented by state UPD_TIME in the model calculates the system load and updates the system internal time representation, called “wall time”.

4.4.3.4.2.5 Software IRQs

Not only the system timers can be deferred as mentioned above, but also other interrupt service routines that are expected to take a long time to completion can be executed after the immediate interrupt handling.
In Linux, this concept was once called "bottom half", but since the 2.6 kernel, this mechanism has been enhanced and renamed Software IRQ (SoftIRQ). A SoftIRQ can be raised by any interrupt handler as well as by any other code with kernel space privileges.
Linux implements six SoftIRQs, the ones with lowest (5) and highest (0) priority allow new deferrable functions to be implemented on top of them. These new deferrable functions are called tasklets. The four remaining ones represent the former bottom halves for the timers (1), network transmission (2) and reception (3) and finally for SCSI command processing (4). Thus any I/O device driver having some deferrable work to do has to register either a high or a low priority Tasklet.

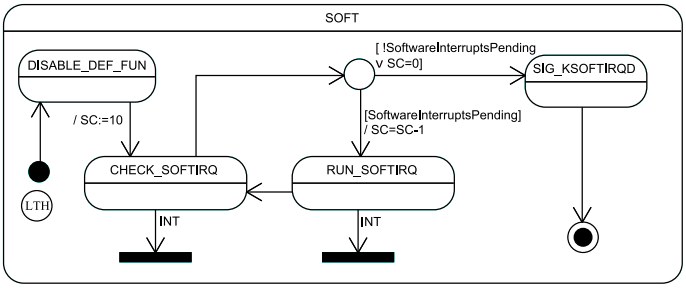


Figure 4.20: Linux SoftIRQ handler

The handling of SoftIRQs is done by means of the C function `do_softirq()` that basically performs the following steps:

1. In state DISABLE_DEF_FUN, at first all other deferrable functions are marked as deactivated to permit interleaving of SoftIRQ handlers. Local interrupts are reactivated at the end of the handling.

2. During the checking for and execution of pending SoftIRQs, CHECK_SOFTIRQ and RUN_SOFTIRQ can be re-entered at least ten times. After that number of repetitions, a counter variable leads straight to the next step.

3. If after ten iterations there are still SoftIRQs pending, the SoftIRQ handler awakes the local kernel daemon for handling pending software IRQS called `ksoftirqd/<cpu>`, this awakening is performed by state SIG_KSOFTIRQD.

The process `ksoftirqd/<cpu>` runs with low priority and thus ensures that user mode processes can run although high traffic network load for example is processed by the corresponding SoftIRQs. Table 4.12 lists the detailed transitions.

4.4.3.4.3 System Calls - SYSCALL

The state SYSENTER in Figure 4.14 is the abstraction of the instruction `syscall` and its processor-internal processing mentioned above. Although local interrupts are physically enabled during the execution, theoretically allowing the processing to be interrupted by any external stimulus, no such transitions are introduced in the Linux model. The reason

ID	Source	Target	Event/Action	Description
INT	CHECK_SOFTIRQ	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	Checking is disrupted by any interrupt
INT	RUN_SOFTIRQ	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	SoftIRQ is disrupted by any interrupt

Table 4.12: Transitions for $DEVT$s in the SoftIRQ handler

for that is the fact that the overhead and the instruction time of `syscall` and also `sysret` are negligibly small and the disruptive event causes the abortion of the current control flow after completion. For that reason, the following state SYSC is modeled adequately, i.e. interruptible from the very beginning. However, if some critical code is to be processed, the routine has to lock interrupts by using the assembly language instruction pair `cli` and `sti`. SYSEXIT is the logical counterpart of SYSENTER. Although these two states enclose privileged code execution, underlying machine instructions are not considered a call/return pair but only companion instructions [Int05].

4.4.3.5 Preemptive IKCPs

In the next step, a preemptive kernel is considered and the changes in the model elaborated accordingly. Although preemption is not considered relevant for a MOSRTOS system (compare Section 5.2.1), a close investigation of the differences is insightful.

In 2001, initial efforts were made to reduce worst case latency scenarios in Linux 2.4 by introducing preemption points into long critical regions. The Montavista preemption patch by Molnar [Mol02] established a basis for the improvements of the Linux 2.6 kernel series. Today, Linux can be compiled in three ways:

1. Non-preemptive, as considered before: `CONFIG_PREEMPT_NONE` is set in the configuration during the build process. This is the traditional Linux preemption model; the system is optimized for high throughput.

2. In a voluntary preemption mode, i.e. some preemption points are introduced into the kernel, but in general, the kernel code does not relinquish the CPU. The flag `CONFIG_PREEMPT_VOLUNTARY` has to be set before building the system.

3. Fully preemptive, i.e. all kernel code except for a few critical sections can be interleaved with other kernel code. To achieve this behavior, Linux has to be built with the `CONFIG_PREEMPT` option enabled.

In addition to this, the "Big Kernel Lock" (BKL) – which is used to lock the whole kernel and is still used in some old drivers – can be configured preemptive by setting `CONFIG_PREEMPT_BKL = y` before compiling as of Linux 2.6.11. In the real-time Linux community, efforts to improve preemption, for example in the IRQ subsystem, are still underway. See e.g. [DW05] for details on current issues.

With the chosen configuration, kernel preemption is allowed by default unless a special field in the process descriptor prohibits this. This field is called `preempt_count` and contains the following data:

1. The preemption counter, bits 0-7, that counts how often preemption was explicitly disabled for the currently running thread.

2. The SoftIRQ counter , bits 8-15, that counts the level of SoftIRQs being disabled. Hence, SoftIRQs can only be executed if this field is equal to 0.

3. The nested interrupts counter (referring only to hardware IRQs), bits 16-27[23] that holds the depth of nested interrupt handlers on the current CPU. This value is incremented or decremented by the macros `irq_enter` and `irq_exit` respectively as described above.

4. The flag `PREEMPT_ACTIVE` indicating whether preemption is active or not.

When dealing with a preemptive Linux kernel, any thread running in kernel mode could be rescheduled by a forced task switch at any time without an asynchronous interrupt from an external source. Thus, critical sections in Linux can be guarded by disabling local interrupts to ensure that nothing can force the current thread to release the CPU unless it does so voluntarily.

If the preemption property of one state cannot be given unambiguously because it changes during the execution of the state, this state has to be subdivided into multiple sequential states.

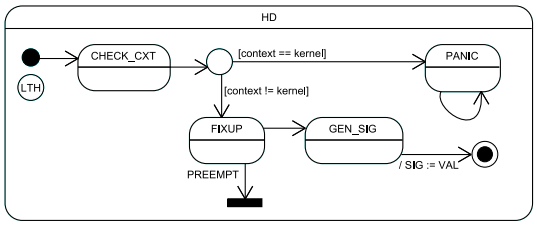


Figure 4.21: Conventional exception handler in Linux, preemptive parts only

Figure 4.21 shows that only one part of the exception handler routine allows for preemption, whereas the whole handler allows for local interrupts.

Two of three basic inter-processor interrupt handlers (see Figure 4.22) are implemented preemptive: only the TLB invalidation routine is not preemptive for reasons of performance optimization. However, it could easily be implemented as preemptive, but the average duration of the corresponding code is too short: it is only a few assembly language instructions. In the special case that the inter-processor interrupt `DO_CALL_FUN` is sent to all CPUs, the preemption is also disabled by a wrapper function.

[23] although values range from 0 to 4095, the IA32(e) specification limits that range.

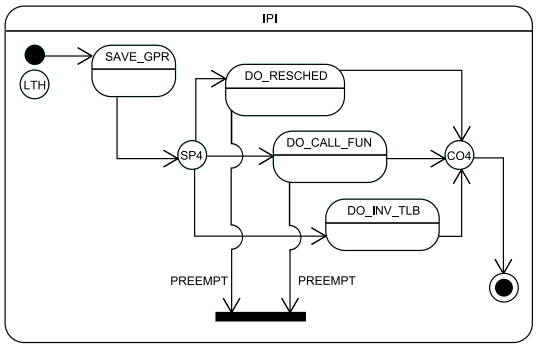


Figure 4.22: Linux 2.6 inter-processor interrupt handler, preemptive parts only

As done before, several assumptions concerning the structure of an I/O device ISR are made. It is now assumed that the noncritical part contains components with data access to critical structures and thus kernel preemption is not allowed – states DISRUPT_0 ... DISRUPT_5 in Figure 4.23 represent this. In addition to those states, some parts of the handler will access own structures only and will thus be preemptive. Their states DISRUPT_PREEMPT_0 ... DISRUPT_PREEMPT_5 depict this part.

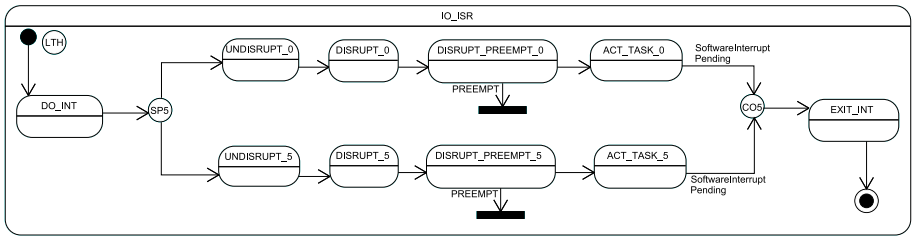


Figure 4.23: Linux 2.6 I/O interrupt handling ISR, preemptive parts only (simplified model, ISRs 1..4 analogous)

Note that Figures 4.21, 4.22 and 4.23 only show the preemptive parts of the routines, the transition tables for those model parts remain unaltered.

4.4.3.5.1 Returning from any Exception or Interrupt Handler

As already mentioned in Section 4.4.3.4, within a preemptive kernel, the return from an interruption or exception handler is much more complex than it was before. The corresponding assembly language code is rather complicated. Figure 4.24 is derived from [BC06], page 185 and shows this control flow when preemption is considered.

4.4.4 OpenBSD

The OpenBSD IHF model created by Gogolok [Gog07] is not completely elaborated in this section. The model is based on version 4.0, released in November 2006.

The classification of events is very close to the universal classification for disruptive events as given in Section 4.2.3.1, i.e. a disrupting event has the form $DEVT_{x,k,t,i}$ with x being the CPU number relevant to the top level CTSC. The set of interrupts caused by I/O

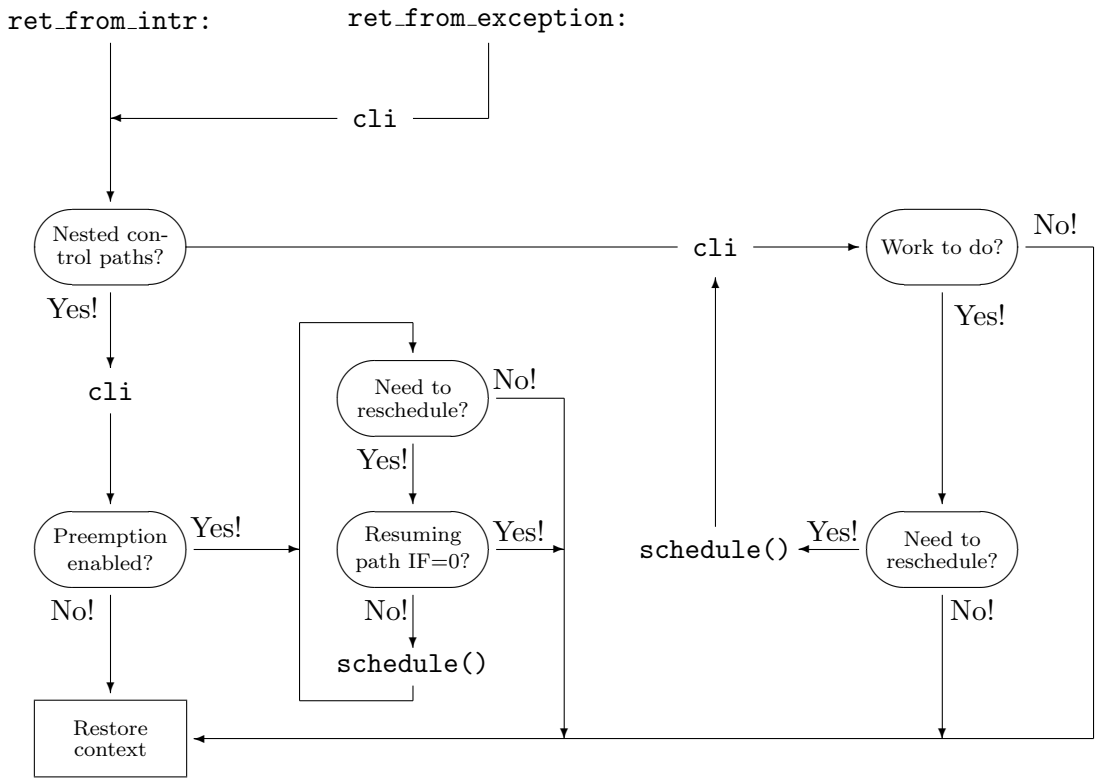


Figure 4.24: Linux 2.6 return from interrupts and exceptions

devices is defined as in the Linux model $\iota = \{HDD, MPEG, HID, SND, NIC, KB\}$. The only deviations from this are that on the level of $DEVT_{x,INT}$, there is the additional class of software inter-processor interrupts $t = SOFT\_IPI$ and exceptions are further categorized as follows:
In the model, all software-generated exceptions other than 7 and 16 (i.e. the device not available (DNA) and the floating point errors (FPU)) are covered by the events $DEVT_{x,EXC,\eta}$ with $\eta \in \{0, 3 \ldots 6, 8 \ldots 14, 17 \ldots 19\}$

OpenBSD supports the following inter processor interrupts : $DEVT_{x,INT,IPI,\iota}$ with $\iota \in \{HLT, SET, TLB, FLUSH, SYNC, DB\}$. Their different handling is not elaborated further. Since the same 4-CPU setup is considered, the OpenBSD top level model is quite similar to that of Linux.

4.4.4.1 The Interrupt Handling Facility Model

On the level of the IKCPs, either interrupt, exception or system call processing takes place (see Figure 4.25). The split/combine pseudo-state pair SP1/CO1 reacts to the first branch identifier $k \in \{INT, EXC, SYSCALL\}$.
On this modeling level, all control paths can only be interrupted when an exception occurs. Table 4.13 lists all interruptions which can possibly occur while the system is in state IKCP.

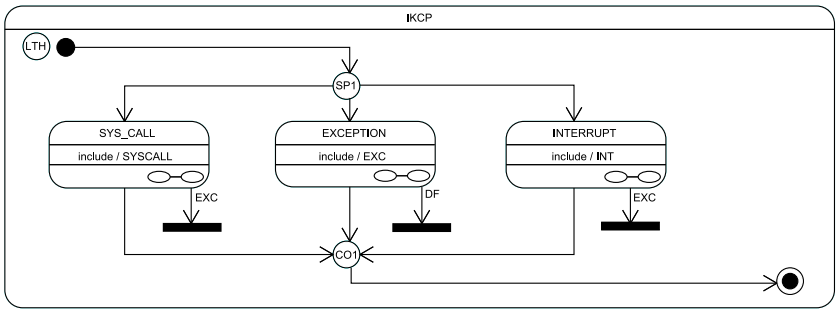


Figure 4.25: OpenBSD intermission kernel control path model

ID	Source	Target	Event/Action	Description
EXC	SYSCALL	IKCP	$DEVT_{x,EXC}/\mu$, x CPU number	System call handler raising any exception
DF	EXC	IKCP	$DEVT_{x,EXC,8}/\mu$, x CPU number	Exception handler raising an exception - double-fault
EXC	INT	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	Interrupt handler raising any exception

Table 4.13: Transition table for $DEVT$s in the OpenBSD IKCP

4.4.4.1.1 System Calls - SYSCALL

Contrary to case of system calls in the Linux operating system, in OpenBSD those are implemented by means of software interrupts. The vector $80_{hex} = 128_{dec}$ is used to jump to the registered system call routine. Figure 4.26 shows the submachine that handles system calls: SYSCALL.

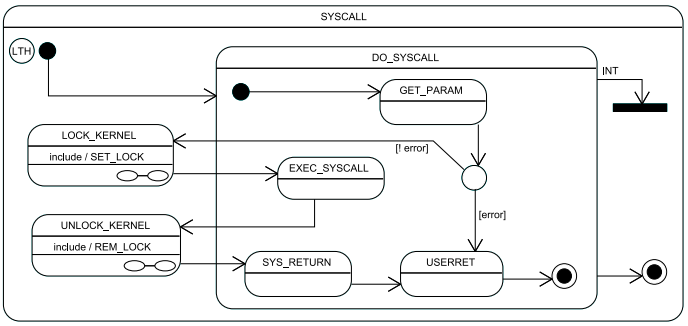


Figure 4.26: OpenBSD system call handling

The initial handling depicted by state GET_PARAM basically fetches the number of the system call[24] issued and retrieves the parameters required to service the call. The serv-

[24]OpenBSD implements more than 300 system calls. A comprehensive list and universal classification can be found in [Gog07], appendix A.

ing itself is carried out while a global kernel lock (see Section 4.4.4.2) (states SET_LOCK and REM_LOCK) protects the critical region (state EXEC_SYSCALL). The postprocessing, i.e. pushing necessary values to the user mode stack etc. is carried out in the two states SYS_RETURN and USERRET. Note that if the preliminary check for valid parameters fails, the handler routine is immediately abandoned.

Since all parts other than the kernel locking operations are interruptible, a grouping state DO_SYSCALL is introduced. Table 4.14 describes the outgoing transition in detail.

ID	Source	Target	Event/Action	Description
INT	DO_SYSCALL	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	System call handler disrupted by any interrupt

Table 4.14: Transition table for $DEVT$s in the OpenBSD system call handler

4.4.4.1.2 Exceptions - EXC

Exception handlers are grouped as described by the address spaces they operate in. The DNA and FPU handlers operate in any address space (states DNA and FPU), the remaining ones are handled by means of the `trap` function (state TRAP) The SP2 and CO2 pseudo-states model the corresponding branches. Figure 4.27 shows the entire submachine EXC.

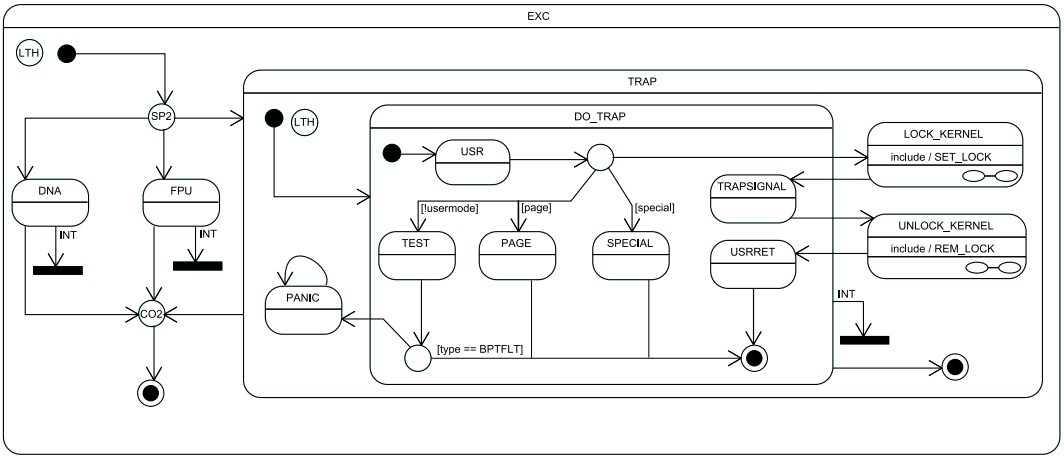


Figure 4.27: OpenBSD exception handling

Again, for the majority of the handlers (TRAPSIGNAL followed by USERRET), the kernel must be locked (states SET_LOCK and REM_LOCK). The special exceptions (SPECIAL)and the page fault exception (PAGE) do not need this precaution. If an exception that is not allowed to occur in kernel mode is sensed (TEST) outside user mode, the kernel enters the state (PANIC) and remains stuck.
Since all parts other than the kernel locking operations are interruptible, a grouping state DO_TRAP is introduced once more. Table 4.15 describes the transitions.

4.4.4.1.3 Interrupts - INT

Different from the Linux model, the next level of branching for hardware interrupts is

ID	Source	Target	Event/Action	Description
INT	DNA	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	DNA exception handler disrupted by any interrupt
INT	FPU	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	FPU exception handler disrupted by any interrupt
INT	DO_TRAP	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	Generic exception handler disrupted by any interrupt

Table 4.15: Transition table for $DEVT$s in the OpenBSD system call handler

modeled by means of a submachine include instead of simply using nested substates. Figure 4.28 shows the submachine INT.
The initial saving of the registers (called frame creation in OpenBSD) including the steps carried out by the CPU itself are modeled as state CREATE_FRAME. The interrupt post-processing also includes various checks for software interrupts (as in Linux). It is modeled in submachine INT_RETURN. The details about this part of the model are omitted in this thesis, but can be found in [Gog07]. The same applies for the inter-processor handlers IPI and SOFT_IPI modeled as black boxes.

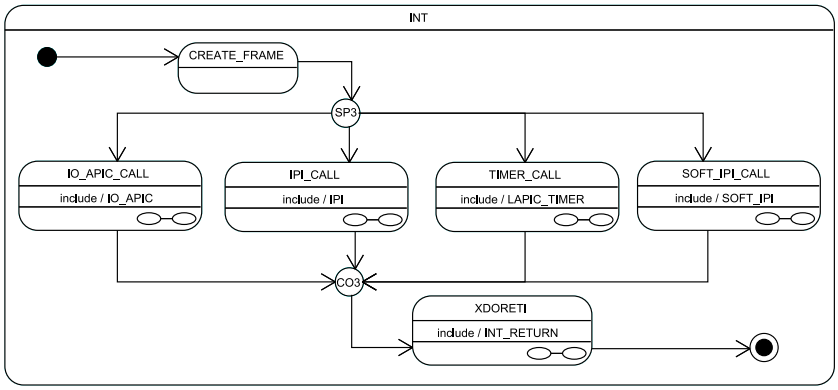


Figure 4.28: OpenBSD IHF details for the interrupt-branch

4.4.4.1.3.1 IOAPIC Interrupts - IO_APIC

All interrupts generated by peripheral devices are treated equally in the model (see Figure 4.29). For any registered handler that is detected (states DET_HANDLER and DET_NEXT), the kernel is locked, the handler routine invoked (state IOAPIC_HDL) and the usage statistics are updated (state UPDATE_STAT). If no handler is registered, an error message is passed to a kernel debugger if any, depicted by state STRAY_INTERRUPT. The flag `end of interrupt` is finally set in state EOI.
A part of an interrupt handler can be only disrupted by interrupts with certain priorities, as OpenBSD implements the Intel hardware priority schema. Table 4.16 lists these possible interruptions.

4.4.4.1.3.2 Local Timer Interrupt - LAPIC_TIMER

The distinct handler for the timer interrupt is modeled by submachine LAPIC_TIMER.

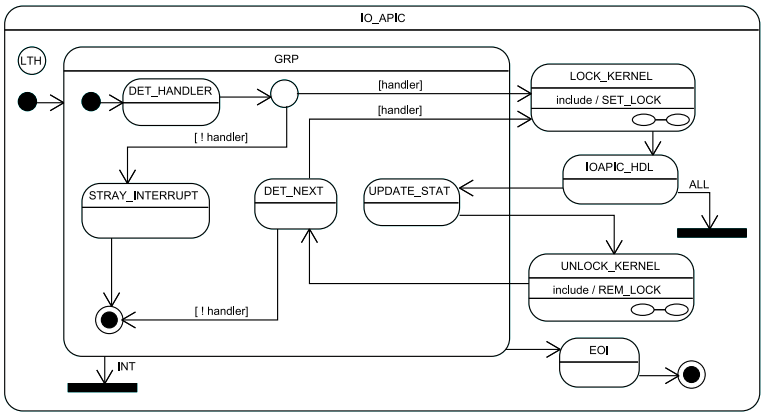


Figure 4.29: OpenBSD I/O interrupt handling

ID	Source	Target	Event/Action	Description
INT	GRP	IKCP	$DEVT_{x,INT,\iota>cur}$, x CPU number	Generic handling disrupted by interrupt with higher priotity
INT	IOAPIC_HDL	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	service routine disrupted by any interrupt

Table 4.16: Transition table for $DEVT$s in the OpenBSD I/O interrupt handler

Contrary to the Linux operating system, the routines for the local and the global timer interrupts are the same.

The interrupt is acknowledged immediately in state EOI and then – with the kernel locked – the hardware-based system clock is accessed. If the interrupt was issued to the primary CPU, i.e. the interrupt represents the global timer interrupt, the TSC (see Section 4.1.1) is read and saved to calculate the system's wall time.
Within the state HARDCLOCK, a series of important tasks is performed:

1. The interval timers (each process in OpenBSD owns up to three of these) are checked to see whether a configured interval has elapsed. If so, the corresponding signal is sent. This is modeled in the state INTERVAL_TIMER.
2. Several data structures containing usage and load statistics are updated every clock-tick (state UPDATE_STAT).
3. Every 4 ticks (i.e. 40 ms), the priority of the running processes is calculated as represented by state PRPRIO.
4. Every 10 ticks (RR_ACT), the scheduler modeled by state RESCHED is invoked.
5. Only on the primary CPU, the process queues are updated (state UPD_QUEUE).
6. If existing, a software timer is registered for the scheduled process (state SETSOFT-CLOCK).

Figure 4.30 shows two possible interruptions of a local APIC timer routine, Table 4.17 lists them.

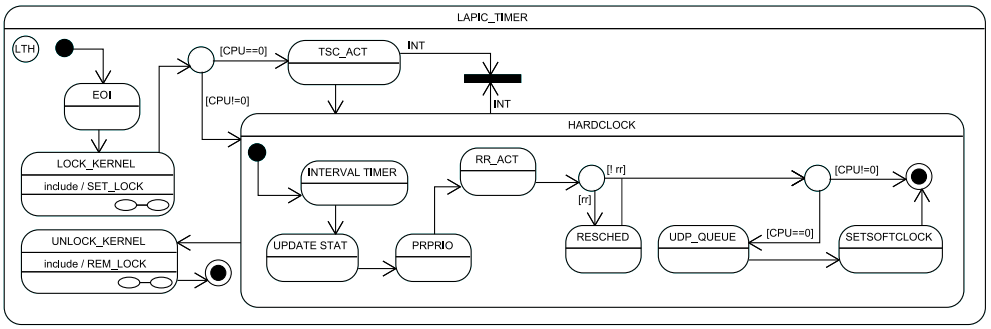


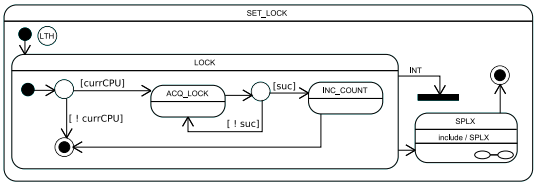
Figure 4.30: OpenBSD timer interrupt

ID	Source	Target	Event/Action	Description
INT	TSC_ACT	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	Primary timer handling disrupted by any interrupt
INT	HARDCLOCK	IKCP	$DEVT_{x,INT}/\mu$, x CPU number	Secondary timer handling disrupted by any interrupt

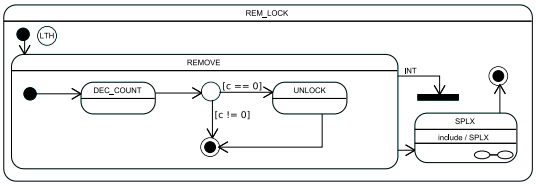
Table 4.17: Transition table for $DEVT$s in the OpenBSD timer interrupt handler

4.4.4.2 The Global Kernel Lock

Most kernel routines require the kernel to be locked by some sort of global spin-lock. This mechanism is very much the same as the Linux big kernel lock abandoned a long time ago by the community due to performance impacts. However, this coarse-grained synchronization means is widely used in OpenBSD. Figures 4.31(a) and 4.31(b) show both the locking and unlocking operations as implemented in the kernel sources. Note that their execution is basically interruptible (transition INT).



(a) Kernel lock operation in OpenBSD



(b) Kernel unlock operation in OpenBSD

Figure 4.31: OpenBSD kernel locking and unlocking operations

The remaining submachine SPLX which is not elaborated in this theses is entirely interruptible. The corresponding kernel function `splx` checks every time a system priority level changes, whether a registered software interrupt must be executed and, if so, invokes it. This very functionality is also implemented in the XDORETI submachine in Figure 4.28.

4.4.5 L4Ka::Pistachio

As the OpenBSD model, Pistachio is not entirely elaborated in this thesis. Since it does not serve the goal of this thesis, the implementation details of the system calls such as IPC or the page fault are not considered. In contrast to the other operating systems investigated so far, a model of the system scheduler is regarded.

4.4.5.1 Further Classification of Events in a Pistachio Model

All interrupts caused by the I/O devices $\iota = \{HDD, MPEG, HID, SND, NIC, KB\}$ are also used in the Pistachio model exactly as in the other models. Inter-processor interrupts are called cross-CPU mailbox mechanism in Pistachio.
Since exception handling is implemented only in a very rudimentary way in the experimental operating system Pistachio, it is not elaborated in the model, and thus no disruptive events underneath the *EXC* category are defined here. For a very detailed analysis, see Wieder [Wie07].

System calls are vital for any micro-kernel operating system. Pistachio distinguishes between six non-privileged system calls, four privileged ones and the IPC calls for sending and receiving messages (see Section 4.3.1). Their technical implementation is not part of the IHF model either.
In contrast to the Linux and OpenBSD models, the Pistachio IHF model also makes use of a different class of other events than disruptive ones: events to broadcast signals and scheduling actions across the model (see Section 4.2.3.1). These events are generated by the action mechanism of statecharts and will be elaborated when they are used within the model.

4.4.5.2 Pistachio Architectural Model

The architecture of a μ-kernel system differs considerably from the other systems considered so far. Figure 4.32 shows the top level for one CPU, i.e. for one path of concurrency only.

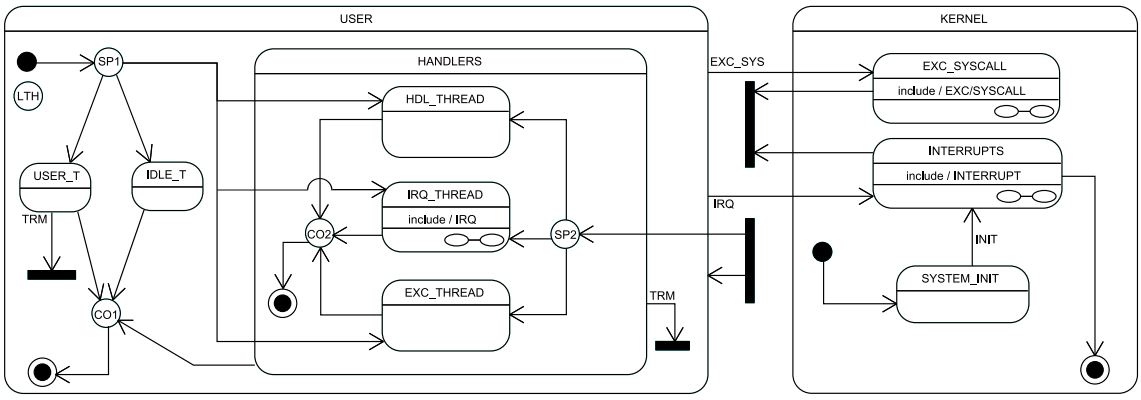


Figure 4.32: Pistachio micro-kernel architectural overview

A top level model with four concurrent paths is very similar to the one provided for the Linux kernel, a four-fold *AND* super state is used to incorporate each upmost state, USER

as well as KERNEL.
While a kernel mode handling of interrupts, exceptions and system calls is represented by the two submachine states INTERRUPT and EXC/SYSCALL just as in the non-μ-kernel system models, in this system even the user space contains several handler threads (grouped by the superstate HANDLERS). This characteristic element of a μ-kernel system will be investigated later in Section 5.4.
The user model thus contains the user mode threads USER_T, a representation of the idle thread IDLE_T as well as the handler threads for hardware interrupts HDL_THREAD, the actual worker thread for each registered interrupt IRQ_THREAD and the handler routines for exceptions EXC_THREAD.
Depending on a generated event representing a signal or a scheduling decision, either the pseudo-state pair SP1/CO1 or SP2/CO2 carries out the branching.
Note that the sequential transition t with $source(t) =$ HANDLERS and $target(t) =$ CO1 is not entirely correct with respect to the *ESF* modeling rules. In the actual model, it is cloned and each clone then originates from one of the three handler substates. The deviation was only made in the figure to improve its readability.

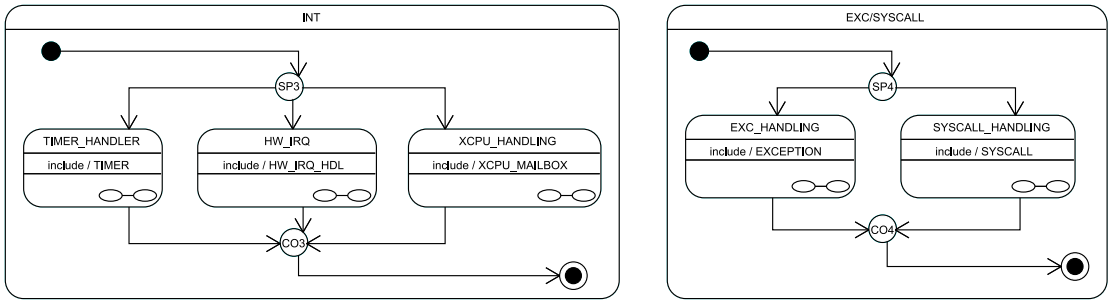
The transitions in this model are provided in Table 4.18. In all following tables, events related to signals are abbreviated as $SIGEVT$ and events corresponding to scheduling decisions as $SCHEVT$.

ID	Source	Target	Event/Action	Description
IRQ	USER	INTERRUPT	$DEVT_{x,INT}$, with x CPU number	An interrupt occurred while in user mode
IRQ	USER	EXC/SYSCALL	$DEVT_{x,\{EXC,SYS\}}$, with x CPU number	An exception occurred or syscall was issued.
TRM	USER_T	USER	$SIGEVT_i$, with i thread ID	User thread terminated by signal
TRM	HANDLER	USER	$SIGEVT_i$, with i thread ID	Handler thread terminated by signal

Table 4.18: Transitions in the Pistachio top level model

4.4.5.3 Pistachio Intermission Kernel Control Path

The kernel-level handling of interrupts differs greatly from the handling approaches investigated so far: no service routine is executed directly in kernel space. One of the driving paradigms of the research is to treat the concrete handler of a dedicated device as a black box and to make a few assumptions about its behavior. Doing so does not at all influence the model of the IHF located in the kernel since these simplifications only affect the handler threads in user space.
The IKCPs in the Pistachio model are structured quite like those in the Linux and OpenBSD models. Figures 4.33(a) and 4.33(b) show the subsequent branching (pseudo-state pairs SP3/CO3 and SP4/CO4).
At this level, no finer distinction of interruptibility is provided, i.e. there are no outgoing transitions in the submachines. As mentioned above, the inter-process interrupts are



(a) interrupt submachine in Pistachio

(b) system call and exception submachine in Pistachio

Figure 4.33: Pistachio hardware intermission kernel control paths

not elaborated further. Nonetheless, the following modification of the branch identifier function is needed to correctly process path events:

$$\mathfrak{bid}(IPI) \stackrel{\text{def}}{=} \mathsf{XCPU\_MAILBOX}$$

The exception (EXCEPTION) and system call (SYSCALL) submachines are also not elaborated further. Details about those black boxes can be found in [Wie07].

4.4.5.3.1 I/O interrupts - HW_IRQ_HANDLER

The kernel mode I/O interrupt handling is kept small and its focus is merely on delegating the work to the appropriate processes. The Pistachio kernel offers a software masking mechanism for interrupts. The kernel functions `mask()` and `unmask()` are implemented as high-level software equivalents to the machine instructions `cli` and `sti`.

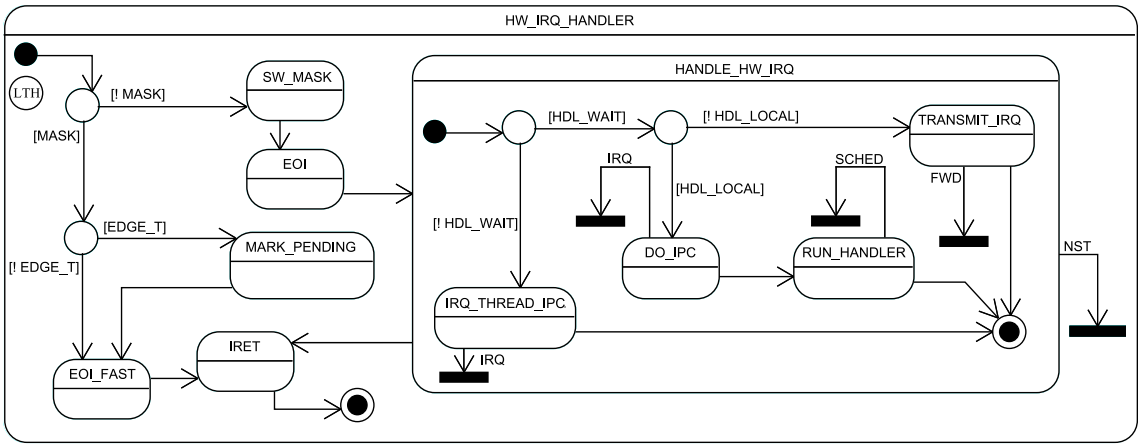


Figure 4.34: Pistachio I/O interrupt, kernel part

Figure 4.34 illustrates the entire process of handling interrupts on the kernel level. If the interrupt in question was masked, it is marked as pending (if it was edge-triggered only) in the state MARK_PENDING and the end of interrupt (EOI) acknowledgment is sent immediately (state EOI_FAST). Otherwise, the interrupt is masked (state SW_MASK) and the internal handling started (state HANDLE_HW_IRQ). In this case, an end of interrupt

is signalled later (state EOI) and in any case, the hardware return that switches from the interrupt handling stack back to the stack frame of the interrupted control paths is carried out in state IRET.

The hardware handling (state HANDLE_HW_IRQ) first determines whether a handler is waiting by evaluating the thread control block. If this is not the case, a message is sent (in IRQ_THREAD_IPC) and the handling is aborted. If a handler is waiting, the remaining handling depends on the location (on which CPU the handler is waiting), states TRANSMIT_IRQ and DO_IPC model the two possible cases. Finally, a local handler can be scheduled (depicted by RUN_HANDLER). The messaging towards and the scheduling of the threads is modeled by means of generated events, Table 4.19 shows the detailed transitions.

ID	Source	Target	Event/Action	Description
NST	HANDLE_HW_IRQ	IRQ	$DEVT_{x,INT}$, x CPU number	An interrupt occurred - nesting
IRQ	IRQ_THREAD_IPC	HANDLERS	$\lambda/SIGEVT_i$, i handler thread ID	IPC delivered to corresponding thread
IRQ	DO_IPC	HANDLERS	$\lambda/SIGEVT_i$, i handler thread ID	IPC delivered to corresponding thread
FWD	IRQ	HANDLERS	$\lambda/DEVT_{x,INT,IPI}$, x target CPU	Interrupt forwarded via XCPU mailbox

Table 4.19: Transitions in the Pistachio I/O interrupt handler model

4.4.5.3.2 Timer Interrupts and Scheduler - TIMER_HANDLER

As in any system, the timer interrupt in Pistachio is of special importance. Here, also the scheduler is of interest since it is part of the IHF. As in OpenBSD, before the timer handler (state TIMER_HANDLER) starts processing the interrupt, it acknowledges the reception and sends the EOI to the hardware circuit up front (state EOI_APIC). Figure 4.35 shows the entire handling.

If a timer interrupt was sensed on the primary processor, i.e. CPU 0, the global timer and usage statistics are updated (state UPD_TIMER). In every case, all inter-processor messages are checked – submachine XCPU_MAILBOX – before the scheduling is started. By this, the kernel ensures that the scheduler reacts to the most recent data values and not to any deprecated information.

The scheduler implements in a straightforward way a priority-based policy enriched by the concept of delayed preemption. Delayed preemption prevents handler threads from being dismissed too early. Basically, a second priority value is considered to decide about the total quantum of computation time a process possesses.

Thus, the scheduler first checks whether a thread with a higher process priority was awoken by any signal, schedules this (WAKEUP) and then terminates. Otherwise, a check on

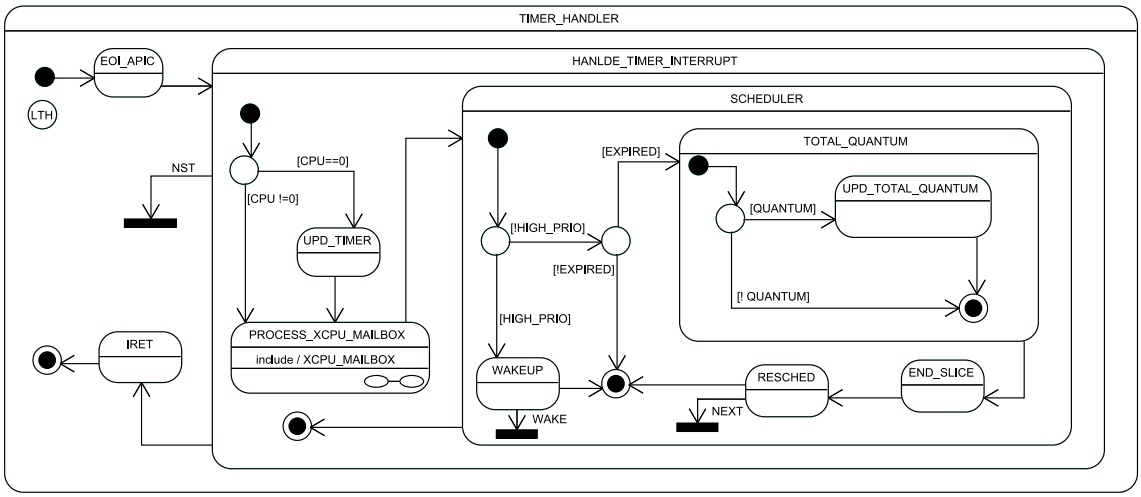


Figure 4.35: Pistachio timer interrupt and scheduler

the time slice is done: if the time slice is not used up, no rescheduling is necessary and the scheduler terminates without any further action.
In state TOTAL_QUANTUM, the calculations concerning delayed preemption, if any, are carried out (UPD_TOTAL_QUANTUM). Finally, a new schedule is calculated in states END_SLICE and RESCHED, and the scheduler terminates. The generated scheduling events and the possible disruption are listed in Table 4.20.

ID	Source	Target	Event/Action	Description
NST	HANDLE_TIMER _INTERRUPT	IRQ	$DEVT_{x,INT}$, x CPU number	An interrupt occurred - nesting
WAKE	IRQ_THREAD_IPC	USER	$\lambda/SCHEVT_i$, i thread ID	Thread with higher priority woken
NEXT	DO_IPC	USER	$\lambda/SCHEVT_i$, i thread ID	Next thread scheduled

Table 4.20: Transitions in the Pistachio timer handler model

Chapter 5

Real-Time in Operating Systems

Since the beginning of computing itself in the early forties, a variety of real-time systems have been used in different fields [LRGW95]. Thus, a large number of real-time definitions from every era of computer science can be found in the literature, e.g. [Mar65], [GHJ+77], [JLT85] and [Liu00].
A commonly agreed standard for real-time operating systems are the POSIX standards P1003.4 and P1003.13, see [Cla90] and [GL91] for a further discussion. Those standards define implementation details, scheduling policies etc. in a bottom-up manner, but do not provide a universal definition nor contribute to the nature of real-time per se.
Although real-time systems are used widely and decades of research have passed, it is surprisingly hard to find a comprehensive and consistent definition of real-time. Even in academic research and teaching, fuzzy definitions are commonly used. Real-time systems are sometimes defined as systems that react to events within a short period of time - how short depends on the field of application.
In particular, the difference between *hard real-time* and *soft real-time* is simply defined as the severeness of not meeting a specified deadline for finishing a certain job: if it is not so dangerous that a deadline is missed (e.g. frame-loss in an MPEG decoder), the system is considered to be soft real-time; if it causes serious damage (e.g. controlling fuel rods in a nuclear power plant), the system is said to be hard real-time. However, this kind of definition is of no use for our purposes: for one thing, it is informal and imprecise. For another, focussing on failure does not allow for the evaluation of the system's ability to meet its constraints. Subsequently, a definition of real-time that satisfies the needs of this thesis is to be given.

5.1 Real-Time

When talking about real-time systems, the focus is on reactive systems rather than on transformational ones. Nevertheless, an adequate real-time definition incorporates transformational – i.e. computational – correctness as well: a punctual but incorrect result of a task is of no use.
The chosen approach separates the two facets of real-time systems: punctuality and correctness.

Definition 5.1 *Real-Time or Time-Constrained System*
A (reactive) system (see Definition 1.2) is called a real-time, or more generally a time-constrained system, if the acceptance of any computational task not only depends on the computational correctness or value of its result, but also on time constraints.

The very term *acceptance* reveals the nature of real-time: Real-time is always judged by or evaluated from an outside perspective. Usually, there are two distinct manifestations of this outside perspective:

1. A human user operates the real-time system directly. The judgement of whether a response meets the requirements is mainly dictated by human sensors and cognitive abilities. Examples: a multimedia player that produces video and audio output will be judged by perceived jitter rather than absolute frame losses; an interactive workstation will be evaluated by its response time behavior.

2. A machine environment (hardware or software) surrounds the real-time facility. The constraints are thus subject to specifications of other systems. Examples: a digital signal processor must meet the time constraints dictated by the sample rate of the system; a digital control circuit steering a machine part must comply with the machine's overall specification; a network device handler must provide data as soon as it is needed by the other parts of the system.

The perceptions of an external entity determine the computational value of a task.

Definition 5.2 ***Computational Task, Computational Value***
Any computational system consists of a finite, nonempty set of computational tasks Θ. A computational task is a job that executes a specified set of actions. The computational (or transformational) value of a task $\theta_i \in \Theta$ is defined by the function

$$value : \Theta \rightarrow [0, 1]$$

It yields the perceived value of the job at the moment of its completion.

The computational value does *not* bear any information about whether the task meets any requirements or fulfills any constraints. It is merely a statement about a task, not about any part of the surrounding system (Section 5.1.1 provides a concrete example).

The chosen interval $[0, 1] \subset \mathbb{R}_0^+$ allows for the definition of a maximal value and provides scalability as well as comparability.

Value Function (VAL)
The definition of computational value only refers to its perceived value at the moment the task is finished. In reality, the value of a task changes over time.

Definition 5.3 ***Value Function***
As the value of a task $\theta_i \in \Theta$ changes over time, the value function

$$value_{\theta_i} : \mathbb{R}^+ \rightarrow]-\infty, 1]$$

yields the concrete perceived value of this task at a given moment in time.
The value function is not necessarily continuous. For t = completion time of θ_i, it holds that $value_{\theta_i}(t) = value(\theta_i)$.

Without loss of generality, the value function is assumed to have a maximum and the completion time of a task is assumed to be $\ll \infty$. The maximum of the function is reached at least by this point in time. Usually, the maximum value is valid for a period of time being a plateau in the function graph.
With respect to scheduling theory, the value function is assumed to be monotonically increasing. However, when using value functions as a vehicle to define real-time, a greater flexibility and accuracy is achieved by allowing negative peaks and even negative values. The latter can be used to represent critical sections (regions) in tasks (cp. Figure 5.1).

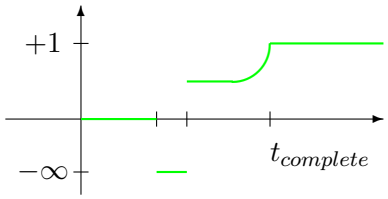


Figure 5.1: Example value function with critical region

The value gained by that task in this time interval is considered negative because any interruption leads to a total failure of the whole system and not only of the task itself.
Note that a negative value at completion time is not by definition allowed: it is absurd if malicious code is not taken into consideration.
In a nutshell: the value function measures the perceived value of a task over time: "what the task *gives*".

Time Utility Functions
It is now vital to define how the benefits of a job develop over time: "what the outside entity *requires*". This is done by means of time utility functions.
Jensen, Locke, Clarke et all invented a generic and flexible mechanism to describe time constraints and the utility [Jen00a] [Jen00b] of final and intermediate results of a task. The perspective taken is the one an outsider has towards the system. The mechanism uses time utility functions (TUFs) [GHJ+77]. In their research, these functions serve as a vehicle to formalize and implement different real-time scheduling algorithms [JLT85] [Loc86] [Cla90]. They show that one of the most important benefits of TUFs is a predictable value for systems with optimized task schedulers.
Nonetheless, TUFs can also contribute in creating a proper definition of real-time. It is defined analogously to the value function as follows:

Definition 5.3.1 ***Time Utility Function***
The actual perceived utility of the results of a task $\theta_i \in \Theta$ for an outside entity changes over time. The utility function

$$utility_{\theta_i} : \mathbb{R}^+ \rightarrow]-\infty, 1]$$

yields the concrete estimated benefit of a task at a given moment in time.

Sufficiency Functions
As mentioned above, the TUF and the value function implement orthogonal views. They can be combined and evaluated in order to determine whether the system or a subset of its tasks meets the expressed requirements (see Figure 5.2). This leads to the definition of sufficiency functions.

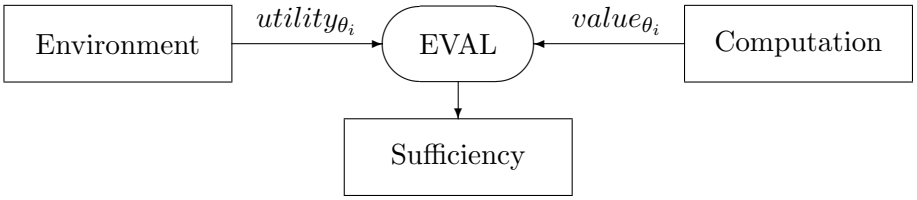


Figure 5.2: Relation between time utility function and value function

There are many possible ways to combine the two functions VAL and TUF. Since in this thesis, it is not intended to exploit the real-time definition for scheduling purposes[1], the simplest and most feasible option for combining them is the preferred choice. This simplest option is to superimpose VAL and TUF.

Definition 5.4 ***Sufficiency Function, Sufficiency Break-Even***
The sufficiency function $suff_{\theta_i} : \mathbb{R}^+ \rightarrow]-\infty, 1]$ *depicts the requirements fulfilment of a task over time.*

$$\forall\, x \in \mathbb{R}^+ : suff_{\theta_i}(x) = \begin{cases} 0 & \Leftrightarrow value_{\theta_i}(x) < utility_{\theta_i}(x) \\ utility_{\theta_i}(x) & otherwise \end{cases}$$

The smallest $x \in \mathbb{R}^+ : suff_{\theta_i}(x) > 0$ *is called sufficiency break-even. If there exists no such* x*, the system is said to be insufficient with regard to* θ_i*.*

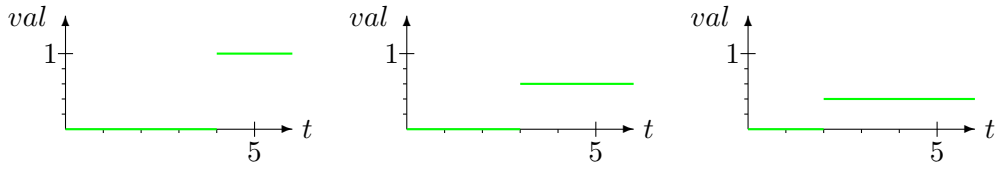
In contrast to scheduling theory, points in time after the sufficiency break-even are also considered.

5.1.1 Working Example for Description of Real-Time Systems

Let us consider again the multimedia car audio navigation system from Section 2.4.1. This time we will focus on the navigation part that has not yet been elaborated. It is assumed to have three computational tasks $(\theta_0, \theta_1, \theta_2)$ available for calculating a specific route around a barrier (e.g. an unexpected construction site). The VAL functions are given in Figure 5.3.

The system is specified by these three tasks. It is considered to be real-time when at least one of the tasks is sufficient with respect to its time constraints.
The different value functions mean that task θ_0 calculates the precise route – its value is maximal and requires 5 time units – whereas tasks θ_1 and θ_2 only provide approximations

[1] In this case, more sophisticated options like defining multiplication classes would be necessary.



(a) $value_{\theta_0}(t)$ (b) $value_{\theta_1}(t)$ (c) $value_{\theta_2}(t)$

Figure 5.3: Value functions for the three tasks in the sample car navigation system

of different quality (0.6 and 0.4) which can be obtained after shorter time intervals (4 and 2 time units).
The working example introduced here will later serve to clarify the difference between hard and soft real-time.
Depending on the purpose of a system, it has to fulfil different requirements with respect to time constraints. Those constraints classify the system as hard or soft real-time. The utility and thus the sufficiency functions can now be used to characterize a real-time system as either or.

5.1.2 Hard Real-Time

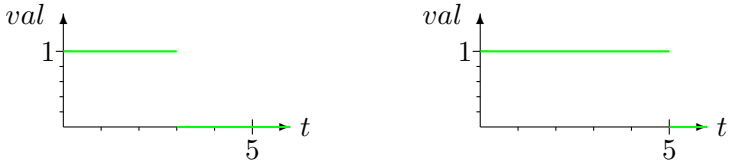
A hard real-time system is characterized mainly by the fact that it has a binary TUF and thus also a binary sufficiency function. If a task produces the absolutely right output (i.e. computational value = 1) in time, it is considered sufficient, if it misses the deadline, the result is considered an error regardless of its computational correctness. Any possible task value $0 \leq val \leq 1$ is deemed insufficient by means of the binary TUF.

Definition 5.5 ***Hard Real-Time System***
A hard real-time system with regard to a set of hard real-time tasks $\Theta_{HRT} \subseteq \Theta$ is characterized by binary time utility functions. It holds that

$$\forall\, \theta \in \Theta_{HRT}, t \in \mathbb{R}^+ : (utility_\theta(t) \in \{0,1\} \Rightarrow suff_\theta(t) \in \{0,1\})$$

Even a hard real-time system might contain some tasks that are not hard real-time tasks at all.

For the given working example, we now assume the two different binary time utility functions $utility_0$ and $utility_1$ as depicted in Figure 5.4.



(a) $utility_0$ leading to an insufficient system

(b) $utility_1$ leading to sufficiency break-even 4 for θ_0

Figure 5.4: Alternative time utility functions for the car navigation system

When superimposed with the value functions $value_{\theta_0}(t)$, $value_{\theta_1}(t)$ and $value_{\theta_2}(t)$, it is clear that in the case of the first TUF, the hard real-time system is insufficient – at no

time is the value of any task bigger than the postulated utility. In the case of the second TUF, the task θ_0 fulfils the requirement with sufficiency break-even $x = 4$. Since in this example one task meeting the constraints is enough, the system is considered to be hard real-time.
Note that, contrary to the common understanding of hard real-time systems, this definition does not consider the magnitude of the specified time-frames. Furthermore, in this thesis, the – very imprecise – severity of an error is not considered either.

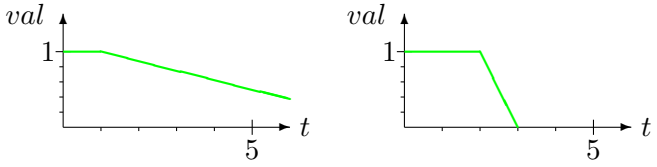
5.1.3 Soft Real-Time

In contrast with hard real-time systems, soft real-time systems are characterized by any kind of time utility functions. According to this definition, any hard real-time system is a special case of a soft real-time system though it will be shown later that this does not hold any more in real-world systems.

Definition 5.6 ***Soft Real-Time System***
A soft real-time system with regard to a set of soft real-time tasks $\Theta_{SRT} \subseteq \Theta$ is characterized by a specific time utility function. It holds that

$$\forall\, \theta \in \Theta_{SRT}, t \in \mathbb{R}^+ : (\mathit{utility}_\theta(t) \in [0,1] \Rightarrow \mathit{suff}_\theta(t) \in [0,1]$$

Referring to the working example, we now assume the two different non-binary time utility functions $\mathit{utility}_2$ and $\mathit{utility}_3$ as depicted in Figure 5.5.



(a) $\mathit{utility}_2$ (b) $\mathit{utility}_3$

Figure 5.5: Alternative non-binary time utility functions for the car navigation system

The sufficiency evaluation for the three tasks θ_0 , θ_1 and θ_2 as well as the two non-binary time utility functions $\mathit{utility}_2$ and $\mathit{utility}_3$ is provided by Table 5.1. The given t depicts the sufficiency break-even. Due to the fact that VAL and TUF are superimposed, no other than those values given by the TUF can be reached.

	θ_0	θ_1	θ_2
$\mathit{utility}_2$	$t = 4$, $\mathit{val} = 1.0$	$t = 4.2$, $\mathit{val} = 0.6$	$t = 5.8$, $\mathit{val} = 0.4$
$\mathit{utility}_3$	$t = \infty$, $\mathit{val} = 0.0$	$t = \infty$, $\mathit{val} = 0.0$	$t = 2.6$, $\mathit{val} = 0.4$

Table 5.1: Soft real-time TUFs: sufficiency break-even

5.2 From Task Perspective to the IHF

So far the focus of defining what real-time is all about was on tasks and sets of tasks[2], but not on complete real-world operating systems. This perspective is fully acceptable for

[2]called a system previously in this chapter

scheduling theory. Since most real-time literature is mainly about scheduling, there exists no suitable projection of a general real-time definition to any other part of a real-world operating system – such as the IHF. Constructing such a projection is the subject of this section.

5.2.1 Architectures

When considering a projection of the real-time properties as defined above to the interrupt handling facility, it is crucial to determine if, and if so how, the operating system's overall architecture influences this projection.

- Different operating systems have different degrees of transparency towards the application layer regarding the handling of hardware interruptions. In other words, effects that are relevant for the IHF might influence the application layer more or less depending on the system architecture.
- Depending on the the way the IHF is embodied in a system, different facets of its real-time capabilities influence the system in very specific ways.

Clearly, the architecture of an operating system matters when taking into account underlying parts of the system such as the IHF. Since in the former considerations, a system's real-time capabilities were judged from an outside perspective of the tasks only, now the overall architecture and the building blocks will be taken into account. Doing so, it becomes possible to isolate parts of a system and to determine their specific real-time capabilities.
Since different operating systems with different architectures are investigated in this thesis, it would be incorrect to use one specific architecture to perform the projection. Doing so would result in the loss of objectivity in judging and comparing the systems' capabilities. To deal with this problem, a target-based approach is taken: for a specific class of real-time application scenarios, a generic architecture is elaborated and from that, a projection is constructed. The research of the chair of operating systems focusses on operating system support for path-based processing of multimedia data. This class of applications allows for the definition of such a generic architecture.
Koenen-Dresp and Schöning [KDSS07] discuss different kinds of real-time systems for use in multimedia-oriented systems based on the component extension (CE) idea [Sch08].

The Component Extension

The underlying concept of the component extension is to interpret any multimedia application as a scenario graph composed of components (vertices) and channels (edges). By using such a structure, it is possible to exploit implicit information about the application itself far beyond the idea of the conventional process model.
Figure 5.6 shows the application scenario graph for an audio-video player with subtitle overlaying. In order to prove the feasibility of the concept and to implement an execution platform for such scenario graphs, a CE system based on the Linux 2.6 kernel is specified [SM07]. Beyond this mere implementation, the specification of a CE system led to another achievement: the possibility of defining a class of real-time systems by means of architectural properties.

This class of Multimedia-Oriented Soft Real-Time Operating Systems (MOSRTOS) is characterized by the CE and its architectural properties discussed later in this section.

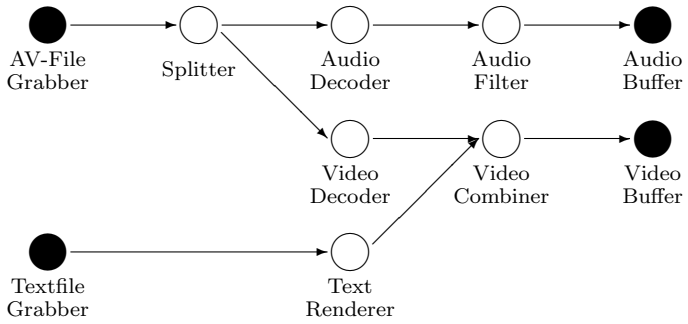


Figure 5.6: Example of a scenario graph of a multimedia application

Like most structured software systems, a component extension software system can be regarded as a multi-tier architecture. Although Müller provides a migration of the CE based on the Linux 2.6 kernel where parts are moved into kernel space [Mül08], the component extension is treated as a pure user-space application in this discussion for the sake of simplicity. Figure 5.7 depicts the overall structure of a CE system and its constituent parts: the application, the component extension, an upper and a lower operating system layer, as well as the hardware. Arrows in the figure represent functional dependencies among the layers.

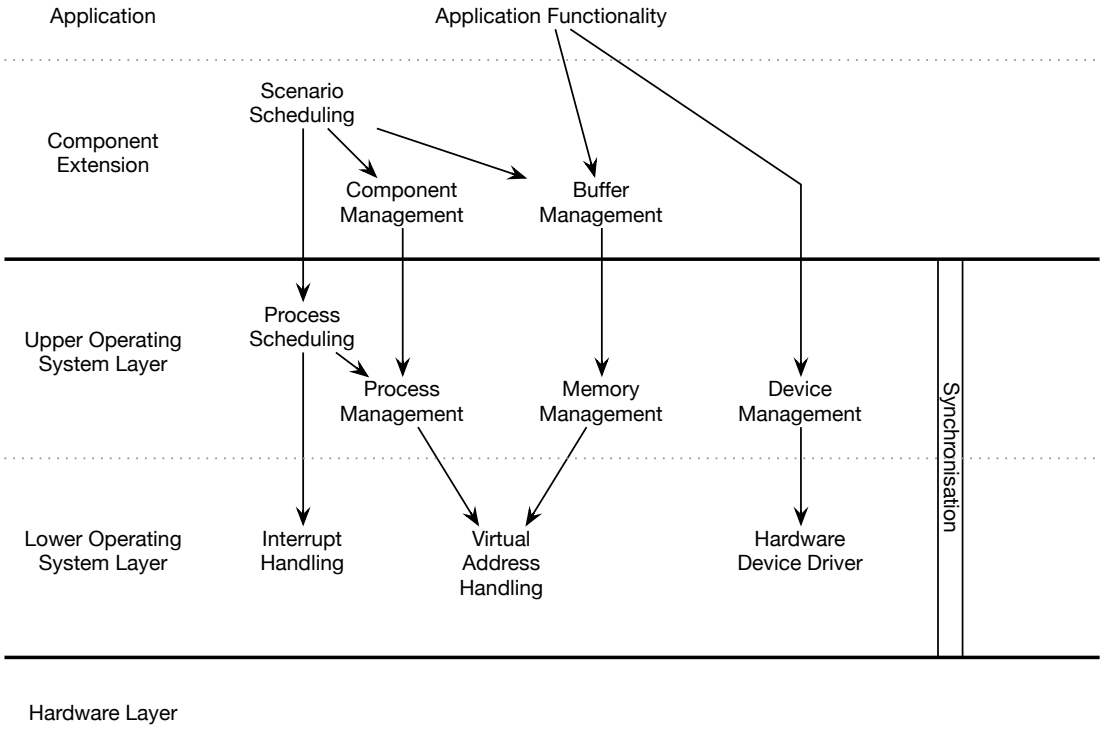


Figure 5.7: Component extension system layers and their interdependencies

The real-time definition from Section 5.1 only applies to the uppermost levels: the application and CE layers. Having defined the functional dependencies, it is now possible to investigate the impact that the lower operating system layer and particularly the IHF may have on this uppermost level.
To explain the architectural implications of the component extension for the projection, let us assume for a moment that we have a normal layered operating system without CE. Keeping the (top-down) functional dependencies between upper and lower OS layer in mind, the corresponding control flow within the OS (bottom-up) is shown in Figure 5.8.

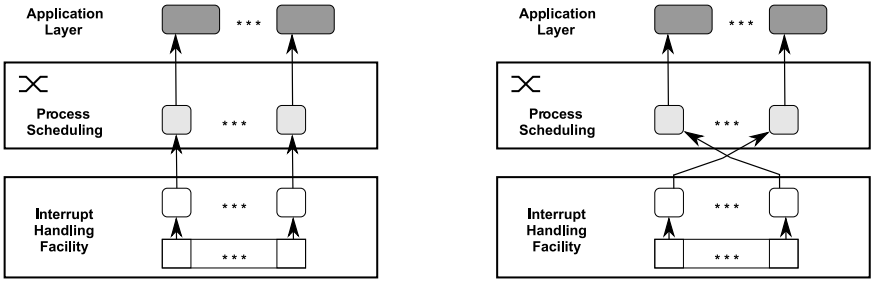
Definition 5.7 ***Transmission Integrity and Priority Compliance***
Transmission integrity (or scheduling consecutiveness in [KDSS07]) is the property of an operating system which guarantees a direct and consistent vertical transmission of asynchronous events from the hardware interrupt level to the scheduler.
Priority compliance means that there exists a unique mapping between interrupt priorities and priorities usable by a scheduler.

Evidently, a system that has transmission integrity is always priority-compliant whereas the converse is not necessarily true.
A set of real-time tasks that is judged by means of a TUF depends on the mentioned transmission of events in two distinct ways:

1. The task execution depends on the overall scheduling of tasks within the operating system. Since a scheduler is driven by clock ticks, it indirectly relies on the accuracy of the IHF.

2. Much more decisive is the fact that often the execution of a task or the evaluation of its punctuality also involves communication with hardware devices. The shape of a TUF often relates to certain points in time such as the disposal of some input data by a specific hardware device such as a hard disk.

For a hard real-time system according to Definition 5.5, transmission integrity and hence priority compliance is the conditio sine qua non. Figure 5.8(a) depicts such a constellation.



(a) scheduling consecutiveness preserves priorities throughout the system.

(b) scheduling is decoupled from the handling of interrupts entirely.

Figure 5.8: Transmission integrity influences the overall system

Unfortunately, the architectural definition of soft real-time is quite complex whereas the formal, theoretical definition is convenient and straight-forward. The situation shown in

Figure 5.8(b) therefore does not show a soft real-time but a *best effort* system as classified e.g. in the taxonomy of Lin et al. [LKPB06], see Figure 5.9[3]. Note that according to the preceding real-time definition, all real-time classes except *best effort* and *hard real-time* are considered soft real-time.

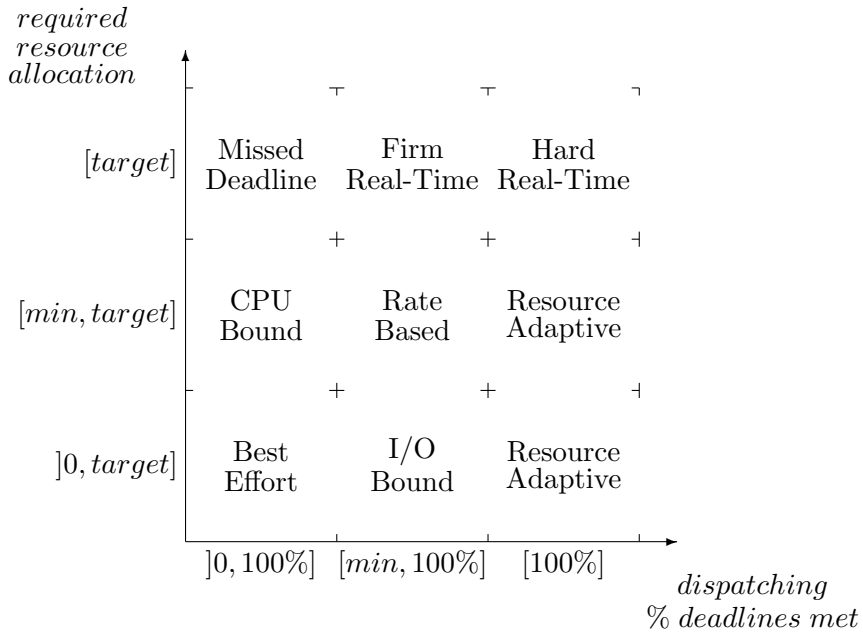


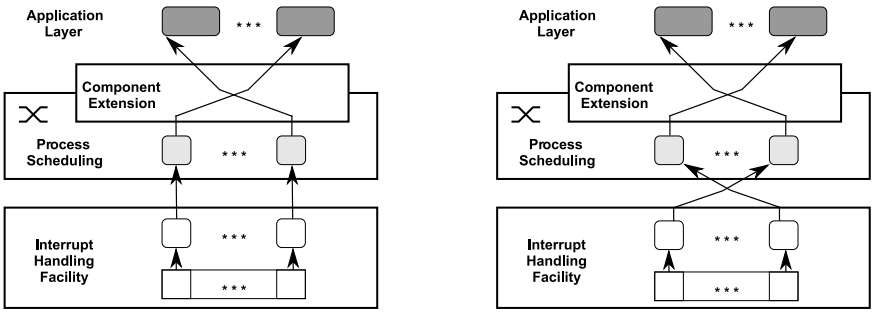
Figure 5.9: Real-time taxonomy according to [LKPB06]

When the IHF and the upper operating system layer allow the control paths to interleave arbitrarily, it cannot be guaranteed that any – however shaped – TUF is ever satisfied. If for such a best effort system statistical statements about interleaving of paths can be given, the system can be reclassified at most as a *missed deadline* real-time system.
As a result, the soft real-time capability of a system relies on there being some restrictions on interleaving control paths and some guarantees about transmission integrity and priority compliance.

We now turn our attention again to a component extension system. Figure 5.10 shows how the component extension is embodied into the previously considered system.
The resulting overall architecture which is the MOSRTOS architecture mentioned needs to implement the following three basic properties:

Complete Decoupling of Architectural Layers
The component extension and the underlying operating system kernel are completely decoupled so that interrupts induce state changes in the scheduler only but never affect the application itself. A control path triggered by a hardware interrupt is completed at the latest within the CE – it never reaches the application layer. The CE itself conducts its own scheduling, resource allocation etc. affecting the application. A real-time capable

[3]In this thesis, the distinction between Resource Adaptive Systems with minimal requirements (upper) and such without minimal needs (lower) is ignored.



(a) scheduling consecutiveness preserves priorities - CE decides consciously

(b) the CE has to compensate for missing priority compliance

Figure 5.10: CE system: the CE mitigates different IHF properties

IHF supports this decoupling.
When the IHF cannot be classified as real-time capable, the consequences of this non real-time IHF can impede the application level due to latencies that can appear. When taking into consideration the time scales discussed in Section 5.2.3.1, latencies caused by interrupts can be considered "white noise" from the application perspective, i.e. they are by orders of magnitude smaller than the ones commonly dealt with on the application layer. Nonetheless, if the IHF does not posses real-time capabilities, this can influence the application layer and thus impede the benefits of decoupling while a real-time capable IHF supports them perfectly.

Omission of Preemption
Multimedia-oriented applications usually do not benefit from preemption since the enforced quasi-parallelism only extends the computation time due to context-switching overhead. Therefore, preemption of multimedia-oriented applications is of no benefit.

Explicit Handover of Control
The structure of any application that is executed on behalf of the component extension is known. Therefore, the handover of control can be accomplished according to logical dependencies known to the CE.

From these preliminary thoughts, the crucial properties of the IHF can now be derived. If the interrupt handling is deferred and completed at the process level (i.e. the completion is subject to scheduling), then it follows that the scheduler has to maintain real-time priorities (with respect to interrupts) in order to satisfy any real-time requirements. Hence, the scheduler as well as the CE can be considered as an enlargement of the interrupt handling facility. Latencies are prolonged by the scheduler control path, yet they remain fixed, i.e. the set of hardware priorities (cp. Section 4.1.1) is partially ordered.
On the contrary, when the handling is immediate and thus prior to scheduling, the system's scheduler and thus the CE lack the ability to reorder priorities to meet non-binary TUFs. Such a system is considered to be primarily event-driven rather than soft real-time capable. Minimal hard real-time systems such as controller circuits in automated produc-

tion facilities lie in this system category.
As discussed, these *architectural properties of the IHF* qualitatively influence a system's real-time abilities considerably. This behavior manifests itself in answering the following architectural IHF questions:

- Does the IHF handle interruptions immediately or is the handling deferred?
- If applicable, what is the moment of the creation of deferred handlers? Are they created at boot time or on demand?
- If applicable, is the deferred handling conducted prior to scheduling or is it subject to scheduling?
- Does the IHF possess the property of priority compliance?

5.2.2 Efficiency, Reliability and Determinism

Beside these qualitative features, the performance properties of the IHF also affect the quantitative behavior that is then sensed by the environment. Note that for this behavior, a notion of time and time-scales is indispensable. Section 5.2.3.1 addresses this issue. Especially with regard to the resource-bounded real-time classes (*I/O Bound* and *CPU Bound*), the effective response behavior of the IHF must be considered. For this, two distinct IHF key figures are vital:

- The disruption path length characterizes the efficiency of the IHF.
- The maximal recursion depth in handling is necessary to properly evaluate the performance.

When it comes to the reliability of an entire system, the notion of correctness in terms of time and computational accuracy (cp. Definition 5.1) differs greatly between that of the IHF and that of the application. The commonly agreed definition $reliability = \frac{errors}{time}$ does not hold for the interrupt handling facility since errors are per se unacceptable in the IHF. The two significant features affecting the system's reliability are:

- The possibility of losing interrupts
- The possibility of handling an interrupt for an infinitely long time

The missing crucial factor for projecting the real-time definitions to the interrupt handling facility is determinism. It is easy to see from Figures 5.8(b) and 5.10(b) that determinism is also an important factor in increasing the number of restrictions on control paths. The two building blocks are:

- Interruptibility: at what points may a control path branch?
- Synchronization: at what points may a control path have to wait?

5.2.3 Timing and Synchrony

5.2.3.1 Time Scales

When considering a model of the interrupt handling facility, the definition of time differs from the real notion of time in a significant way. The system's internal perception of time is created by the interrupt handling facility itself (by means of the timer-interrupt).
It is therefore crucial to explain the different notions of time and their granularity and to elaborate the differences between them. Figure 5.11 provides an overview of the existing notions of time in a layered CE system.

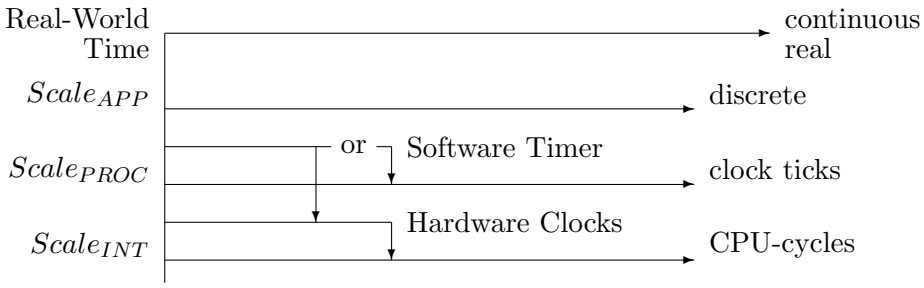


Figure 5.11: Different time scales and timing facilities

The real-world time can be regarded as the overall reference time that is expressed by continuous real values. On the level of the interrupt handling facility, the time scale $Scale_{INT}$ is given in machine *cycles*. Those can simply be converted into real-world time by using the processor frequency. The IHF now induces the time scale $Scale_{PROC}$ that is used for all internal system activities, such as process scheduling. The elapsed internal time is measured in *ticks*. For applications, the time scale $Scale_{APP}$ is defined by discrete real values.

For the interrupt handling facility, the granularity is drawn from the range of nanoseconds. The granularity of the process time scale is characterized by orders of milliseconds. The granularity of applications might be within hundreds of milliseconds or even seconds. Time values related to the time scales $Scale_{PROC}$ and $Scale_{INT}$ can, by definition, be converted directly with respect to latencies. The time scale $Scale_{APP}$ cannot easily be normalized to either $Scale_{PROC}$ or $Scale_{INT}$. Thus, the comparison of (a) latencies induced by the interrupt handling facility and (b) latencies induced by the application level differs in orders of magnitudes. Nonetheless, in reality, the complete decoupling of the architecture (that is present only in a CE system) allows for dealing only with the time scale of the current level of abstraction. Due to this architectural decomposition, it is possible to define dedicated time scales.

From these considerations, one can easily derive that for the overall quality of soft real-time, one additional consideration is significant:

- What is the granularity of the hardware clock and the $Scale_{PROC}$?

5.2.3.2 Synchronous and Asynchronous Events

According to [Int08b], interrupts are asynchronous events, whereas exceptions are called synchronous events. This is reasonable from the micro-architectural point of view. Interrupts (as defined by [Int08a]) can occur at any point in the real-world time (see Figure 5.11). Exceptions are fixed causal reactions of the CPU on the previous[4] instruction. From the modeling and analysis point of view, there are three distinct kinds of synchrony:

Cycle Synchrony. Due to the fact that any micro-processor operates on a discrete timing basis, i.e. its clock pulse, any external event will be sensed and reacted upon in synchrony with the next clock-cycle, more precisely with the next rising edge of the pulse. Cycle synchrony is the weakest form of synchrony considered.

Instruction Synchrony. If an exception or system call occurs synchronous only with respect to previous machine instructions, this property is called instruction synchrony. The Intel architecture ensures that an interruption is handled at the boundary of the currently processed instruction. From an operating systems point of view, the synchrony is not recognized since the control and data flow of the issuing process is unknown. This form of synchrony is stronger than the previous one.

Call Synchrony. A reaction to a system call is considered call synchronous if it occurs synchronously with it. This strongest form of synchrony is still disregarded by the operating system kernel, but it can be sensed in the *ESF* models.

5.3 Quality factors

The previously discussed features, which will be analyzed in Chapter 6, can be mapped to software quality factors. The International Organization for Standardization (ISO) provides several standards dealing inter alia with software quality and quality measurement.

Definition 5.8 ***Quality Factor***
A quality factor according to [ISO05] and [ISO94] is a distinct feature of a system that can be isolated, measured and evaluated by means of a dedicated metric.

The standard set of quality factors for software is defined by [ISO01]. It comprises functionality, reliability, usability, efficiency, maintainability and portability. When considering the above discussion about soft real-time parameters and this set of quality factors, it becomes obvious that there is some overlap. Usability is not an issue at all because the IHF is not operated by users but integrated into a software environment. Maintainability and portability of dedicated IHFs could be assessed by means of the provided models as well, but that is not considered in this thesis. Wieder touches on this topic in [Wie07].

In summary, here are the soft real-time properties that correspond to quality factors in the ISO specification:

[4] With respect to super-scalar architecture, the term "previous" is not entirely correct since the relevant instruction might still be in any pipeline stage of the CPU.

1. Reliability (as in the ISO specification)
2. Architectural Properties (corresponds to functionality)
3. Response Behavior (corresponds to efficiency)
4. Response Determinism (corresponds to functionality)

The last discussed feature – temporal resolution – does not have a counterpart in the ISO specification.

Table 5.2 now assembles the quality factors and the corresponding features of the interrupt handling facility. Those features are now denoted as indicators (see Definition 5.9).

Quality Factor	Indicators
Reliability	Lost Interrupts Infinite Handling
Architectural Properties	Priority Compliance Prior or Subject to Scheduling Immediate or Deferred Handling Creation of Deferred Handlers
Response Behavior	Recursion Depth Disruption Path Length
Response Determinism	Interruptibility Synchronization
Temporal Resolution	Timer Granularity

Table 5.2: Quality factors and their indicators

5.4 Indicators

As stated above, features of the IHF that indicate real-time (or any other) capabilities of a system or subsystem are called *indicators*.

Definition 5.9 ***Indicator***
A quality factor of a system manifests itself in one or more imminent indicators. These indicators may differ for diverse implementations of the same kind of system.

From a first heuristical consideration it can already be seen that the IHF indicators differ significantly among the operating systems that have been modeled. The set of indicators is now compiled from Sections 5.2.1, 5.2.2 and 5.2.3.1 and elaborated on in detail.

5.4.1 Immediate or Deferred Handling

In most operating systems, some part of the interrupt handling is deferred. Deferred handling of interruptions is not a discrete aspect that can be described by a single figure, but it comprises several aspects:

- What part of the handling is conducted in a deferred way?
- In what way (i.e. until what point) is the handling postponed?
- What other entities influence the displacement?

We imagine a "slider" between complete handling of interrupts just when they occur and completely deferred handling with a minimal immediate portion. In an idealized IHF, the slider (or more precisely: the spectrum) could be dynamically adjusted depending on the priority of an interruption and the current load situation of the system.

5.4.2 Creation of Deferred Handlers

If interrupt handling is carried out in a deferred manner, there are two distinct ways of creating the handler processes that conduct the corresponding service:

1. **In advance**, e.g. during the boot process or the initialization phase of the operating system the handlers are created.
2. **On demand**, i.e. at the very point during execution when the process is needed, its data structures are allocated.

Depending on the load profile of a system, either way has advantages. However, an overall paradigm in a CE system is to exploit any knowledge of the applications in question in order to allow for static resource planning. From this, it is easy to see that an in-advance strategy for the interrupt handling facility works best.

5.4.3 Prior or Subject to Scheduling

Postponing the handling of interrupts can cause the real handler to be subject to the overall system scheduling. As a matter of fact, this indicator is equivalent to the question of whether a deferred handler is implemented by means of an activity (subject to scheduling) or not (prior to scheduling). The following definition of activities slightly deviates from the one provided e.g. by Kalfa [GKS94] which was then later adopted by Schöning [Sch08]:

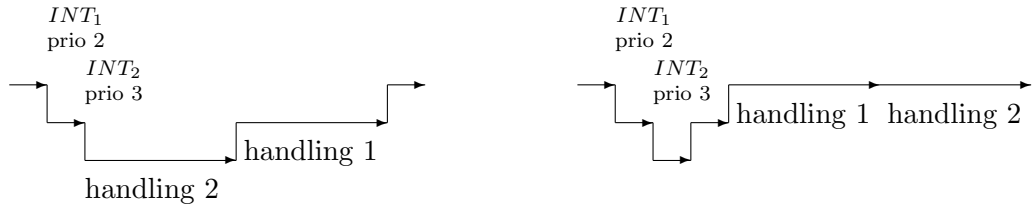
Definition 5.10 ***Activity***
An operating system activity has its own context, i.e. it is implemented as a thread, task or process or a primitive based upon one of these that is perceived by the system scheduler. Any activity context must be created and thus can be destroyed. An activity has the three following properties:

1. *Weight - a figure that depicts the size of an activity's context*
2. *Domain - the memory region (usually kernel or user space) the context resides in*
3. *Type - the kind of activity, i.e. tasklet, thread process, task etc.*

This architectural property is crucial since it allows not only for direct statements about the system architecture but also greatly influences the discussion about priority compliance.

5.4.4 Priority Compliance

For the selected hardware platform, a fixed priority schema (see Section 4.1.1) exists that can be tweaked by means of the `TPR` register. This kind of manipulation of priorities is easy to assess and evaluate because it is statically defined by the system's source code. A far more complicated way of in effect abandoning the priority schema becomes imminent when considering deferred handling of interruptions. Although the immediate part of the handling may follow the priority schema, the real action to be taken can be reordered or completely omitted.
From these considerations arises the question of whether a system is priority compliant[5].



(a) immediate handling - interrupt priorities are preserved

(b) deferred handling, priorities might be abandoned

Figure 5.12: Priority compliance can get lost when deferred handling is regarded

Figure 5.12 shows two interrupts INT_1 and INT_2 with the second being the one with higher priority. In the case of immediate handling (Figure 5.12(a)), the priority schema is preserved, whereas in the case of deferred handling, priority inversion is possible (as in Figure 5.12(b)).

If the deferred handling is conducted by means of activities, i.e. if it is subject to scheduling, the priority compliance of the deferred handling only depends on the scheduler. Anticipating the later analysis, this is the case in Pistachio. With the implemented priority schema for handler routines (in Pistachio they have the highest process priorities within the entire operating system) and the principle of delayed preemption, a consistent software priority schema is established and the hardware priority schema is abandoned. The implementation of use case tailored, novel scheduling policies as discussed by Wieder [Wie07] depends on these software priorities. However, the question of how to prioritize user level handling threads in an adequate way in general is for example in the focus of Löser's research on interrupt priority schemas [LH00].
When, on the other hand, the deferred handling is processed within the kernel path, the question of whether the system is priority-compliant is a design criterion of the OS. The order of invoking deferred handler routines forms an additional interrupt priority schema. Whether this is congruent with the hardware priority schema is a system-inherent design decision.
In the investigated operating systems Linux and OpenBSD, this congruence was not intended. The only priority scheme regarded is the priority schema that evolves from the OS itself: hardware priorities are intentionally ignored.

[5]The term "Priority Compliance" is used as suggested by Jensen et al., see e.g. http://www.real-time.org

From this exhaustive discussion it follows that the indicator of priority compliance will not be investigated further in the indicator analysis. Nonetheless, the formal confirmation about the "schedulability" will be conducted.

Interdependencies
Since the four indicators are closely related, Figure 5.13 shows how they influence each other. For these behavioral indicators, the later model analysis will focus on the dedicated models instead of providing a universal formula that is valid for all models at the same time. For all other indicators, a general method of formal analysis is presented.

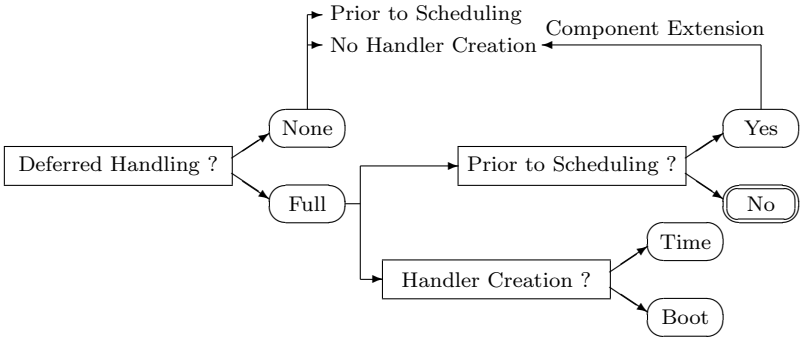
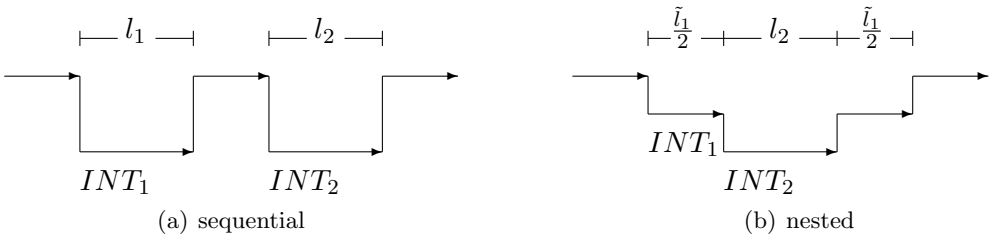


Figure 5.13: Interdependencies between the architectural indicators

5.4.5 Disruption Path Length

The disruption path length is quantified in system steps. A disruption path describes the sequence of actions that are taken between the issuing of a specific disruption and the continuation of the disrupted computational task.
For each dedicated disruption, i.e. each scenario with size 1, the maximal disruption path length is the number of these executed actions under a weak worst case assumption, i.e. the worst combination of runtime variables and conditions.
When investigating more than one disruption, the paths are either independent, i.e. strictly sequential (see Figure 5.14(a), or nested (see Figure 5.14(b)). The latter case directly leads to a combination of this indicator with the indicator recursion depth.



(a) sequential (b) nested

Figure 5.14: Sequential and nested disruptions

In the case of the nested path, it generally holds that $\tilde{l}_1 \geq l_1$ for it causes delay to interrupt the handling of INT_1. A high disruption path length indicates that disruptions trigger a high workload in the SuI.

5.4.6 Synchronization

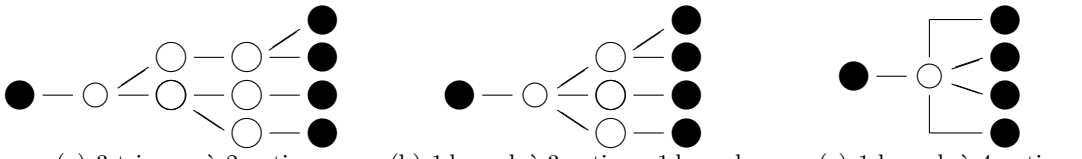
As already discussed in Section 4.3.2, synchronization in an operating system is performed on two different levels:

- The kernel provides synchronization primitives in order to synchronize user processes. This kind of synchronization is not investigated in this thesis.
- The kernel itself needs synchronization in different situations. Avoiding race conditions and protecting critical regions of code are the most common. Kernel data structures that can be accessed simultaneously by interleaving kernel control paths are also protected using primitives, see Section 4.3.

In the following, only the consequences of the usage of synchronization primitives within the kernel are analyzed. Hence (kernel) synchronization is an abstraction from these primitives.
The length of a disruption path with synchronized elements becomes relative: waiting for an external trigger (condition) to occur can take a long "time". Since we use the notion of system steps, there may be a large number of steps where a transition in a path cannot be taken because the synchronization condition (as specified by Definition 4.3) is false. This will severely impede the comparison between different paths, as Section 6.2.1 will show.

5.4.7 Interruptibility

A significant property of any kernel control path is its interruptibility. It is easy to see that the determinism of a path execution is decreased when there are many points where it can be disrupted.
To quantify the determinism of a certain path, it is not only important to know if and how often it can be interrupted but also how this can occur – i.e. in what states and by what external triggers. From a purely stochastic point of view, it is worse if a path can be disrupted in three different states by one trigger each than if one single state can be disrupted by three triggers. Figures 5.15(c), 5.15(b) and 5.15(a) illustrate this difference.



(a) 3 triggers à 2 options (b) 1 branch à 3 options, 1 branch à 2 options (c) 1 branch à 4 options

Figure 5.15: Comparison of branches and options

From these considerations it becomes clear that the later model analysis of an IHF's interruptibility must provide more than a single figure indicating how often a certain path can be disrupted.

5.4.8 Recursion Depth

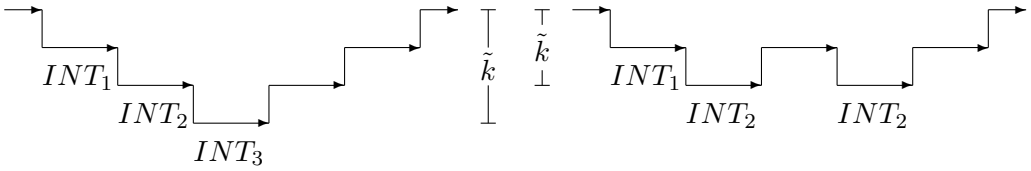
When interrupts are nested, the nesting depth is of great interest as discussed above. It is limited by the k-boundedness property. This upper boundary is derived from the Intel Architecture (cp. Section 4.1.1). The maximal number of nested interruptions ($32 \leq INT \leq 255$) delivered via the IOAPIC and two exceptions: an arbitrary exception and the registered double fault handler on top of it. If system calls are implemented by means of the assembly language instruction `int n`, this one additional recursion layer is already included in the number of disruptions possible. Otherwise, i.e. if the modern `SYSENTER` and `SYSEXIT` instructions are used by the operating system, an additional nesting level is achieved. For the boundary value it holds that $k = 224 + 2 + 1 = 227$. The recursion depth indicator now depicts the following:

1. The maximal possible recursion depth k_{sys} that can be reached by a system running on a specific machine setup is one crucial property of an IHF.

2. The concrete recursion depth k_{scen} that is reached when a specific scenario is considered.

3. The maximal value of system steps kl_{max} that make up the worst case scenario for both: recursion depth and disruption path length.

Figure 5.16(a) illustrates the indicator recursion depth with $\tilde{k} = 3$, whereas $\tilde{k}$ could be either k_{sys}[6] or k_{scen}. The third case, the flexible combination of the two indicators, needs more detailed consideration.



(a) nested interrupts INT_1, INT_2 and INT_3 (b) transient interrupt INT_2

Figure 5.16: Nested disruptions - legal and illegal case

A heuristical approach leads to the formula $kl_{max} = k_{sys} \times l_{max} + overhead$. Unfortunately, this heuristic does not at all take into account the case where interrupts might be transient as depicted in Figure 5.16(b). It is obvious that this situation can easily result in $kl_{max} = \infty$. As a matter of fact, this situation is not ruled out by the Intel micro-architecture. In order to exclude this situation of transient interrupts from the later analysis, a fairness constraint (see Definition 6.8) will be given.
Note that this artificial fairness constraint only supports the analysis of the models. It does not reflect any inherent property of the operating systems under investigation. Strong fairness amongst scheduled processes as a system property is a non-trivial aspect in operating systems research that has not been solved satisfactorily yet (see for example [LS07]).

[6] for a hypothetical minimal system

5.4.9 Infinite Handling

The problem of starvation and deadlocks has always been a common one in operating systems research. For the IHF, the probability of such a situation existing is exacerbated by the fact that external stimuli influence the control flow. The indicator "Infinite Handling" summarizes these two problems. Since the case of transient interrupts certainly gives rise to such a situation, a meaningful evaluation of this indicator also requires additional fairness constraints limiting the set of valid scenarios as discussed in Section 5.4.8. It serves no purposes to identify that there exists an (infinite) number of scenarios that for example create infinite path lenghts and depths. It is more valuable to create a way of proving that there are no additional situations that unexpectedly lead to a starvation or deadlock situation.

5.4.10 Lost Interrupts

Any CPU hardware platform that follows the Intel Architecture allows for buffering interrupts that occur while interrupt lines are masked. It is the responsibility of the system software to handle these buffered interrupts before a buffer overflow occurs. It is important to analyze all possible situations the IHF can be in when such a situation arises. Obviously, one special case where interrupts are lost is when the system handles an interrupt for an infinite period of time (see previous section).
The second alternative where interrupts could be lost is when, for whatever reasons (e.g. implementation errors), the appropriate handling routine is not invoked during the IKCP. Thus is has to be checked as well whether all IKCPs are correctly traversed.

5.4.11 Timer Granularity

As already mentioned in Sections 4.1.1 and 5.2.3.1, the temporal resolution of the system under investigation may vary due to implementation and usage of the underlying hardware facilities. An evaluation of the real temporal resolution provided by the hardware and the timer interrupts – that are necessarily part of any IHF – thus benefits this thesis as well. It was a conscious choice not to pursue this indicator later in the thesis for there already exists much research dealing with it. For example, the author discusses this topic for the old Linux 2.4 Kernel [Koe02].

Chapter 6

Techniques for Indicator Analysis

In this chapter, different approaches to analyzing the models with respect to the different indicators are presented. As already discussed in Section 5.4.11, the indicator dealing with timer granularity that is outside the scope of the models is not regarded further in this chapter. Although the building block of synchronization primitives is not elaborated in the models, the consequences of synchronization, i.e. the influences on paths that contain synchronized elements certainly are. Hence, the synchronization indicator is included in the analysis of the disruption path length.
The remaining indicators are subdivided into three disjoint classes, determined by the way the analysis is conducted. Table 6.1 lists these categories.

Architectural Analysis	Control-Based Analysis	Data-Based Analysis
Prior or Subject to Scheduling	Disruption Path Length	Infinite Handling
Immediate or Deferred Handling	Interruptibility	Lost Interrupts
Creation of Deferred Handlers	Recursion Depth	

Table 6.1: Methods of checking the different indicators – a classification

Indicators belonging to the first category, architectural analysis, can be checked in the simplest way: a mere investigation of some axiomatic properties of the *ESF* models is performed – i.e. the behavior of a system and thus its model is of no interest here. The axioms can be postulated without taking much of the *ESF* or statechart semantics into account. The axiomatic properties are formulated in detail on a per-model basis.

The indicators of the second category are based on general, system-inherent parameters regarding the handling of interrupts. Best case and worst case numerical figures will be derived here. Taking the applied best case and worst case assumptions as fixed points, constructing indicator-tailored representations of the *ESF* models, e.g. using graphs, is possible. This allows for focussing on each indicator omitting model details that are unnecessary for the respective indicator.

The indicators of the third category will give information on the worst case consequences of the actual behavior of a SuI. Note that these indicators cannot be investigated without considering all possible variants of stepping through a system including multiple runs, recursion etc. Temporal logic model checking is used for them.

6.1 Architectural Analysis

The architectural analysis is an axiomatic approach to checking the architectural indicators. It is quite natural that architectural properties are fully independent of system behavior. In software engineering, the architectural peculiarities are defined mainly by structural information such as class diagrams for object oriented software systems. In the *ESF* models, the architectural properties are integrated into the behavioral information. A set of simple axioms defines how to evaluate this.

6.1.1 Immediate or Deferred Handling

In the models presented, this architectural indicator can be analyzed as follows: If there is one handling state in the kernel for every interrupt, the IHF applies immediate handling. When there are separate handler states for all interruptions, the IHF handles interruptions in a deferred manner. If both cases exist, an IHF implements hybrid handling.
A set of functions for assigning disruptive events to their handling model state(s) is given as follows:

Definition 6.1 ***Handling Functions*** $\mathfrak{hdl}$, $\mathfrak{hdl}^*$ ***and*** $\mathfrak{hdl}^b$
All IHF models allow for an unambiguous definition of a function $\mathfrak{hdl} : \overrightarrow{E} \rightarrow S$ *that associates any disruptive path event with a model state located within the kernel that represents the handling of this path event, i.e. a state which is dedicated to the handling of this path event only.*
The function $\mathfrak{hdl}^* : \overrightarrow{E} \rightarrow 2^S$ *yields multiple handler states for the particular interrupts. Those states do not necessarily need to be located in kernel space.*
The function $\mathfrak{hdl}^b : \overrightarrow{E} \rightarrow S'$ *with* $S' = \{s | s \in S;\ s\ basic\}$ *yields the one basic state that unambiguously represents the kernel handling of the interrupt. It is defined as follows:*

$$\mathfrak{hdl}^b(e) = \begin{cases} \mathfrak{hdl}(e) & \mid \textit{if } \mathfrak{hdl}(e)\ \textit{basic} \\ \delta^n(\mathfrak{hdl}(e)) \textit{with } n \geq 1 \textit{ so that } \delta^n(\mathfrak{hdl}(e))\ \textit{basic} & \mid \textit{otherwise} \end{cases}$$

The definition of $\mathfrak{hdl}^b$ that will be used later indicates that the default function is applied until the default basic state is found.

With these definitions, the axioms for defining immediate, deferred and hybrid handling are as follows:

1. Immediate handling iff $\forall\, e \in \overrightarrow{E}, e$ is DEVT : $[\{\mathfrak{hdl}(e)\} = \mathfrak{hdl}^*(e)]$
2. Deferred handling iff $\forall\, e \in \overrightarrow{E}, e$ is DEVT : $[\{\mathfrak{hdl}(e)\} \neq \mathfrak{hdl}^*(e)]$
3. Hybrid handling iff neither of the previous cases applies

Hybrid systems can be further investigated in order to determine which interrupts are handled immediately and which are deferred. In this case, either a per-interruption or a per-group investigation can be conducted.

6.1.2 Creation of Deferred Handlers

As for the following architectural indicators, this indicator is only applicable when deferred interrupt handling is operative. An investigation of the creation of deferred handlers can only be conducted if it is an activity. Deferred handling that is implemented as a simple kernel function is not taken into account since no context needs to be allocated.

The position of the context creation state, i.e. whether it is accessed once during initialization or every time the handler is called will be investigated. As the models depict only the interrupt handling as a white box, the initialization itself is blanked out. Thus, it has to be checked whether there even is a context creation state within the model. If so, the handler context is created when a handler is actually called. Otherwise, it follows that the context creation takes place during system initialization.

6.1.3 Prior or Subject to Scheduling

Deferred handling can either be processed prior to or be subject to scheduling. For the latter case, two possibilities exist: a part of the handling is done in user-process space or as a kernel-process thread. Priority to scheduling is only possible if no part whatsoever of the handling is displaced to a separate context. Thus, the following axioms are given:

1. Subject to scheduling if $\exists\, s \in \mathfrak{hdl}^*(e) : [s \in U]$ with DEVT $e \in \overrightarrow{E}$ being the path event belonging to the handling state, $U \subset S$ being the set of states within user-process space
2. Subject to scheduling if $\exists\, s \in \mathfrak{hdl}^*(e) : [s \in KT]$ with DEVT $e \in \overrightarrow{E}$ being the path event belonging to the handling state, $KT \subset S$ being the set of states representing kernel-space threads
3. Prior to scheduling iff neither of the previous cases applies

6.2 Control-Based Analysis

In order to derive general statements about the real-time capabilities of the SuIs, system-inherent numerical properties must be investigated. Best case and worst case figures are therefore of great interest: they determine the scope of the actual values expected in a system. Concentrating on statements depending on specific scenarios would considerably reduce the general significance of the statements.

The indicators of this category, disruption path length and interruptibility, can thus be defined based on best and worst case assumptions abstracted from actual system behavior. Doing so, the indicators can be analyzed based on the static models themselves without needing to take any run-dependent properties into account.

6.2.1 Disruption Path Length

The goal of the analysis of this indicator is to quantify the minimum and maximum of system steps necessary to process the path induced by an interruption. To achieve this, a

graph based on basic states and full transitions is constructed. This path analysis graph (PAG) also includes information on synchronization points which then allows for further analysis of the synchronization indicator in this context.

The use case tailored graph representation of a concurrent component of an *ESF* statechart has the following properties:

1. its vertices represent the basic states within this component
2. its edges represent full sequential transitions and have two possible colors:
   (a) black if the corresponding statechart transition has a label without condition (stepping through)
   (b) red if the corresponding statechart transition has a label with synchronization condition (synchronized step), i.e. the transition does not pass any condition connector

   This edge coloring is a more intuitive binary weighting. As the PAG will be later extended to serve the analysis of interruptibility (cp. Section 6.2.2), the edge coloring will be enhanced as well.
3. condition connectors create alternative paths in the graph being translated into multiple black edges
4. split states are translated into different edges, one per outgoing transition
5. combine states are translated into different edges, one per incoming transition

In statecharts, states have to be unique within their direct domain (i.e. the superstate) only. When flattening this hierarchy to create a graph based only on basic states, ambiguous identifiers could appear. Therefore, the hierarchy of the basic states has to be taken into account to allow for unique node names. The identifier of a node is now an n-tuple with n being its depth in the state hierarchy.

Definition 6.2 ***Traverse Hierarchy Name Function*** $\mathfrak{thn}$
The function $\mathfrak{thn} : S \rightarrow S^n$, $n \in \mathbb{N}$ *yields the full hierarchy structure of a state* s*, i.e. all states above and including* s *in the hierarchy to the root state* r*.*

$$s = r \Rightarrow \mathfrak{thn}(s) = s$$
$$s \in S \Rightarrow \mathfrak{thn}(s) = \mathfrak{thn}(s'), s \qquad \text{with } s \in \rho(s'), s' \in S$$

The path analysis graph is now defined as follows:

Definition 6.3 ***Path Analysis Graph*** PAG
The path analysis graph $\mathfrak{G}_{PAG}$ *of an ESF statechart XOR state* s_{PAG} *(concurrent component) is a weighted directed graph, i.e. a 3-tuple* $(V_{PAG}, E_{PAG}, \mathfrak{col}_{PAG})$ *with* V_{PAG} *being a finite set of vertices (also called nodes),* E_{PAG} *a finite set of edges,* $E_{PAG} \subseteq V_{PAG} \times V_{PAG}$ *and* $\mathfrak{col}_{PAG} : E_{PAG} \rightarrow \{black, red\}$ *a coloring function that assigns a color, here a binary weight to each edge.*

Note that the finiteness of V_{PAG} and E_{PAG} is a direct consequence of the finiteness of statecharts.

Definition 6.3.1 ***Vertices*** V_{PAG}
The set of vertices V_{PAG} of a PAG is defined on the set $\hat{S}$ with $\hat{S} \subseteq S, \forall\, s_1, s_2 \in \hat{S}$: $s_1 \not\perp s_2, \hat{S} = \rho(s_{PAG})$:

$$\begin{array}{lll} \forall\, s \in \hat{S}, s \text{ basic} & : & \exists v \in V_{PAG} \quad : v \stackrel{\text{id}}{=} \mathfrak{thn}(s) \\ con_d(s_{PAG}) & : & \exists v \in V_{PAG} \quad : v \stackrel{\text{id}}{=} START \\ con_t(s_{PAG}) & : & \exists v \in V_{PAG} \quad : v \stackrel{\text{id}}{=} FINAL \end{array}$$

The path analysis graph contains all basic substates as vertices that are identified by the complete state hierarchy of each state.

Definition 6.3.2 ***Color Function*** $\mathfrak{col}_{PAG}$***, Edges*** E_{PAG}
The color function $\mathfrak{col}_{PAG} : E_{PAG} \to \{black, red\}$ assigns a color to each edge. The set of edges E_{PAG} and the corresponding edge coloring are constructed as follows:

1. *Sequential and conditional transitions are translated one-to-one:*

$$\begin{array}{ll} \forall\, t_{seq} \in \tilde{T} & : \exists\, e \in E_{PAG} : e = (src(t_{seq}), tgt(t_{seq})), \mathfrak{col}_{PAG}(e) = black \\ \forall\, t = (s_1, \lambda[c]/\mu, s_2) \in \tilde{T} & : \exists\, e \in E_{PAG} : e = (s_1, s_2), \mathfrak{col}_{PAG}(e) = red \end{array}$$

   where $s_1, s_2 \in \tilde{S}, c \in C$.

2. *Transitions to and from split and combine states are pairwise combined into transitions from state to state:*

$$\forall\, t_1 = (s_1, l, s_{SP}), t_2 = (s_{SP}, \lambda/\mu, s_2) \in \tilde{T} \quad : \exists\, e \in E_{PAG} : e = (s_1, s_2), \quad \mathfrak{col}_{PAG}(e) = \begin{cases} black & \text{if } l = e/a \\ red & \text{if } l = e[c]/a \end{cases}$$

$$\forall\, t_1 = (s_1, l, s_{CO}), t_2 = (s_{CO}, \lambda/\mu, s_2) \in \tilde{T} \quad : \exists\, e \in E_{PAG} : e = (s_1, s_2), \quad \mathfrak{col}_{PAG}(e) = \begin{cases} black & \text{if } l = e/a \\ red & \text{if } l = e[c]/a \end{cases}$$

   where $s_1, s_2 \in S, s_{SP} \in SP, s_{CO} \in CO$.

3. *Full compound transitions involving condition connectors are mapped as black edges:*

$$\forall\, ts_1 = (s_1, l, s_2), ts_2 = (s_2, \lambda[c]/\mu, s_3) \in \widetilde{TS} \quad : \exists\, e \in E_{PAG} : e = (s_1, s_3), \quad \mathfrak{col}_{PAG}(e) = black$$

   where $s_1, s_3 \in S$, $s_2 \in COND$.

4. *Full compound transitions involving default or termination connectors are mapped one-to-one:*

$$\forall\, ts_1 = (s_1, l, s_2), ts_2 = (con_d(s_2), \lambda/\mu, s_3) \in \widetilde{TS} \quad : \exists\, e \in E_{PAG} : e = (s_1, s_3),$$
$$\mathfrak{col}_{PAG}(e) = \begin{cases} black & \text{if } l = e/a \\ red & \text{if } l = e[c]/a \end{cases}$$

$$\forall\, ts_1 = (s_1, l, con_t(s_2)), ts_2 = (s_2, \lambda/\mu, s_3) \in \widetilde{TS} \quad : \exists\, e \in E_{PAG} : e = (s_1, s_3),$$
$$\mathfrak{col}_{PAG}(e) = \begin{cases} black & \text{if } l = e/a \\ red & \text{if } l = e[c]/a \end{cases}$$

where $s_1, s_2, s_3 \in S$.

5. *Transition segments from the default connector and to the termination connector of state* s_{PAG} *are translated as follows:*

$$\forall\, ts = (con_d(s_{PAG}), \lambda/\mu, s_1)) \in \widetilde{TS} \quad : \exists\, e \in E_{PAG} : e = (START, s_1), \mathfrak{col}_{PAG}(e) = black$$
$$\forall\, ts = (s_1, l, con_t(s_{PAG})) \in \widetilde{TS} \quad : \exists\, e \in E_{PAG} : e = (s_1, FINAL),$$
$$\mathfrak{col}_{PAG}(e) = \begin{cases} black & \text{if } l = e/a \\ red & \text{if } l = e[c]/a \end{cases}$$

where $s_1, s_{PAG} \in S$.

Note that transitions labeled with path events do not have any counterpart in PAG edges.

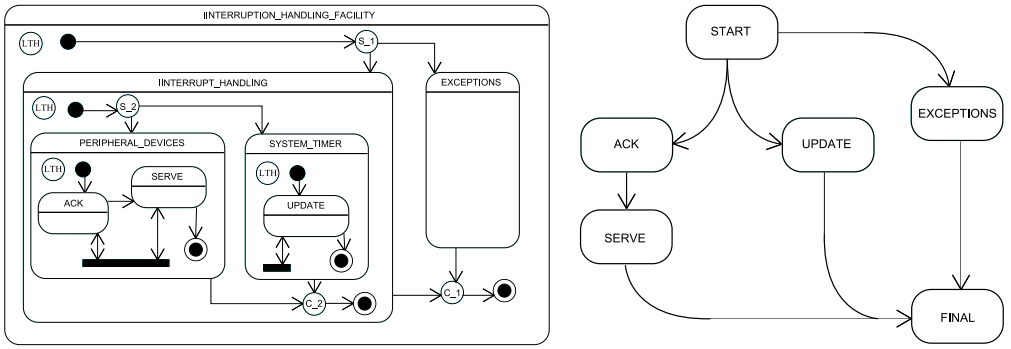
Theorem 1 *The complexity of the PAG construction from an ESF model is* $\mathcal{O}(n^2)$ *with* n *being the number of states.*

The complexity is composed as follows:

- Every basic state is translated into a PAG vertex according to Definition 6.3.1: complexity $\mathcal{O}(n)$
- Every transition originating from every basic state is translated into one PAG edge according to Definition 6.3.2. In a fully intermeshed graph, the number of edges and thus the complexity of translation based on n is $\mathcal{O}(n^2)$.

The overall complexity is then $\mathcal{O}(n+n^2) = \mathcal{O}(n^2)$. □

Figure 6.1 shows an example of an *ESF* state and its PAG. The PAG nodes are due to layout considerations given with their, here unique, short names instead of their full hierarchical identifiers, e.g. INTERRUPTION_HANDLING_FACILITY, INTERRUPT_HANDLING, PERIPHERAL_DEVICES, ACK is simply given as ACK.



(a) simple engineering statechart

(b) path analysis graph for the state INTERRUPTION_HANDLING_FACILITY

Figure 6.1: Example of the creation of path analysis graphs

Definition 6.4 ***Path***
A disruption path is a sequence of edges $e_0, \dots e_{n-1} \in E_{PAG}, n \in \mathbb{N}$ *that are linearly connected:*

$$\forall\, \nu \in \{0, \dots, n-1\} : e_\nu = (u, v), e_{\nu+1} = (v, w)$$

where $u, v, w \in V_{PAG}$*. The number of edges* n *is called the numerical length of the path.*

For each node $v_{ana} = \mathfrak{bbl}^b(e)$, with $e \in \overrightarrow{E}$ being the path events of the interruption subject to analysis, there is a set of disruption paths $P_{v_{ana}}$ from *START* to *FINAL* that passes through this very node, i.e. v_{ana} is the origin of at least one edge. The node being analyzed is the PAG representation of basic state $\mathfrak{bbl}^b(e)$ for the path event $e \in \overrightarrow{E}$.
A cycle within a path is a (sub-)path with length ≥ 1 where the origin of the first edge is the endpoint of the last:

$$e_{cyc_0} = (v, v'), \dots, e_{cyc_l} = (v'', v)$$

where $v, v', v'' \in V_{PAG}, l \in \mathbb{N}$. A cycle of size 1 is called a self loop. $P_{v_{ana}}$ contains all paths passing through a specified node, even those with cycles. Since each cycle can be taken an unlimited number of times, $|P_{v_{ana}}|$ can be ∞. Therefore, now the set $P'_{v_{ana}} \subset P_{v_{ana}}$ is defined which only contains those paths with exactly one occurrence of each possible cycle.
The process of deriving $P'_{v_{ana}}$ is an application of depth-first search (DFS) in graphs. Subsequently, the complexity of this process corresponds to that of established DFS algorithms [AHU74]. Various optimized or distributed variants exist in the literature [SS86] going down to $\mathcal{O}(|E| + |V| \log |V|)$ [Bar98]. For all disruption paths $\in P'_{v_{ana}}$, the length of a disruption path is specified by four degrees of freedom. Those parameters allow for the unambiguous comparison of path lengths among different SuIs as well as for different states v_{ana}. For both minimum and maximum path length, the following parameters must be derived:

1. The numerical path length excluding cycles using a best case or a worst case assumption:

- for the minimum path length: if there exist several sub-paths of different length, the shortest is assumed to be the one taken
- For the maximum path length: if there exist several sub-paths of different length, the longest is assumed to be the one taken

2. The number of cycles that the path contains when taking the longest or shortest sub-paths.

3. The accumulated numeric length of all cycles allows the evaluation of an average cycle length in the path. This is then used to consider overall repetitions.

4. The number of red edges that represent the synchronization points in a path. This parameter must be used to compare path lengths between idle and heavily loaded systems.

A simple example of a disruption path is depicted in Figure 6.2. The path contains exactly one cycle, namely the one $((c_0, c_2), (c_2, c_3), (c_3, c_0))$ of length 3 applying for both the minimal and the maximal paths. Furthermore, there is only one red edge $(b, FINAL)$. The maximal numerical path length without cycles is $l_{max} = 9$, the minimal is $l_{min} = 8$. The overall metric values for the example path are $(9, 1, 3, 1)^{max}, (8, 1, 3, 1)^{min}$. The numerical path lengths in this example are composed as follows:

$$\begin{aligned} l_{min} &= (START, c_0) + (c_1, v_{ana}) + (v_{ana}, a) + min(l_0, l_1) + (b, FINAL) \\ l_{max} &= (START, c_0) + (c_1, v_{ana}) + (v_{ana}, a) + max(l_0, l_1) + (b, FINAL) \end{aligned}$$

Although there might be numerous circles, their numbers of repetitions that occur in the models is bounded. This is implied by the l-boundedness property as defined in Section 3.4.

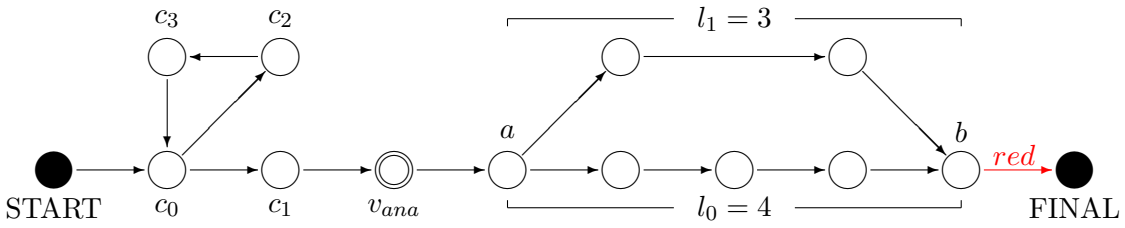


Figure 6.2: Example of a disruption path

The reference system uses model steps instead of real execution times. The path steps match system steps due to the construction rules of the PAG. This principle can be seen as a functional abstraction and is regarded as sufficient for this thesis. In fact, this principle has been used widely in Petri Nets research and applications (see e.g. [Zur97]).
Nonetheless, it is possible to enhance this approach with timing. A valid worst case estimate of the run times is a necessary prerequisite for this (even when the minimal disruption path length is regarded). There are two basic variants:

1. The sequential transition (cp. Definition 3.1) can be used to incorporate real timing directly into the *ESF* models. For that, the condition for stepping further must be redefined. Currently when a state is reached, the enable event automatically is raised

and triggers the next step. By adding a timeout condition to this definition, runtime values for specific parts of the OS code can be embodied. It is then necessary to give an enhanced rule set for translating such timed *ESF* models into a PAG so that the timing information does not get lost on the way.

2. The timing can be applied directly to the PAG after it is generated. Since the timing information is not relevant for the statechart execution itself, unnecessary complexity can thus be avoided. The execution times are then assigned directly to the PAG edges.

In both cases, the path analysis graph would be a weighted graph where the weights represent the execution times. The disruption path length would then be calculated differently (all numeric path lengths would be sums of weights).

6.2.2 Interruptibility

The analysis of the interruptibility indicator further illustrates the benefits of a use case tailored graph representation. Based on the PAG, an interruptibility analysis graph (IAG) is created. Such an IAG has the following properties:

1. The PAG to start with only contains sequential, trivial and conditional transitions.
2. Transitions labeled with path events, i.e. transitions that model interruptions of the control flow are added to the graph. These transitions get a new edge color, green.
3. Each green edge is weighted with one attribute representing one path event. This may result in multiple edges originating from one node.
4. The green edges all lead towards a single dummy node as the target state of the interruption is of no interest for the analysis.
5. The mapping of path events to weights is unique.

In the resulting IAG, those nodes along a path (as defined for the PAG) that have outgoing green edges are of interest. Two questions are then to be answered:

1. How often and where is a path $\in P'_{v_{ana}}$ interruptible, i.e. which nodes with outgoing green edges exist and which path events are associated with this set? The path $\in P'_{v_{ana}}$ is defined as in Section 6.2.1 containing cycles only once.
2. For a given path event, how often is a corresponding green edge traversed within the IAGs for all $P'_{v_{ana}}$?

The formal definition of the IAG is similar to the one for the PAG.

Definition 6.5 ***Interruption Analysis Graph*** *IAG*
The interruption analysis graph $\mathfrak{G}_{IAG}$ of an ESF statechart XOR state s_{IAG} (concurrent component) is a weighted directed graph, i.e. a 3-tuple (V_{IAG}, E_{IAG}, $\mathfrak{col}_{IAG}$) with V_{IAG} being a finite set of vertices (also called nodes), E_{IAG} a finite set of edges, $E_{IAG} \subseteq V_{IAG} \times V_{IAG}$ and $\mathfrak{col}_{IAG} : E_{IAG} \rightarrow \mathbb{N}_0$ a coloring function that assigns a color, i.e. a numerical weight to each edge.

The set of vertices V_{IAG} of an IAG is defined as for the PAG, i.e. the interruption analysis graph also contains all basic substates as vertices that are identified by the complete state hierarchy of each state. Additionally, there exists one artificial node called *dummy* that is the target of all green-colored edges.

The coloring function $\mathfrak{col}_{IAG}$ and the set of edges E_{IAG} differ significantly from those of the PAG. In order to define the coloring function, a mapping between path events and numerical values is defined.

Definition 6.5.1 ***Weight Function*** $\mathfrak{wgt}$
The weight function $\mathfrak{wgt} : \overrightarrow{E} \rightarrow \{2, 3, \ldots, p+1\}$ *assigns a numerical value to a single path event. The values 0 and 1 are excluded since they are reserved for black and red edges. The boundary p is the number of existing different path events.*

Note that since the set of path events is structured, the default implementation of $\mathfrak{wgt}$ is to traverse the tree given in Figure 4.8 on page 59. In this thesis, this default is regarded as sufficient. When it comes to a software implementation, it can be worthwhile to redefine this function to allow for optimization.

Definition 6.5.2 ***Color Function*** $\mathfrak{col}_{IAG}$***, Edges*** E_{PAG}
The color function $\mathfrak{col}_{IAG} : E_{IAG} \rightarrow \mathbb{N}_0$ *assigns a numerical color value to each edge. The set of edges* E_{IAG} *and the corresponding edge coloring are constructed as follows:*

1. *The set* $E_{IAG_{PAG}}$ *is the set of PAG edges as defined in Definition 6.3.2. For the coloring it applies that:*

$$\forall\, e \in E_{IAG_{PAG}} : \mathfrak{col}_{IAG}(e) = \begin{cases} 0 & \text{if } | \ \mathfrak{col}_{PAG}(e) = black \\ 1 & \text{if } | \ \mathfrak{col}_{PAG}(e) = red \end{cases}$$

   In other words, black and red edges are henceforth treated as edges with weights 0 and 1.

2. *For all defined nodes, a green edge is created if the corresponding basic state in the statechart has an outgoing transition that is labeled with a path event:*

$$\forall\, t = (s_1, e/a, s_2), e \in \overrightarrow{E}, s_1 \text{ basic} :$$
$$\exists\, e \in E_{IAG} = (\mathfrak{thn}(s_1), dummy), \mathfrak{col}_{IAG}(e) = \mathfrak{wgt}(e)$$

3. *For each node whose corresponding statechart superstate at any level of the hierarchy has an outgoing transition that is labeled with a path event, an additional green edge is defined:*

$$\forall\, t = (s_1, e/a, s_2), e \in \overrightarrow{E}, s \text{ basic}, s \in \rho^*(s_1) :$$
$$\exists\, e \in E_{IAG} = (\mathfrak{thn}(s), dummy), \mathfrak{col}_{IAG}(e) = \mathfrak{wgt}(e)$$

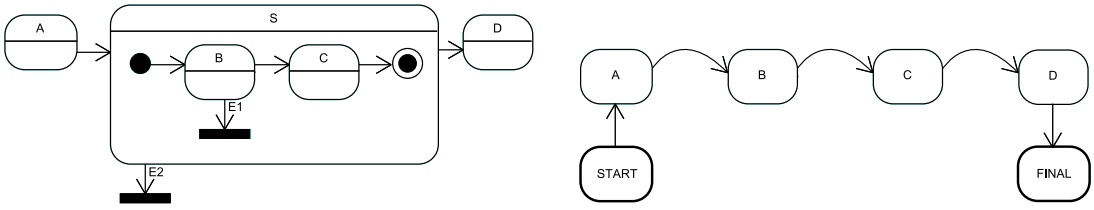
Theorem 2 *The complexity for the construction of the IAG is* $\mathcal{O}(n^2)$.

The complexity is composed as follows:

- The complexity of constructing the PAG which is the basis of the IAG is $\mathcal{O}(n^2)$ according to Theorem 1.
- For each basic state, the entire hierarchy is traversed to create the green edges. At a maximum, the hierarchy depth is n. Thus, the complexity of this step is $\mathcal{O}(n^2)$ as for each hierarchy level, the factor p (the number of possible path events) limits the number of actions.

The overall complexity is therefore $\mathcal{O}(n^2) + p \cdot \mathcal{O}(n^2) = \mathcal{O}(n^2)$. □

Having defined the set of edges E_{IAG} and the coloring applied to all green edges going to the dummy state, the following example shows the creation of a simple IAG. Figure 6.3(a) shows an *ESF* model that contains the transitions E1 and E2 each assumed to be labeled with a path event with the same name. Figure 6.3(b) then shows the corresponding PAG. Note that the set of vertices $V_{PAG} = \{START, A, B, C, D, FINAL\}$ is the basis for the later IAG, depicted in Figure 6.4. This figure also depicts the additional state *dummy* as well as the three green edges, that are (only for the sake of the brevity of the explanation) labeled with the original path events instead of their numerical values.



(a) simple statechart model (b) path analysis graph for the model

Figure 6.3: Sample input for the creation of IAGs

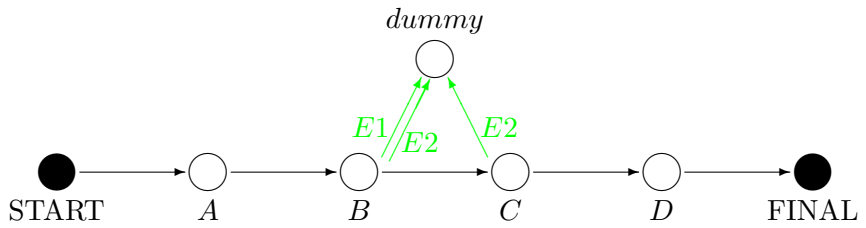


Figure 6.4: Example of an interruptibility analysis graph

Note that the interruptibility indicator depicts at what points along a kernel path the system *can* be interrupted and not *when* a concrete sequence of interrupt handling *is* actually interrupted.
The indicator of interruptibility is then defined in two figures answering the two basic questions stated above:

1. Interruptibility of a specific path by all path events under investigation:

$$Int(path) = \{n_1 \grave{} e_1, \ldots, n_p \grave{} e_p\}$$

with p being the number of different path events, $n_1, \ldots, n_p \in \mathbb{N}_0$, $e_1, \ldots, e_p \in \overrightarrow{E}$ and *path* being any path in the IAG. This figure is defined as a multiset [Bli89], i.e. a special set that can contain one element several times, e.g. $\{2 \grave{} a\}$ is a multiset with two occurrences of element a.

2. Influence of a path event on all paths under investigation:

$$Inf(e) = \{n_1 \grave{} path_1, \ldots, n_x \grave{} path_x\}$$

with x being the number of different paths under investigation, $n_1, \ldots, n_x \in \mathbb{N}_0$, $e \in \overrightarrow{E}$ and $path_1, \ldots, path_x$ being paths in the IAG. This figure is defined as a multiset as well.

6.2.3 Recursion Depth

As discussed in Section 5.4.8, the indicator recursion depth incorporates different forms, i.e. the overall architectural maximum as already dealt with, the system-dependent boundary, a concrete scenario-dependent value and finally the combination of path length and recursion depth. While for the disruption path length and the interruptibility, a graph-based approach relying on worst and best case assumptions was used, for the system- and scenario-dependent recursion depths, a direct and algebraic approach is chosen. A possible analysis for the combination of path length and recursion depth will be discussed later in Section 6.4.

System-Dependent Recursion Depth

The number of different disruptive events (cp. Section 4.2.3.1) that possibly increment the recursion depth – or more precisely the interrupt nesting level – is to be counted. This machine-based maximal recursion depth is composed as follows:

$$k_{sys} = 2 \text{ exceptions} + x_s \text{ syscalls } + x_t \text{ timer } + x_i \text{ IPI } + x_d \text{ IO devices} \tag{6.1}$$

The exceptions and system calls are counted as in Section 5.4.8. The number of IRQ lines allocated by timers is usually $x_t = 1$. If a different line is used for the local and the global timer interrupt, this variable is increased up to 5 (one global one and one local per CPU). Usually, there is only one interrupt vector used by inter-processor interrupts x_i. Nevertheless, in any OS implementation, many of them may be implemented. For the I/O devices, x_d is defined by the machine-specific setup (cp. Section 4.1.2), here 8. The machine-based maximal recursion depth usually varies among concrete OS implementations.

Scenario-Dependent Recursion Depth

Whereas the machine-based maximal recursion depth is an upper bound for the possible nesting of hardware interrupts, a more specific, use case based version of the indicator can be given in preparation for the scenario-aware point of view that will be presented in Section 6.4. Such a use case is given as a finite sequence of disruptive events. The scenarios constructible from that sequence (by mapping delays between the events to it) can be divided into three disjoint sets depending on the concrete SuIs:

- The set of best case scenarios: all scenarios where the distances between the events are large enough so that for all SuIs, all events can be treated without nesting. The recursion depth for these scenarios is 1 for all SuIs.
- The set of worst case scenarios: all scenarios where the delays between the events of the sequence are so small that it is impossible for any of the SuIs to process any of the events completely before the next event arises. This also holds for the last event of the sequence. The recursion depth for these scenarios is m for all SuIs where $m \leq n$ is the maximal subsequence of events without repetition.
- The set of all scenarios with event occurrence delays that lie between those of worst and best case scenarios. This set contains all these scenarios where the maximal recursion depth can vary from system to system.

Not all events that make up a sequence are able to change the recursion depth. Even with no explicit locking of all interrupt sources, the microprocessor automatically locks the currently processed interrupt. Hence, only maximal subsequences of different events can be the trigger increasing the recursion depth.

Definition 6.6 ***Maximal Sequence without Repetition***
A sequence without repetition is a sequence $e_0 \dots e_{n-1}$ *with* $e_\nu \in \overrightarrow{E}, \nu \in \{0, \dots n-1\}$ *being mutually distinct:* $\forall\, e_{\nu_1}, e_{\nu_2} : e_{\nu_1} \neq e_{\nu_2}$.
A sequence without repetition is maximal if $\forall\, e_\mu, \mu \notin \{0, \dots n-1\} : e_0 \dots e_{n-1} e_\mu$ *is no sequence without repetition.*

Having given the precise meaning of a maximal sequence without repetition, subsequences now need to be defined.

Definition 6.7 ***Subsequence***
A sequence $seq_s = e_i \dots e_j$ *is a subsequence of sequence* seq_t *if* $seq_t = e_0 \dots e_i \dots e_j \dots e_{n-1}$. *This is denoted by* $seq_s \sqsubset seq_t$.

Let $scen = (seq, \{x_0, \dots, x_{n-1}\}) \in \overrightarrow{E}^n \times \mathbb{N}_0^n$ be a given scenario that is neither a best nor a worst case scenario with respect to seq. Let $\{seq_{max_0} \dots seq_{max_j}\}$ be a set of maximal sequences without repetition that are all subsequences of seq, i.e. $\forall\, \iota \in \{0, \dots, j\} : seq_{max_\iota} \sqsubset seq$. The following holds for the maximal recursion depth:

$$recursion\_depth(scen) = \left|\{x_\nu | x_\nu \leq length(PAG_{\mathfrak{hdl}^b(e_\nu)}), e_\nu \in seq_{max_\iota}\}\right| \qquad (6.2)$$

For all maximal subsequences without repetition, a maximal recursion depth n_ι is derived by means of the inequation $x_\nu \leq length(PAG_{\mathfrak{hdl}^b(e_\nu)})$, i.e. the number of times when the delays between the events of the subsequence are so small that the previous disruption path was not finished and thus the system continues the recursive descent.

6.3 Data-Based Analysis – Model Checking

Some indicators depend on system runs of a SuI since they describe the worst case reaction of a given system on all possible inputs, i.e. sequences of interrupts. In contrast to the

previously defined indicators that are by their nature independent of any scenario, here a dynamic approach is chosen.
When simulating the behavior of a system modeled as an *ESF*, nothing less than an infinite number of simulations could cover all interrupt scenarios. Therefore, model checking (MC) is the optimal course of action: it allows for exhaustive state space investigation to derive statements about the respective indicator using temporal logic formulae.

6.3.1 Model Checking Foundations

Temporal logic model checking (TLMC), the model checking variant used in this thesis, was founded by Clarke and Emerson in the United States [Eme81], [CE82] [CES83] and by Quielle and Sifakis[1] in Europe [QS82] at almost the same time. It is a technique for verifying reactive and concurrent finite state systems [CGP00] with regard to a formal specification. Counterexamples are generated automatically if the model check proves the specification to be incorrect.
For a long time, state space explosion was the limiting factor that prevented large systems such as complex software to be verified by means of model checking. Techniques confronting this problem emerged during the past decades. Symbolic model checking with binary decision diagrams was founded by McMillan [McM92a] for synchronous models (e.g. of circuits), partial order reduction by Godefroid et al. [GP93] for asynchronous models (e.g. of communication protocols). Huge models – i.e. 10^{20} states and even more [BCM+92] – can be processed nowadays.

6.3.1.1 Modeling

The model that is used for MC is usually given as a state transition graph. One special type of such graphs is called Kripke structure. Kripke structures associate (or more precisely: label) each state with a set of atomic propositions (AP) that are true in this very state. All following basic definitions are according to [CGP00].

Definition 6.8 ***Kripke Structure, Fair Kripke Structure, Sequence***
A Kripke structure M over the set of atomic propositions $AP = \{p_1, \ldots, p_n\}$ is a quadruple $M = (S_K, R_K, L_K, S_{K_0})$ with:

1. *S_K being a finite set of states*
2. *$R_K \subseteq S_K \times S_K$ being a transition relation that is total, i.e.*

$$\forall\, s \in S_K : [\exists s' \in S_K : R_K(s, s')]$$

3. *L_K being a function $L_K : S_K \to 2^{AP}$ that labels every state with a set of atomic propositions that hold in this state*
4. *A set of initial states S_{K_0}*

A path from state s in a Kripke structure is an infinite sequence of states $\pi = s_0 s_1 \ldots$ with $\forall\, i : [R(s_i, s_{i+1})]$ and $s_0 = s$. A path is a calculation on a Kripke structure.

[1] Emerson, Clarke and Sifakis were granted the A.M. Turing Award of the ACM in 2007 for their pioneering work on model checking.

A fair Kripke structure is a 5-tuple $(S_K, R_K, L_K, F_K, S_{K_0})$ *with* S_K,R_K,L_K,S_{K_0} *as defined above and* $F_K \subseteq 2^{S_K}$ *a set of Büchi acceptance conditions [Büc62] or fairness constraints. With*

$$inf(\pi) = \{s \mid s = s_i \text{ for infinitely many } i\}$$

a sequence π *is defined fair* $\Leftrightarrow \forall f \in F_K : [inf(\pi) \cap f \neq \emptyset]$*, i.e. a path is fair if all constraints are true infinitely often along the path.*

Kripke structures can easily be derived from first order representations of the states and transitions of any system. First order representations themselves can be created from sequential or concurrent programs (cp. e.g. [CGP00] Chapter 2).

6.3.1.2 Specification

In the verification of transformational systems, a mere description of input/output relations, e.g. using the Hoare calculus, is sufficient. In reactive systems however, the specification of properties relies on the description of transitions between states. Temporal logic invented by Pnueli [Pnu77][2] is used to describe sequences of transitions within such a system. The basics for these specifications is the set of atomic propositions that hold in specific states.
The computation tree logic (CTL*) [CES83] [CE82] [EH86] formally describes properties of a computational tree. This tree is created by unwinding Kripke structures[3], i.e. preserving the initial state as the root state and then creating an infinite tree by unrolling all cycles and enumerating the infinite input state space.
Due to its extensive expressiveness, model checking for CTL* formulae is very complex and costly in terms of computational time and memory. Furthermore, there is often no need to explore the entire expressiveness provided. For these reasons, there exist two subsets of CTL*: the branching time computation tree logic CTL [BAMP81] and the linear time logic LTL [Pnu81].
As the indicators to be checked refer only to linear paths and not to branching trees, LTL is the right choice for formulating these in temporal logic (cp. Section 6.3.5). Note that however, the Kripke representation of *ESF* that will be given in Section 6.3.4.1 would allow for the entire expressiveness of CTL*.

Definition 6.9 ***Linear Time Logic (LTL)***
In LTL, every specification formula is composed of temporal operators. The following operators exist:

- **X** - *"Next time" - Unary operator that states if the property holds for the second state in the path*
- **F** - *"In the future" or "Eventually" - Unary operator that states if the property holds at any future state in the path*

[2] Pnueli received the Turing Award as well (1996).
[3] We do not distinguish between Kripke structures and fair Kripke structures where the meaning is clear from the context or the differences are irrelevant.

G - *"Globally" - Unary operator that states if the property holds for all future states in the path*

U - *"Until" - Binary operator that holds if there is one state along the path where the second property holds and at every preceding state the first property holds*

R - *"Release" - Binary operator that states if the second property holds along the path up to and including the first state where the first property holds*

An LTL formula is created by applying the following compositional rules:

1. $p \in AP \Rightarrow p$ *is a path formula*
2. f, g *are path formulae* $\Rightarrow \neg f, f \vee g, f \wedge g, \mathbf{X}f, \mathbf{F}f, \mathbf{G}f, f\,\mathbf{U}\,g, f\,\mathbf{R}\,g$ *are path formulae*

6.3.1.3 Checking Models against Specifications

The following term denotes that a path formula f holds along a sequence π of the Kripke structure M.

$$M, \pi \models f$$

The relation $\models$ is defined recursively as follows:

1. $M, \pi \models \neg g_1 \Leftrightarrow M, \pi \not\models g_1$
2. $M, \pi \models g_1 \vee g_2 \Leftrightarrow M, \pi \models g_1$ or $M, \pi \models g_2$
3. $M, \pi \models g_1 \wedge g_2 \Leftrightarrow M, \pi \models g_1$ and $M, \pi \models g_2$
4. $M, \pi \models (X)g_1 \Leftrightarrow M, \pi^1 \models g_1$
5. $M, \pi \models (F)g_1 \Leftrightarrow \exists k \in \mathbb{N} : M, \pi^k \models g_1$
6. $M, \pi \models (G)g_1 \Leftrightarrow \forall\, i \in \mathbb{N} : M, \pi^i \models g_1$
7. $M, \pi \models g_1(U)g_2 \Leftrightarrow \exists k \in \mathbb{N} : (M, \pi^k \models g_2$ and $\forall\, 0 \leq j < k : M, \pi^j \models g_1)$
8. $M, \pi \models g_1(R)g_2 \Leftrightarrow \forall\, j \in \mathbb{N}$, if $\forall\, i < j : M, \pi^i \not\models g_1)$ then $M, \pi^j \models g_2$

with g_1, g_2 being path formulae.

With the given definitions, the problem of checking a model M against any temporal logic formula f is to find the set of paths that fulfill the given specification:

$$\{\pi \in 2^S \mid M, \pi \models f\}$$

6.3.2 Tools for Model Checking

There exists a variety of mature tools and frameworks to find $\{\pi \in 2^S \mid M, \pi \models f\}$ for given M and f. Some commonly used non-commercial tools are the following:

- McMillan developed the symbolic model checker SMV [McM92b] based on ordered binary decision diagrams OBDD [Bry86] within his PhD Thesis [McM92a]. The SMV uses CTL as the specification calculus. There exists a widely used open source re-implementation of the SMV called NuSMV [CCGR99]. This newer program checks for both specification subsets, CTL and LTL. It also uses sequential extended regular expressions (SEREXP), a path-based variant of classical regular expressions to enrich the temporal logic formulae.
- The SPIN tool [Hol03] created by Holzmann [Hol97] uses the process meta language (PROMELA) to specify models that then can be checked for LTL properties.
- The UPPAAL tool environment [LPY97] is an integrated model checking tool that uses timed automata as the input language for models and checks those models against CTL specifications.
- Apart from these very general model checkers, there exists a variety of special purpose model checkers, e.g. to verify VHSIC hardware description language designs or to check safety constraints of programming languages.

This thesis has no intention of developing a new or of enhancing an existing model checking tool. All further work uses Kripke structures and LTL formulae as a de facto standard for model checking tools.

6.3.3 Model Checking Statecharts

To this day, model checking of statecharts is the subject of intensive research. There exists a variety of approaches and solutions, Bhaduri presents a good while incomplete survey [BR04]. In this section, not only are the conventional Harel statecharts considered, but also UML 2.0 statecharts as well as other dialects.

Harel Statecharts
In the early 1990ies, Day [Day93a] [Day93b] translated a subset of statecharts into ML (meta language) code [MTM97] for the theorem prover for high order logic HOL-Voss [Jef94]. This early approach restricts the statechart semantics considerably and does not allow transitions to cross any level of hierarchy.
Around that time, Kelb et al. also studied model checking of statecharts omitting history but already considering inter-level transitions [Kel95] and translated statecharts to OBDD [HK94].
Later Mikk et al. used extended hierarchical automata (EHA) [MLS97] to verify statecharts by translating them into both PROMELA for SPIN and the input language for the SMV [MLSH98] [Mik00]. This approach was the first that led to a usable tool chain. It still had a number of shortcomings though: no history, no timing, no data-transformations and only generated events as actions could be modeled.
Brockmeier verified Statemate statecharts [BW98a] by means of symbolic timing diagrams (STD), translated into CTL [BW98b]. The Siemens model checker [Fil97] is used to verify the requirements of reactive systems.
There also exists a translation of statecharts to Esterel [SSBD99], this approach evidently would require the means of verifying Esterel code. Since there are still problems with that

verification task, this approach only shifts rather than solves the problem at hand.
In 2000, Clarke and Heinle translated statecharts into SMV exploiting the modularity of the specification language [CH00]. Their so-called STP approach was to define a new basic language named ETL. The fact that the reported translation does not handle inter-level transitions greatly limits its applicability.
The researchers at OFFIS, Oldenburg University (Germany) developed the Statemate verification environment [BBD+99] by means of the VIS model checker [RHS+96], the basis of the commercial Telelogic product [BDW00]. This commercial product comes with unpublished algorithms and a template language instead of full temporal logic calculus. Despite that, the commercial Statemate ModelVerifier is the most powerful tool to check statecharts against specifications.
A translation from statecharts to Communicating Sequential Processes (CSP) [Hie01] and the verification by failures divergences refinement (FDR) was achieved by Fuhrmann and Hiemer [FH01]. A mapping from statecharts to the Verilog model checker [IEE95] is articulated by Qin et al. [TQC04].
A translation from statecharts to a labeled transition system (LTS) by means of basic process algebra is presented by Qian in [XQ05]. This approach offers the great advantage of being independent of any dedicated input language or format. When creating an LTS or Kripke model equivalent to the statechart model, the problem of model checking is reduced to just checking the LTS.

UML 2.0 Statecharts
Apart from the efforts to model check Harel statecharts, various approaches to verifying UML statecharts exist. Latella et al. converted UML 2.0 statecharts to PROMELA / SPIN [LMM99], as Lilius et al. [LP99a] did, even providing a tool called vUML [LP99b]. Unfortunately, none of the approaches considers the full UML statechart dialect but only subsets of it.
As for conventional statecharts, the inter-level transitions turn out to be the most difficult part. They impede a compositional approach and thus complicate the transformation rules considerably. However, Dong et al. showed how to apply hierarchical automata to UML statecharts [DWQQ01] [WDQ02] and outlined a way of model checking UML statecharts including hierarchy. Kwon [Kwo00] achieved a full translation of UML statecharts to the SMV model checker by means of conditional term rewriting systems.

Other Statechart Dialects
There exists a variety of statechart subsets and enhancements [vdB94]. In [IBDR03], a translation from communicating statecharts – an extension for statecharts that allows communication through buffered channels – is presented. Philipps et al. show how to verify μ-statecharts [PS98] using the model checker μ-cke [Bie97].
Alur et al. proposed hierarchical state machines as a variant of statecharts [AKY99]. The model checking input is then based on hierarchic reactive modules (HRM) that can then be symbolically model checked [AY01] by means of the BDD package that is part of VIS.
Büssow translated a combination of the specification language Z and statecharts into a generic intermediate format facilitating model checking by means of numerous tools [Büs03].

It would be unrewarding to adopt one of the numerous approaches and to enhance its formal translation schema to match the *ESF* specification: all presented methods still lack vital features and/or are not flexible enough to be adopted for the purposes of this thesis. Instead, the goal is to model check an *ESF* model directly or to translate the *ESF* models into a Kripke structure or Kripke model M_K that then serves as a universal vehicle to be used by any model checker.

6.3.4 Model Checking *ESF* Models

The common goal of all approaches surveyed is to provide a universal technique that can be applied for model checking of any possible statechart model M_{SC}. Since all presented approaches aim at generality, it is clear that none of them allows the exploitation of any inherent semantic properties of an *ESF* model M_{ESF} as it is. Having said that, in general, there are three different possible courses of action to conducting model checking with a model M_{ESF}.

1. Formulating model check algorithms that directly operate on LTL formulae and the model M_{ESF} in order to determine the set $\{\pi \in 2^S \mid M_{ESF}, \pi \models f\}$. Any model M_{ESF} possesses a formally defined syntax and semantics, it contains all necessary information, i.e. the set of atomic propositions on which to check. From a theoretical perspective, formulating such an algorithm is possible.

2. A sequential translation of the *ESF* models to statecharts exploiting their semantic peculiarities and then from statecharts to a labeled transition system such as a Kripke structure (including possible intermediate steps) can be defined assuming the semantic transformation rules given in Chapter 3. This way of applying model checking practices to a model M_{ESF} is very easy, in terms of the necessary rule set, and straightforward. In a nutshell, a chain $M_{ESF} \xrightarrow{trans} M_{SC} \xrightarrow{trans} M_K$ needs to be created.

3. A semantic rule set can be given that allows for the direct translation of *ESF* models to a labeled transition system by exploiting the very peculiarities directly. This translation $M_{ESF} \xrightarrow{trans} M_K$ then allows for introducing considerable optimizations for model checking algorithms.

The issue with the first approach is that the definition of an *ESF*-based algorithm for model checking, although theoretically possible, is not a feasible solution. Reinventing the results of two decades of research and facing the same complexity problems does not at all seem desirable. Major difficulties arise, at the latest, when the concepts of recursion and long term history (that demand multidimensional data structures such as lists) are to be verified directly. Therefore, this theoretical solution is ruled out.
But even the second approach has serious shortcomings, especially when compared to the third. The translation of *ESF* models into conventional statecharts would make it impossible to exploit all existing peculiarities that could make the target representation simpler. However, for all *ESF* constituents except CTS, a translation into conventional statecharts and a later translation to a Kripke structure would be possible (although not beneficial). The reason is that those constituents have a kind of “compile time” semantics. If such a

compile time approach were applied to CTS as well, the entire cartesian product of the implicated *AND* states would have to be explicitly elaborated. To avoid this, a runtime semantics is defined for the CTS. This runtime semantics must be incorporated into the translation rules.
Moreover, the two translation processes that would be necessary result in a less specific exploitation than if only one such process was applied. The result is that with the second option, the Kripke structures would be far more bulky.
The intermediate step would not only result in extremely large models to be processed by the model checking tool, but also in unnecessarily dense ones. Usually, only a few parts of the bulky models are reachable. Büssow for example reports in his PhD thesis [Büs03] that a statechart model consisting of 92 states and 30 events resulted in a model M_{SC} with $3 \cdot 10^{25}$ states where only $3 \cdot 10^{6}$ states were reachable. Taking the magnitude of those figures and the IHF model sizes as a basis for this work, it is evident that this approach leads to severe complexity issues.

Therefore, in this thesis, the third approach is chosen: since all models that are to be checked are based on the same kind of systems and follow fixed modeling rules, there exists a variety of properties that can be exploited to simplify the model checking process. This approach is consistent with the paradigm of use case tailored formal methods developed throughout this thesis. As the next section shows, in particular the cartesian transition set allows for massive optimization. The semantic properties taken into account for this optimization are the *ESF* modeling rules on the one hand and the defined application rules for modeling operating systems.

6.3.4.1 Transformation Rules: Translating *ESF* into Kripke Models

As Definition 6.8 shows, Kripke structures – or Kripke models M_K – are based on a set of atomic propositions. For any translation $M_{ESF} \xrightarrow{trans} M_K$, this set AP_K comprises the following subsets:

1. The set of all reachable system configurations P_{conf}, represented by an n-tuple of the special proposition $in(s)$, for a statechart model being in state s, one proposition for each concurrent component

2. The set of explicitly given statechart properties such as conditions, variables and expressions, called P_{exp}

3. The implicitly given semantical *ESF* properties such as path conditions or recursion variables, P_{imp}

The distinct temporal logic formulae that represent the indicators in question (cp. Section 6.3.5) could be formulated based solely on the proposition $in(s)$. Elaborating this fact, one could immediately reduce the set of atomic propositions relevant for the Kripke structure thus creating smaller Kripke structures (fewer number of states).
Unfortunately, this simple translation would also result in quite "crowded" Kripke structures, leading to an accumulation of transitions $\in R_K$ at semantical hot-spots such as recursive model parts or states that precede a split state. In fact, the models M_K would

be nearly completely intermeshed. In any event, such a translation would preserve the semantic peculiarities of *ESF* models instead of taking advantage of them. Hence, the model checking results would still bear these very properties.
As a consequence, any counterexample produced by model checking must be validated to determine if it represents a correct *ESF*-compliant path. This is most undesirable since an unnecessarily large number of invalid counterexamples would be produced by the model checker and the validation of the results is a complex step to be avoided. Even if no valid counterexample is produced, numerous invalid ones would still be checked.
The atomic properties that are subject to model checking are only state specific, never path dependent. From there it follows that regardless of how a specific Kripke state was entered, the properties will have the same value. In the *ESF* this is not necessarily true since a path can influence this set of propositions considerably. In other words: the check of the result mentioned could not be done by means of the Kripke structure but only by means of the *ESF* model itself.

From these considerations it follows that for the translation $M_{ESF} \xrightarrow{trans} M_K$, additional data from the models has to be taken into account. Accordingly, the set of atomic propositions relevant to the Kripke model M_K is defined as

$$AP_K \stackrel{\text{def}}{=} P_{conf} \cup P_{exp} \cup P_{imp} \; with \; P_{conf} \cap P_{exp} \cap P_{imp} = \emptyset$$

This definition detaches the Kripke states from the mere statecharts basic configurations. By this, the semantics of all *ESF* constituents is exploited before the model checking itself is performed. Since the Kripke states and the statecharts basic configurations no longer directly correspond, the transition relation definition and the definition of the Kripke states are much more complex. By taking a complex translation into account, one avoids the problems mentioned above: the resulting Kripke structure can be optimized already, i.e. it is larger but sparse, it thus contains fewer unreachable states and there exist no hot-spots.

A translation $\xrightarrow{trans}$: $M_{ESF} \xrightarrow{trans} M_K$ with M_{ESF} being an *ESF* model following the given modeling approach and $M_K = (S_K, R_K, L_K, S_{K_0})$ being a Kripke structure results in the equivalency of the following set S:

$$\{\pi \in 2^S \mid M_{ESF}, \pi \models f\} \Leftrightarrow \{\pi \in 2^S \mid M_K, \pi \models f\}$$

By this equivalency, the problem of model checking an *ESF* statechart model is reduced to the classical problem of model checking as discussed above [CGP00]. A similar approach is presented in [XQ05].

Since the translation $\xrightarrow{trans}$ is comprehensive, it is another crucial point to decide on the means of defining it. There are three alternatives:

- **Graph grammar.** Since both *ESF* models and Kripke structures are graph descendants, a graph grammar [Ehr88] would be a quite natural approach to describe a conversion between them. On the contrary, graph grammars are neither intuitive nor satisfactorily manageable for the intended purpose.

- **Term rewriting system.** In this approach contrary to graph grammars, translation rules are defined using a textual form. This is a feasible solution due to the simplicity

of the rules. But although there is a variety of textual representations of statecharts (e.g. [KP91]), it would require much additional work to create such a representation for the *ESF*, e.g. based on the XMI format [OMG05].

- **Transformation rule set.** In analogy to the *ESF* semantics definition, a set of transformation rules based on the formal definition of *ESF* syntax and semantics can be given in a short and formally precise way.

The transformation rule approach is closest to the nature of the modeling method whereas the other two are bound to the mere outer form of a model. Therefore, it suits the purposes of this thesis best to give the translation in a transformation rule set.

6.3.4.2 Transformation Algorithm

The basic idea of the direct transformation approach is to subdivide the translation from *ESF* to Kripke into two separate steps. In the first step, each concurrent component is treated separately. For each such component ω, a certain preliminary Kripke model $M^{\perp}_{K,\omega}$ is created.
Since the CTS mechanism makes a grouped approach obsolete (cp. Section 3.5), we can assume that all models M_{ESF} use the cartesian transition set mechanism alone and no simple *AND* states to represent concurrency. From that, a global property (i.e. a natural number) specifying the currently considered concurrent component can be assigned. This property is crucial for gaining a simple and compact translation rule set for the first step as well as an efficient rule to later combine the preliminary Kripke models.
Informally, a substate of an *AND* state cannot be left until its final state is reached. Formally, this is analogous to the fact that there is no case in which a full transition is different from its full compound transition.

$$\forall\, fct \in FCT, fct = ts_0, \ldots, ts_n : (\nexists\, ft \in FT, ft = ts_0, \ldots, ts_n : ft \neq fct),\ ts_0 \ldots ts_n \in TS$$

The concurrent component in question can therefore be treated independently of the others. Subsequently, there is one dedicated preliminary Kripke model $M^{\perp}_{K,\omega}$ per orthogonal partition of the model M_{ESF}. For every transition that originates within the scope of the concurrent component, it holds that its target state is not located outside the scope. Let r_ω be the root state of concurrent component ω:

$$\forall\, t \in T : [source(t) \in \rho^*(r_\omega) \Rightarrow \mathfrak{arena}(t) = \rho^*(r_\omega)]$$

During the first step, all transformation rules defined later are applied in the order that is predefined by the original statechart semantics. From this it follows that all necessary semantic properties of conventional statecharts – such as for example the greediness property – are preserved while the rules themselves exploit the semantic benefits of the *ESF*.
Since the progress relation of statecharts is not guaranteed to terminate, an additional termination condition is stated:
When no application of any transformation rule creates a new Kripke relation tuple, the transformation terminates.

No concrete inputs (i.e. occurring events or evaluated conditions or variables) are considered. Therefore, all path conditions are omitted and only the currently valid conditions

for split and combine states are relevant (they are implicit, i.e. $\in P_{imp}$). The actual non-concurrent portions of a statechart model are traversed like hierarchical graphs. This further enhances the idea of PAGs and IAGs used for graph-based model analysis.

A state of a preliminary Kripke model $M^{\perp}_{K,\omega}$ differs from a state of the final Kripke model in three ways:

1. The first part P_{conf} only contains basic states, no configurations.
2. A dedicated partition of the state, P_{evt}, contains the event that triggered the preceding transition.
3. A dedicated partition of the state, P_{act}, contains the action carried out during the translated transition.

The additional information preserved by the latter two sets is required to later assemble the preliminary models $M^{\perp}_{K,\omega}$.
A Kripke state $\in S^{\perp}_{K,\omega}$ of a preliminary Kripke model $M^{\perp}_{K,\omega}$ has the following form:

$$\underbrace{\left(\underbrace{[in(s)]}_{\in P_{conf}}, \underbrace{[p_0^{imp},\ldots,p_{k'}^{imp}]}_{\in P_{imp}}, \underbrace{[p_0^{exp},\ldots,p_{l'}^{exp}]}_{\in P_{exp}}, \underbrace{[p_0^{evt},\ldots,p_{m'}^{evt}]}_{\in P_{evt}}, \underbrace{[p_0^{act},\ldots,p_{n'}^{act}]}_{\in P_{act}}\right)}_{=s'_K \in S^{\perp}_{K,\omega}} \tag{6.3}$$

Note that since only one concurrent component is considered, for this definition the set of configurations P_{conf} is equal to the set of basic states $\subset \tilde{S}$. The actions are only part of the intermediary Kripke states and are evaluated by the composition rule (**AND** rule, see Section 6.3.4.3.6). They are not part of the final Kripke states $\in S_K$.

The **AND** rule that constructs the final Kripke structure M_K from the preliminary structures $M^{\perp}_{K,\omega}$ in the second step produces an intermediate format, which is defined as:

$$\underbrace{\left(\underbrace{[in(s_0),\ldots,in(s_n)]}_{\in P_{conf}}, \underbrace{[p_0^{imp},\ldots,p_{k'}^{imp}]}_{\in P_{imp}}, \underbrace{[p_0^{exp},\ldots,p_{l'}^{exp}]}_{\in P_{exp}}, \underbrace{[p_0^{evt},\ldots,p_{m'}^{evt}]}_{\in P_{evt}}, \underbrace{[p_0^{act},\ldots,p_{n'}^{act}]}_{\in P_{act}}\right)}_{=s''_K \in S_{K,\omega}} \tag{6.4}$$

Since the basic idea behind all transformation rules is to exploit the fact that the three subsets of AP_K are disjoint, i.e. $P_{conf} \cap P_{exp} \cap P_{imp} = \emptyset$, any Kripke state $s'_K \in S^{\perp}_{K,\omega}$ – as well as $s_K \in S_K$ – relies on this fact. The final states $s_K \in S_K$ are defined by using AP_K as follows:

$$\underbrace{\left(\underbrace{[in(s_0),\ldots,in(s_n)]}_{\in P_{conf}}, \underbrace{[p_0^{imp},\ldots,p_{k}^{imp}]}_{\in P_{imp}}, \underbrace{[p_0^{exp},\ldots,p_{l}^{exp}]}_{\in P_{exp}}\right)}_{=s_k \in S_K} \tag{6.5}$$

It is clear that the transition between each of the three final partitions is dictated mainly by different parts of the semantics. In other words, one can define three distinct relations, each describing parts of the transition relation $\Delta : S_K \rightarrow S_K$, $\Delta \subset R_K$:

$$\Delta_{conf} : P_{conf} \rightarrow P_{conf} \tag{6.6}$$

$$\Delta_{exp} : P_{exp} \rightarrow P_{exp} \tag{6.7}$$

$$\Delta_{imp} : P_{imp} \rightarrow P_{imp} \tag{6.8}$$

The Δ_{conf} relation is the progress relation that is defined by the statechart semantics (cp. Section 2.4.4) which determines how a statechart model can be "executed" in a step-by-step manner. The Δ_{exp} relation is as well predefined by the chosen semantics. It is based on the action mechanism of statecharts.
The entirely new part for *ESF* models is the third part, Δ_{imp}. It is completely defined by a composition of the semantic transformation rules for the *ESF* constituents and the overall semantics. The transformation rule set thus starts with elaborating Δ_{imp} and hence adopting the necessary parts of Δ_{conf} and Δ_{exp} as well.

The rules that are applied during the first step ensure that all semantic peculiarities of the *ESF* constituents are exploited as discussed. The later composition makes use of the CTS semantics and thus eases the transformation considerably. Finally, the model checking itself takes care of the recursion. When the model checker unwinds the Kripke model into an infinite computational tree, all possible paths including the recursive descent and ascent are taken into account.
For this, the only requirement is to translate the transitions originating from the recursion delimiter state (see Definition 3.18) according to the translation rules.

It is not necessary to validate any generated counterexamples: as the Kripke models are created based on the step relation and the *ESF* semantics, there can be no path in the Kripke models that does not correspond to a correct path in the *ESF* model. Thus, the problem of validating paths outside the model checking (cp. Section 6.3.4.1) is avoided.

6.3.4.3 Transformation Rule Set

The first 15 rules make up the first step of the transformation algorithm. As mentioned above, they are applied in a predefined order given by the statechart semantics. For using the two rather complex definitions of Kripke states in the following, a few rules apply:

- There exists an additional value any proposition p can take: *undefined*, i.e. $p = \lightning$.
- Any undefined proposition is omitted in any formula unless it changes its value.
- The set membership of the propositions, i.e. if a proposition p is either $\in P_{conf}$, $\in P_{imp}$, $\in P_{exp}$ or $\in P_{act}$, is not given unless it is ambiguous. A situation where ambiguity occurs is for example when generated events are handled in order to translate broadcast communication.

Note that in the following, *ESF* definitions will not be marked separately, whereas Kripke elements are, e.g. $s \in S$ depicts a statechart state, $s_K \in S_K$ a Kripke state.

In general, a transformation rule has the following format:

$$\textbf{NAME}\ \frac{\textit{ESF constituents}}{\textit{Kripke states, relation, rule}}\ \Downarrow$$

6.3.4.3.1 Construction of the Set $S^{\perp}_{K,\omega}$

The set of Kripke states for all preliminary Kripke structures can be defined in two different ways:

1. All Kripke states, i.e. all possible permutations of the atomic propositions are assumed to exist from the beginning. After applying all rules, the set of states is limited to a subset where all states are part of the relation $R^{\perp}_{K,\omega}$ at least once.

2. A creation rule is added to the rules that creates a not yet existing Kripke state on demand. Such a creating function $\mathfrak{cre} : S \rightarrow 2^{S^{\perp}_{K}}$ would basically perform the following pseudo-code:

```
kripkeState *k createOnDemand(statechartState s){
if (! existingKripkeStateFor(s))
  return createStateFor(s);
 else
  return fetchStatesFromSet(s); // can be multiple
}
```

The major shortcoming of the first option is that all rules must implicate changes to all existing Kripke states related to the given *ESF* constituent in order to not accidentally create micro Kripke structures that cannot be merged for each rule that is applied. Having said this, it is clear that by creating all permutations, all Kripke states (whether they should be part of the model or not) become part of R_K, thus the models would not only be maximal in size and density but also not fully equivalent to the *ESF* model.
From these considerations it follows that an explicit creation procedure must be used. To avoid cluttering the rules with the formally correct embodiment of a creation function (which is set-theoretically complex), the following notation is used:

$$\textbf{RULE}\ \frac{--\,s_1\,--}{\begin{array}{c}([in(s_1)],\dots):\\ (([in(s_1)],\dots),--) \in R^{\perp}_{K,\omega}\end{array}}\ \Downarrow$$

The term "$([in(s_1)],\dots)$:" denotes the implicit use of the creation function $\mathfrak{cre}$ for s_1 when needed in the second part of the formula (after the ":") where the transformation itself is described. In most cases, the transformation is a simple addition to the relation $R^{\perp}_{K,\omega}$.
The unspecific "..." are a wildcard for any possible combination of propositions that appears. If there yet exists no set of Kripke states for s_1, the least specific one, i.e. $([in(s_1)], [\,], [\,], [\,], [\,])$ is created, i.e. added to the set. When a rule is actually specified, the more detailed notation $([in(s)], [\dots], [\dots], [\dots], [\dots])$ is used to specifically address the partitions of propositions. This degree of precision is required for example when one part has to be set such as $([in(s)], [c_{imp_1}, \dots], [\dots], [\dots], [\dots])$ or $([in(s)], [\dots], [\,], [\dots], [\dots])$.

Since there is a difference between the null event and null action and the undefined value, a general function for converting events / actions to propositions is defined as follows:

Definition 6.10 ***Conversion Functions*** $(\mathfrak{con}_E, \mathfrak{con}_A)$
The conversion function for events $\mathfrak{con}_E : \tilde{E} \rightarrow P_{evt}$ *and the conversion function for actions* $\mathfrak{con}_A : \tilde{A} \rightarrow P_{evt}$ *are defined as:*

$$\mathfrak{con}_E(e) \begin{cases} e & : e \neq \lambda \\ \lrcorner & else \end{cases}, \qquad \mathfrak{con}_A(a) \begin{cases} a & : a \neq \mu \\ \lrcorner & else \end{cases}$$

The application of any rule is written using the application operator:

Definition 6.11 ***(Default) Application Operator***
The following notation stands for applying the rule **NAME** *to the given ESF constituent* $(element)$*:*

$$\xrightarrow{\textbf{NAME}} (element)$$

If element does not match the upper side of **NAME***, the application is simply disregarded. The following short notation applies any matching rule to element:*

$$\xrightarrow{\textbf{RULE*}} (element)$$

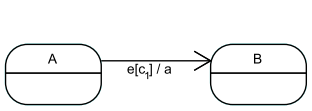
6.3.4.3.2 General, Sequential and Synchronization Transition Rules

For any *ESF* transition (with or without a guarding condition) originating from and leading to a basic state, the following transformation rules expands the corresponding preliminary Kripke structure with $s_1, s_2 \in \tilde{S}$, $e \in E$, , $a \in \tilde{A}$ and s_1, s_2 *both* being basic states:

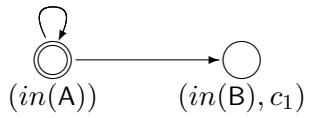
$$\textbf{TRANS 1} \quad \frac{s_1, s_2, t = (s_1, e/a, s_2)}{\begin{array}{c}([(in(s_1)], \dots), ([(in(s_2)], \dots) \ : \\ (([(in(s_1)], \dots), ([(in(s_1)], \dots)), \\ (([(in(s_1)], \dots), ([(in(s_2)], [\lrcorner], [\lrcorner], [\mathfrak{con}_E(e)], [\mathfrak{con}_A(a)])) \in R^{\perp}_{K,\omega}\end{array}} \Downarrow \tag{6.9}$$

$$\textbf{TRANS 2} \quad \frac{s_1, s_2, t = (s_1, e[c]/a, s_2)}{\begin{array}{c}([(in(s_1)], \dots), ([(in(s_2)], \dots) \ : \\ (([(in(s_1)], \dots), ([(in(s_1)], \dots)), \\ (([(in(s_1)], \dots), ([(in(s_2)], [\lrcorner], [c_1], [\mathfrak{con}_E(e)], [\mathfrak{con}_A(a)])) \in R^{\perp}_{K,\omega}\end{array}} \Downarrow \tag{6.10}$$

The term *ESF transition* also includes all conventional statechart transitions. Since the *ESF* in general contains all conventional statechart elements, from now on only *ESF* components will be mentioned, representing the conventional statechart constituents as well.
Preliminary Kripke structures $M^{\perp}_{K,\omega}$ can be constructed from these two transition rules depending on whether an explicitly given condition is part of the transition or not. Figure 6.5 shows the application of **TRANS 2**. The simple statechart example (Figure 6.5(a)) contains a transition labeled with $e[c_1]/a$. Figure 6.5(b) depicts the resulting preliminary Kripke model. Note that the resulting model would look very similar if **TRANS 1** were applied, with only the difference that the state would be $(in(\mathsf{B}))$ instead of $(in(\mathsf{B}), c_1)$.



(a) M_{ESF} with 1 transition



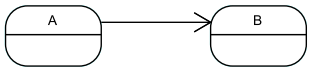
(b) application of **TRANS 2**: $M^{\perp}_{K,\omega}$ for any c_1

Figure 6.5: Sample application of the general transition rule

Both rules induce a self loop for the Kripke state representing the source of the transition. If a transition cannot be taken during a system step, the system stays in the current state – a self loop is created in the preliminary Kripke structure. Hence, self loops always occur when *ESF* transitions are guarded by a condition such as a synchronization condition. When dealing with transitions that do not have a guarding condition at all, the semantics [HN96] dictates that, according to the progress relation, a self loop is possible since the system may traverse further triggered by an event noticeable to another orthogonal component. Consequently, the only cases in which no self loop can possibly occur are the ones in which the source state is left without any explicit triggering: a trivial transition or a sequential transition as in Definition 3.1. Therefore, with $s_1, s_2 \in \tilde{S}$ and s_1, s_2 *both* being basic states and $e \in \{\lambda,\, en(s_1)\} \subset E$, the following special rule is defined:

$$\textbf{TRANS 3}\ \frac{s_1, s_2, t = (s_1, e/\mu, s_2)}{\begin{array}{c}([[in(s_1)], \dots), ([[in(s_2)], \dots)\ : \\ (([[in(s_1)], \dots), ([[in(s_2)], [\lrcorner], [\lrcorner], [\lrcorner], [\lrcorner])) \in R^{\perp}_{K,\omega}\end{array}} \Downarrow \tag{6.11}$$

Figure 6.6(a) shows the modified example statechart model that now has one sequential transition, leading to a preliminary Kripke model without a self loop, depicted in Figure 6.6(b).



(a) M_{ESF} with 1 sequential transition

(b) application of **TRANS 3**: $M^{\perp}_{K,\omega}$ without self loop

Figure 6.6: Sample application of the general transition rule

For transitions reacting on path events, the following rules are defined with $s_1, s_2 \in \tilde{S}$, $e \in \overrightarrow{E}$, $a \in \tilde{A}$ and s_1, s_2 *both* being basic states:

$$\textbf{TRANS 4}\ \frac{s_1, s_2, t = (s_1, e/a, s_2)}{\begin{array}{c}([(in(s_1)], \dots), ([(in(s_2)], \dots)\ : \\ (([(in(s_1)], \dots), ([(in(s_1)], \dots)), \\ (([(in(s_1)], \dots), ([(in(s_2)], [\mathfrak{pc}(e)], [\lrcorner], [\mathfrak{con}_E(e)], [\mathfrak{con}_A(a)])) \in R^{\perp}_{K,\omega}\end{array}} \Downarrow \quad (6.12)$$

$$\textbf{TRANS 5}\ \frac{s_1, s_2, t = (s_1, e[c]/a, s_2)}{\begin{array}{c}([(in(s_1)], \dots), ([(in(s_2)], \dots)\ : \\ (([(in(s_1)], \dots), ([(in(s_1)], \dots)), \\ (([(in(s_1)], \dots), ([(in(s_2)], [\mathfrak{pc}(e)], [c_1], [\mathfrak{con}_E(e)], [\mathfrak{con}_A(a)])) \in R^{\perp}_{K,\omega}\end{array}} \Downarrow \quad (6.13)$$

These rules differ from **TRANS 1** and **TRANS 2** only with respect to the implicit propositions: here, the initial path condition corresponding to the triggering path event is included in the state.

Finally, for synchronization transitions as introduced in Definition 4.3, the rule **TRANS 2** applies with c being the synchronization condition.

6.3.4.3.3 Condition and Variable Branching Rule

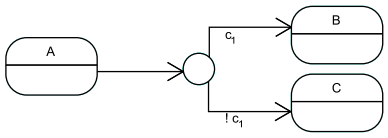
As described in Section 4.2.2.4, branches can be modeled in two different ways: using either variables/expressions or conditions. The transformation rules **BRANCH 1** and **BRANCH 2** are defined with $ts_1, \dots, ts_n \in \widetilde{TS}$, $a \in \tilde{A}$, $con \in COND$ and $s_1, \dots, s_n$ *all* being basic states:

$$\textbf{BRANCH 1}\ \frac{\begin{array}{c}ts_1 = (s_1, e/a, con)\,, e \in \{\lambda, en(s_1)\} \\ ts_2 = (con, \lambda[c]/\mu, s_2), ts_3 = (con, \lambda[\neg c]/\mu, s_3)\end{array}}{\begin{array}{c}([(in(s_1)], \dots), ([(in(s_2)], \dots), ([(in(s_3)], \dots)\ : \\ (([(in(s_1)], \dots), ([(in(s_2)], [\lrcorner], [c], [\lrcorner], [\mathfrak{con}_A(a)])), \\ (([(in(s_1)], \dots), ([(in(s_3)], [\lrcorner], [\neg c], [\lrcorner], [\mathfrak{con}_A(a)])) \in R^{\perp}_{K,\omega}\end{array}} \Downarrow \quad (6.14)$$

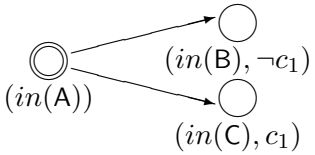
$$\textbf{BRANCH 2}\ \frac{\begin{array}{c}ts_1 = (s_1, e/a, con)\,, e \notin \{\lambda, en(s_1)\} \\ ts_2 = (con, \lambda[c]/\mu, s_2), ts_3 = (con, \lambda[\neg c]/\mu, s_3)\end{array}}{\begin{array}{c}([(in(s_1)], \dots), ([(in(s_2)], \dots), ([(in(s_3)], \dots)\ : \\ (([(in(s_1)], \dots), ([(in(s_1)], \dots)), \\ (([(in(s_1)], \dots), ([(in(s_2)], [\lrcorner], [c], [\mathfrak{con}_E(e)], [\mathfrak{con}_A(a)])), \\ (([(in(s_1)], \dots), ([(in(s_3)], [\lrcorner], [\neg c], [\mathfrak{con}_E(e)], [\mathfrak{con}_A(a)])) \in R^{\perp}_{K,\omega}\end{array}} \Downarrow \quad (6.15)$$

$$\textbf{BRANCH 3}\ \frac{\begin{array}{c} ts_1 = (s_1, e/a, con), \\ ts_2 = (con, \lambda[v_1 \doteqdot v_{x_1}]/\mu, s_2), \\ \dots \\ ts_n = (con, \lambda[v_1 \doteqdot v_{x_n}]/\mu, s_n) \end{array}}{\begin{array}{c} ([(in(s_1)], \dots), ([(in(s_2)], \dots), \dots, ([(in(s_n)], \dots) \ : \\ (([(in(s_1)], \dots), ([(in(s_2)], [\lightning], [v_1 \doteqdot v_{x_1}], [\mathfrak{con}_E(e)], [\mathfrak{con}_A(a)])), \\ \dots \\ (([(in(s_1)], \dots), ([(in(s_n)], [\lightning], [v_1 \doteqdot v_{x_n}], [\mathfrak{con}_E(e)], [\mathfrak{con}_A(a)])) \in R^{\perp}_{K,\omega} \end{array}} \Downarrow \tag{6.16}$$

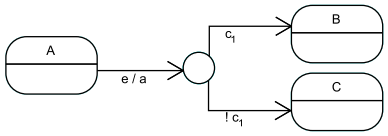
The distinguishing factor between the two methods of branching is the fact that by using a conditional branch, a self loop can be avoided when a sequential transition is used. See Figures 6.7(c) , 6.7(d) and 6.7(a) respectively 6.7(b).
Only binary branching can be realized this way, whereas by means of variables, a variety of different paths can be branched out and the resulting Kripke model always contains a self loop (see Figures 6.7(e) and 6.7(f)).



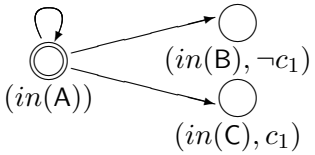
(a) M_{ESF} with conditional branch, sequential transition



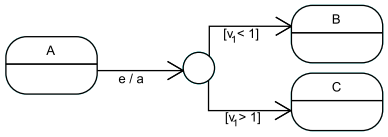
(b) application of **BRANCH 1:** $M^{\perp}_{K,\omega}$



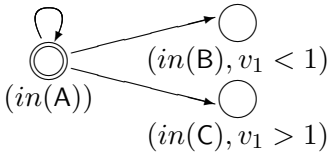
(c) M_{ESF} with conditional branch



(d) application of **BRANCH 2:** $M^{\perp}_{K,\omega}$



(e) M_{ESF} with value selection branch



(f) application of **BRANCH 3:** $M^{\perp}_{K,\omega}$

Figure 6.7: Sample application of branching rules **BRANCH 1**, **BRANCH 2** and **BRANCH 3**

6.3.4.3.4 Split / Combine State Rule

When a split state is the origin or target of a transition, the resulting $M^{\perp}_{K,\omega}$ is similar to the one that is constructed by application of rule **BRANCH 3** as depicted in Figures 6.7(e) and 6.7(f).

The major difference in the formal definition of the transformation rule is the fact that the branching values are not given explicitly in the model but are provided by means of path events and path conditions. Since in this implicit branching it cannot happen that the system does not take any outgoing transition, no self loop is possible (cp. Figure 6.7(d)).

$$\textbf{SPLIT}\ \frac{\begin{array}{c} t_1 = (s_1, e/a, sp) \\ t_2 = (sp, \lambda/\mu, s_2), \ldots, t_n = (sp, \lambda/\mu, s_n) \end{array}}{\begin{array}{c} ([(in(s_1)], \ldots), ([(in(s_2)], \ldots), \ldots, ([(in(s_n)], \ldots)\ : \\ (([(in(s_1)], \ldots), ([(in(s_2)], [pc = (\ldots, i_{\mathfrak{cat}(sp)} = \mathfrak{bid}(s_2), \ldots)], \\ [\lrcorner], [\mathfrak{con}_E(e)], [\mathfrak{con}_A(a)])), \\ \ldots \\ (([(in(s_1)], \ldots), ([(in(s_n)], [pc = (\ldots, i_{\mathfrak{cat}(sp)} = \mathfrak{bid}(s_n), \ldots)], \\ [\lrcorner], [\mathfrak{con}_E(e)], [\mathfrak{con}_A(a)])) \in R^{\perp}_{K,\omega} \end{array}} \Downarrow \tag{6.17}$$

$$\textbf{COMBINE}\ \frac{\begin{array}{c} t_1 = (s1, e_1/a_1, co), \ldots, t_{n-1} = (s_{n-1}, e_{n-1}/a_{n-1}, co) \\ t_n = (co, \lambda/\mu, s_n) \end{array}}{\begin{array}{c} ([(in(s_1)], \ldots), \ldots, ([(in(s_{n-1})], \ldots), ([(in(s_n)], \ldots)\ : \\ (([in(s_1)], [pc = (\ldots, i_{\mathfrak{cat}(co)} = \lrcorner, \ldots)], \\ [\lrcorner], [\mathfrak{con}_E(e_1)], [\mathfrak{con}_A(a_1)]), ([in(s_n)], \ldots)) \\ \ldots \\ (([in(s_{n-1})], [pc = (\ldots, i_{\mathfrak{cat}(co)} = \lrcorner, \ldots)], \\ [\lrcorner], [\mathfrak{con}_E(e_{n-1})], [\mathfrak{con}_A(a_{n-1})]), ([in(s_n)], \ldots)) \in R^{\perp}_{K,\omega} \end{array}} \Downarrow \tag{6.18}$$

The term $pc = (\ldots, i_{\mathfrak{cat}(sp)} = \mathfrak{bid}(s_n), \ldots)$ denotes the part of the path condition that corresponds to the split state in question.

For combine states, the branch category identifier that corresponds to the combine state in question is set to $\lrcorner$. This represents the truncation of path conditions as described in Section 3.2.

6.3.4.3.5 Hierarchical State Rule (XOR Rule)

All transition rules so far are defined on the basic state level only. The **XOR** rule allows for the treatment of hierarchy and inter-level transitions. The fact that the Kripke models use the basic system configuration eases the problem of inter-level transitions considerably. In the final Kripke states, only basic states have to be regarded, hence changes in the state hierarchy that occur when basic states are left and entered are of no interest. In short: no $in()$-statements for non-basic states, i.e. no hierarchical information is necessary to construct Kripke models:

$$\underbrace{[in(s_0), \ldots, in(s_n)]}_{\in P_{conf}}$$

This part can be used to cover hierarchy and inter-level transitions without any modification. Three general cases are possible when dealing with transitions originating from and leading to non-basic states:

1. The non-basic state is the source of the transition in question
2. The non-basic state is the target of the transition
3. Both, source and target are non-basic states

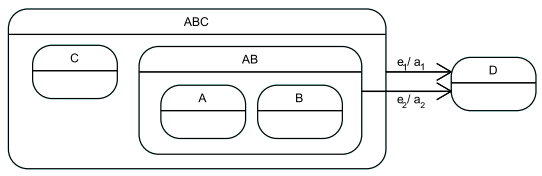
Since history symbols must be considered, the first two alternatives must be treated differently, the third case then is just a functional concatenation of the first two.

6.3.4.3.5.1 Non-Basic Source State

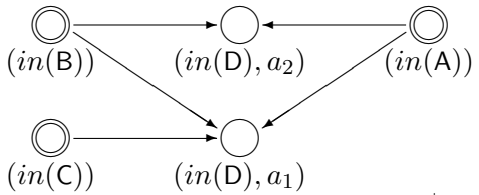
As history symbols do not influence this constellation at all, this is the simplest case. The approach is very similar to the one exploited for the IAG (see Section 6.2.2). The transition is taken as an outgoing transition for all basic states that are located anywhere downwards the hierarchy. With $t, t_0, t_1 \in \tilde{T}$, $s_1, s_2, s_{1_0}, \ldots s_{1_n} \in \tilde{S}, e \in \tilde{E}, a \in \tilde{A}$ and s_1 non-basic, s_2 basic, the rule is given as follows:

$$\textbf{XOR SRC}\ \frac{t = (s_1, e/a, s_2)}{\begin{array}{c} s_{1_0}, \ldots, s_{1_n}, \{s_{1_0}, \ldots, s_{1_n}\} \subseteq \rho^+(s_1) : \\ \xhookrightarrow{\textbf{RULE*}} (t_0 = (s_{1_0}, e/a, s_2)), \ldots, \xhookrightarrow{\textbf{RULE*}} (t_n = (s_{1_n}, e/a, s_2)) \end{array}} \Downarrow \tag{6.19}$$

Figure 6.8(b) gives a sample application of this rule. There are two distinct preliminary Kripke states $(in(\mathsf{D}), a_1)$ and $(in(\mathsf{D}), a_2)$ since the statechart model M_{ESF} as depicted by Figure 6.8(a) has two outgoing transitions labeled with e_1/a_1 and e_2/a_2 emerging from different hierarchical levels.



(a) the source of the transitions is a non-basic state



(b) two applications of **XOR SRC**: $M^{\perp}_{K,\omega}$.

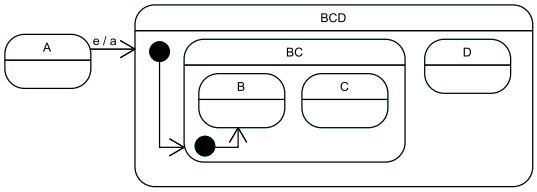
Figure 6.8: Sample application of the **XOR SRC** rule.

6.3.4.3.5.2 Non-Basic Target State

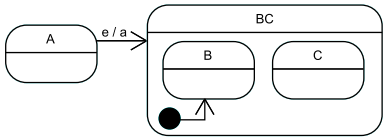
When the target state of the transition in question is a non-basic one, the rule must take history connectors into account: If there are none, the target state is simply the default state. With $s_1, s_2 \in \tilde{S}, e \in \tilde{E}, a \in \tilde{A}, t, t_0 \in \tilde{T}$ and s_1 basic, s_2 non-basic, the following rule defines the translation:

$$\textbf{XOR TGT 1}\ \frac{t = (s_1, e/a, s_2),\ (\nexists h \in \tilde{H} : [\gamma(h) = s_2])}{\xhookrightarrow{\textbf{RULE*}} (t_0 = (s_1, e/a, \delta(s_2)))} \Downarrow \tag{6.20}$$

A simple example for this rule is given by Figure 6.9. The statechart model M_{ESF} as in Figure 6.9(a) contains two levels of hierarchy (BC and BCD), so the rule has to be applied twice. The intermediate statechart that is then considered for a second application is depicted in Figure 6.9(b). A second application then leads to the very basic preliminary Kripke model shown in Figure 6.9(c).



(a) the target of the transitions is a non-basic state, no history connectors



(b) "intermediate step", after one application of the rule **XOR TGT 1**

(c) two applications of **XOR TGT 1**: $M^{\perp}_{K,\omega}$.

Figure 6.9: Sample application of the **XOR TGT 1** rule.

Otherwise, i.e. if there exists a history connector, all direct descendants of the target state must be considered as possible target states. The rules for non-basic target states do not go down the entire hierarchy (**XOR TGT 2**) unless a deep history connector is applied (**XOR TGT 3**). With $s_1, s_2, s_{2_0} \dots s_{2_n} \in \tilde{S}, e \in \tilde{E}, a \in \tilde{A},\ t, t_0 \dots, t_n \in \tilde{T}$ and s_1 basic, s_2 non-basic, the following two rules are defined:

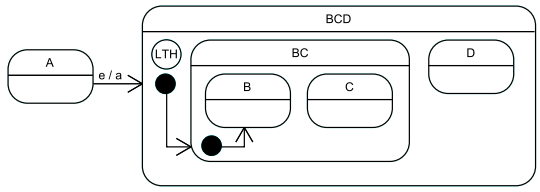
$$\textbf{XOR TGT 2}\ \frac{t = (s_1, e/a, s_2),\ (\exists h \in \tilde{H} : [\gamma(h) = s_2], h\ flat\ \ or\ \ LTH)}{\begin{array}{c} s_{2_0}, \dots, s_{2_n}, \{s_{2_0}, \dots, s_{2_n}\} \subseteq \rho(s_2) : \\ \xhookrightarrow{\textbf{RULE*}} (t_0 = (s_1, e/a, s_{2_0})), \dots, \xhookrightarrow{\textbf{RULE*}} (t_n = (s_1, e/a, s_{2_n})) \end{array}} \Downarrow \quad (6.21)$$

$$\textbf{XOR TGT 3}\ \frac{t = (s_1, e/a, s_2),\ (\exists h \in \tilde{H} : [\gamma(h) = s_2], h\ deep)}{\begin{array}{c} s_{2_0}, \dots, s_{2_n}, \{s_{2_0}, \dots, s_{2_n}\} \subseteq \rho^{+}(s_2) : \\ \xhookrightarrow{\textbf{RULE*}} (t_0 = (s_1, e/a, s_{2_0})), \dots, \xhookrightarrow{\textbf{RULE*}} (t_n = (s_1, e/a, s_{2_n})) \end{array}} \Downarrow \quad (6.22)$$

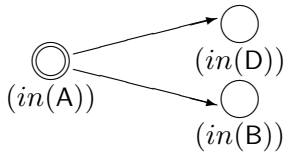
Figures 6.10(b) and 6.10(d) show the difference if a flat (as in Figure 6.10(a)) or deep (as in Figure 6.10(c)) history symbol is contained in the non-basic target state. The rules are nearly identical except for the use of either the direct descendants $\rho(s_2)$ or the whole hierarchy downwards $\rho^{+}(s_2)$.

6.3.4.3.5.3 Non-Basic Source and Target States

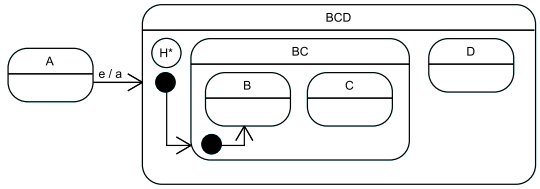
The remaining case in which both source and target of a transition are non-basic states is the simple functional concatenation of the previously defined rules, which is defined with $s_1, s_2 \in \tilde{S}$ non-basic, $t \in \tilde{T}, e \in \tilde{E}, a \in \tilde{A}$ as:



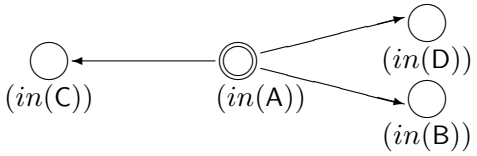
(a) the target of the transitions is a non-basic state, a long term history connector exists.



(b) rule chain: **XOR TGT 2**, **XOR TGT 1** and finally **TRANS 1**



(c) the target of the transitions is a non-basic state, a deep history connector exists.



(d) rule chain: **XOR TGT 3** and finally **TRANS 1**

Figure 6.10: Sample application of the rules **XOR TGT 2** and **XOR TGT 3**.

$$\textbf{XOR SRC TGT}\ \frac{t = (s_1, e/a, s_2)}{\xrightarrow{\textbf{XOR TGT 1,2,3}} (\xrightarrow{\textbf{XOR SRC}} (t))}\ \Downarrow \tag{6.23}$$

Note that the presented form of functional concatenation is a simplification. It represents the following procedure: First, apply the rule **XOR SRC** on all transitions assuming all targets are basic. In the resulting Kripke structure, replace all states – and the related transitions – with $in(\mathsf{x})$ (x being a non-basic target state) with the results of the application of **XOR TGT 1,2,3** on the respective transitions $(s, e/a, x)$ from the *ESF* model (assuming again s basic). All propositions that already emerged from **XOR SRC** are taken over to the new replacement states.

6.3.4.3.6 Composition Rule (AND Rule)

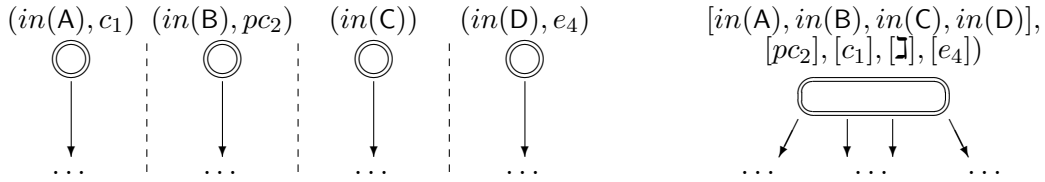
This rule is applied in the second step of the transformation algorithm. It combines the preliminary Kripke models $M^{\perp}_{K,\omega}$ to form the final Kripke model M_K. Basically, a product automaton of the substructures is created. The only deviation from a conventional product automaton is the evaluation of the events and actions. This additional step finally ensures that broadcast communication between orthogonal components is part of the Kripke models. This simple but very efficient step is only possible when CTS is used throughout the models.

The first rule to apply is the **UNITE** rule. It creates intermediary Kripke states $s'' \in S_{K,\omega}$.

$$\textbf{UNITE}\ \frac{([in(x_0)], [p_0^{imp}][p_0^{exp}][p_0^{evt}][p_0^{act}]), \ldots, ([in(x_{\omega-1})], [p_{\omega-1}^{imp}], [p_{\omega-1}^{exp}], [p_{\omega-1}^{evt}], [p_{\omega-1}^{act}])}{([in(x_0), \ldots, in(x_{\omega-1})], [p_0^{imp} \cup p_{\omega-1}^{imp}], [p_0^{exp} \cup p_{\omega-1}^{exp}], [p_0^{evt} \cup p_{\omega-1}^{evt}], [p_0^{act} \cup p_{\omega-1}^{act}])}\ \Downarrow \tag{6.24}$$

It is to be applied to all initial states of the ω preliminary models. A preliminary state $s' = ([in(s)], \ldots)$ is initial iff the statechart state s is a default state of the concurrent component, i.e. $s = \delta(r_\omega)$.

Figure 6.11(a) shows a model extract with $\omega = 4$ (as discussed in Section 4.1.2) parallel initial preliminary Kripke states. The combination by means of the **UNITE** rule is depicted in Figure 6.11(b).



(a) four parallel preliminary Kripke states.

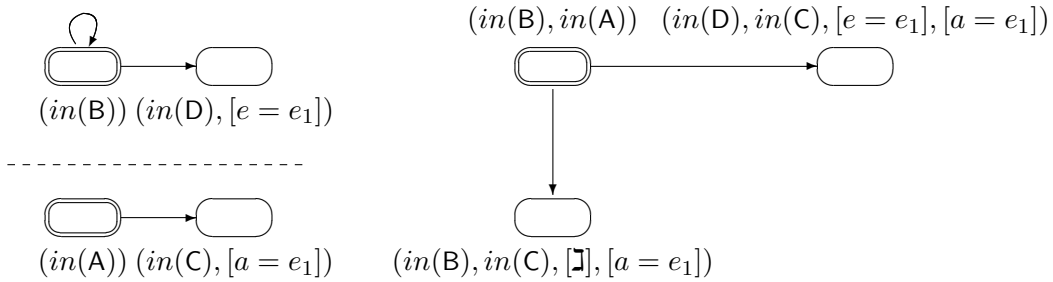
(b) application of **UNITE**: $M_{K,\omega}$

Figure 6.11: Sample application of the **UNITE** rule.

For each existing intermediary Kripke state, the state relation (the arrows in Figure 6.11) is processed by the following rule:

$$\textbf{TRAVERSE}\ \frac{([in(x_0), \ldots, in(x_{\omega-1})], \ldots)\ ,\ ([in(x_0)], \ldots), \ldots, ([in(x_{\omega-1})], \ldots)}{\begin{array}{c}(s'_0, \ldots, s'_{\omega-1}) \in \\ \{s'_0 | (([in(s_0)], \ldots), s'_0) \in R^{\perp}_{K,\omega}\} \\ \times \cdots \times \\ \{s'_{\omega-1} | (([in(s_{\omega-1})], \ldots), s'_{\omega-1}) \in R^{\perp}_{K,\omega}\} : \\ (([in(x_0), \ldots, in(x_{\omega-1})], \ldots), \xhookrightarrow{\textbf{UNITE}} (s'_0, \ldots, s'_{\omega-1}))) \in R_K\end{array}} \Downarrow \quad (6.25)$$

This is the most complex rule of the entire rule set. It describes the creation of a cartesian product of the ω preliminary models. Possible permutations of the preliminary Kripke states that are in relation to the intermediate state in question are constructed by application of the **UNITE** rule. Figure 6.12(a) shows a minimal example with two parallel preliminary models consisting of only two states each. The resulting intermediate states and the relation between them are shown in Figure 6.13(b).



(a) two parallel preliminary Kripke models.

(b) application of **TRAVERSE**: $M_{K,\omega}$

Figure 6.12: Sample application of the **TRAVERSE** rule.

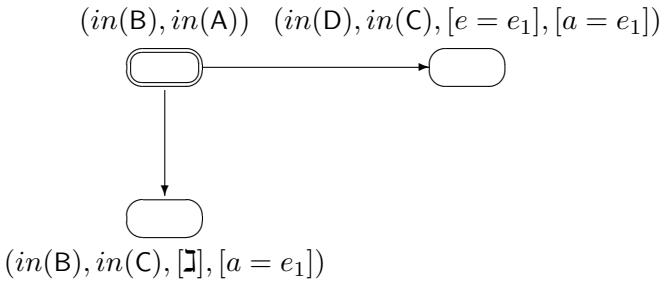
As the intermediate states still contain some unevaluated information, i.e. P_{evt} and P_{act}, they are depicted as bigger ovals in the figures. The last rules finally processes this

information in order to realize broadcast communication between the orthogonal statechart components.

$$\textbf{REDUCE 1}\ \frac{\begin{array}{c}(r'', s_0''), \ldots, (r'', s_n) \in R_K, \\ s_x'' = (\underbrace{[\ldots]}_{\in P_{conf}}, \underbrace{[\ldots]}_{\in P_{imp}}, \underbrace{[\ldots]}_{\in P_{exp}}, \underbrace{[\ldots]}_{\in P_{evt}}, \underbrace{[a]}_{\in P_{act}}),\ s_y'' = (\underbrace{[\ldots]}_{\in P_{conf}}, \underbrace{[\ldots]}_{\in P_{imp}}, \underbrace{[\ldots]}_{\in P_{exp}}, \underbrace{[a]}_{\in P_{evt}}, \underbrace{[\ldots]}_{\in P_{act}})\end{array}}{(r'', s_y''/P_{evt}, P_{act}) \in R_K} \Downarrow \tag{6.26}$$

$$\textbf{REDUCE 2}\ \frac{\begin{array}{c}(r'', s_0''), \ldots, (r'', s_n) \in R_K, \\ s_x'' = (\underbrace{[\ldots]}_{\in P_{conf}}, \underbrace{[\ldots]}_{\in P_{imp}}, \underbrace{[\ldots]}_{\in P_{exp}}, \underbrace{[\ldots]}_{\in P_{evt}}, \underbrace{[a]}_{\in P_{act}}),\ s_y'' \neq (\underbrace{[\ldots]}_{\in P_{conf}}, \underbrace{[\ldots]}_{\in P_{imp}}, \underbrace{[\ldots]}_{\in P_{exp}}, \underbrace{[a]}_{\in P_{evt}}, \underbrace{[\ldots]}_{\in P_{act}})\end{array}}{(r'', s_0''/P_{evt}, P_{act}), \ldots (r'', s_n''/P_{evt}, P_{act}) \in R_K} \Downarrow \tag{6.27}$$

While **REDUCE 1** applies if broadcast communication needs to be considered, **REDUCE 2** simply strips out the unnecessary information. Figures 6.13(a) and 6.13(b) depict the final minimization.



(a) an intermediate structure

(b) application of **REDUCE 1**: M_K

Figure 6.13: Sample application of the **REDUCE** rule.

The presented transformation rule set only allows for the translation of *ESF* models that follow the given guidelines. It is not applicable to every kind of statechart model. In a nutshell: *The translation is a use case tailored one as is the formalism itself.*

6.3.5 Temporal Logic Representation of Indicators

The two indicators to be verified by means of model checking are the lost interrupts and the infinite handling of interrupts as shown in Table 6.1.

The translations of the indicators in temporal logic are quite simple and straightforward: as the Kripke structures as such are created using the *ESF* semantics, the unwinding directly leads to valid paths that can be investigated.

6.3.5.1 Infinite Handling of Interruptions

To make this indicator suitable for model checking, the following question is to be formulated in temporal logic: Is there any case where the interrupt handling is never finished? Or simply: can it occur that once interrupt handling is started, the system will never go back to the abandoned user-space process?

Let U be the set of basic states within user-process space, e.g. in the Linux model (see Figure 4.14) all basic states underneath superstate USER_LEVEL. The set K of basic states is the set of the first basic states that can be entered when entering the IKCP; e.g. in the Linux model: $K = \{$ EXC.CPU_HANDLING, INT.CPU_HANDLING, SYSENTER $\}$.

The following formulae show that for any computation possible ($\mathbf{G}(\dots)$) a started IKCP ($in(k)$) always ends in a basic state located in user-process space ($in(u)$) at some point in the future ($\mathbf{F}(\dots)$):

$$\mathbf{G}(in(k) \rightarrow \mathbf{F}\big(in(u)\big)) \qquad \text{where } k \in K, u \in U \tag{6.28}$$

$$\Leftrightarrow \mathbf{G}(\bigwedge_{k \in K} in(k) \rightarrow \mathbf{F}(\bigvee_{u \in U} in(u))) \tag{6.29}$$

Thus, it can be checked whether *all* paths traversing *any* of the possible "starting points" of an IKCP finally reach user-process space – indicating that the interrupt handling has at last terminated.

As already discussed in Section 5.4.8, transient interrupts can lead to infinite handling regardless of the operating system. This is an illegal case which is to be excluded from the model checking results as discussed in Section 5.4.9. Without a fairness constraint, the model checking results would always contain the cases where transient interrupts lead to infinite handling. As there is an infinite number of these situations, a model checker cannot always determine whether there is a situation aside from these cases that leads to infinite handling. In order to prevent this situation, a fairness constraint excluding these cases that are a priori known is to be defined. According to Definition 6.8, this is equivalent to defining, the set F_K of acceptance conditions that have to hold infinitely often along a path.

$$F_K = \{P_1, \dots, P_n\} = \{pc_1, \dots, pc_n\} \tag{6.30}$$

This constraint is based on the following idea: If all possible interruptions have to occur infinitely often, their ratio must be finite. That is: if only some interruptions occur infinitely often, i.e. if they alternate in any way, the other interruptions cannot occur infinitely often as well and the fairness constraint is violated. The only remaining case that all n different interruptions alternate does not lead to infinite handling due to the constraints within the hardware architecture.

The different interruptions are represented by the path conditions that are set when the corresponding path event occurs and that are at least part of the Kripke states representing the targets of such *ESF* transitions.

6.3.5.2 Losing Interrupts

Besides infinite handling, there are two error sources causing the loss of interrupts:

1. An overflow occurs in the buffer storing interrupt requests while interrupt handling is underway. This topic will be discussed in Chapter 7.
2. The handling of a specific interrupt is initiated, but the dedicated handler routine is never reached. The basic state representing this routine is identified by the handler function $\mathfrak{hdl}^b$ as given by Definition 6.1.

The second possibility can also be checked using an LTL formula:

$$\bigwedge_{e \in \overrightarrow{E}} \mathbf{G}\Big(pc_e \rightarrow \mathbf{F}\big(in(h_e)\big)\Big) \tag{6.31}$$

with $pc_e = \mathfrak{pc}(e) \in PC$ being the path condition of a specific path event $e \in \overrightarrow{E}$ and $h_e = \mathfrak{hdl}^b(e) \in S$. If this formula is true, the handler routine of any possible path event is always reached.

6.4 Combination of Path Length and Recursion Depth

When the two indicators path length and recursion depth are combined, their expressiveness also allows for further evaluation. Two additional statements about the overall number of handling steps kl as discussed in Sections 5.4.8 and 6.2.3 can be made. A fairness constraint as discussed in Section 6.3.5.1 is necessary for any statements about this combined indicator as well.

1. For a given maximal number of system steps, the set of all possible scenarios can be divided into the two subsets of scenarios: the ones that can and those that cannot be completed within that number of steps.
2. For a dedicated set of scenarios, it is possible to determine the maximal number of steps necessary to process these scenarios.

To investigate this combined indicator, a dynamic analysis has to be done. However, the model checking approach as used in Section 6.3 cannot be used without adjustment: it has to be possible to formulate temporal logic formulae including numerical values (e.g. "can a scenario be completed in x steps?") as well as to derive a numeric value as a result (e.g. "what is the maximal number of steps for scenario y?"). To achieve this, temporal logic calculi other than LTL or CTL* are necessary.

1. Koyman proposed a metric temporal logic (MTL) [Koy90] based on linear temporal logic equivalents enriched by a $<$ relation for specifying distances in paths. With such a calculus, one can for example specify a distance that is not to be exceeded.

2. Alur later enhanced this idea by defining parametric temporal logic PLTL [AETP01]. Parametric temporal logic not only allows for checking whether a certain condition is fulfilled within a given number of steps, it also allows for the following kinds of statements:

   (a) in at most x steps f occurs - $\mathbf{F}_{\leq x} f$

   (b) for at least y steps g holds - $\mathbf{G}_{\leq y} g$

These two calculi are well suited for the two possible kinds of statements that deal with the combined indicator. For all paths starting in a state s with $\exists k \in K : [M_K, s \models in(k)]$ and a numerical value x being the maximal number of steps, the following MTL formula determines if a scenario can be processed within this number of steps:

$$\bigvee_{u \in U} \left[\mathbf{F}_{<x+1}(in(u)) \right]_{MTL} \tag{6.32}$$

The sets K and U are used as defined in Section 6.3.5.1. With the defined fairness constraint (Formula 6.30) still applying, a very similar PLTL formula can be used to compute the maximal necessary number of steps. It uses a slightly modified Kripke model M_K which is constructed using only those path events which correspond with the scenarios that are to be investigated.

$$\bigvee_{u \in U} \left[\mathbf{F}_{\leq x}(in(u)) \right]_{PLTL} \tag{6.33}$$

Both statements directly require the notion of scenarios which was avoided for the other indicators. Chapter 7 deals with this scenario dependence.

Chapter 7

Interpretation of Real-Time Capabilities

In the previous chapter, different methods of evaluating the distinct indicators were given. Now, the actual analysis and interpretation of the systems' real-time capabilities is performed.
For the entire analysis, the code is presumed to contain no malicious parts. Since all operating system kernels run in the privileged mode of the CPU, any malicious driver software or module can cause harm to the overall system and irrational behavior. Taking all possible malevolent code portions into account would impede the analysis and considerably taint the results.
All results are interpreted with respect to the use case: soft real-time for multimedia applications. Thus, the evaluation of real-time capabilities based on the IHF indicators refers to a MOSRTOS with a CE handling multimedia scenarios (cp. Section 5.2).

7.1 Analysis of Architectural Properties

Evaluation of the soft real-time capabilities a system implements cannot be conducted without taking into account the kind of real-time application (as pointed out in Chapter 5). Above all, a hard real-time system for a control unit requires the sustaining of interrupt priorities. A multimedia application demands IHF reactivity and flexibility to facilitate overall timeliness; soft real-time requirements of another kind of application might be totally different. As a result, it is pointless to pose the question "how soft real-time capable is a SuI?" – soft real-time capabilities have to be judged with respect to a certain kind of application and its requirements.

The statements derived in this thesis apply to soft real-time requirements of multimedia applications. To achieve the IHF reactivity and flexibility needed in a MOSRTOS, deferred handling of interrupts is to be favored as it keeps the execution of non-interruptible, privileged kernel sections flexible.
Table 7.1 lists the immediate and deferred handling parts in the SuIs as derived from the axioms defined in Section 6.1.1. All possible intermission kernel control paths for the three SuIs are listed in appendix A.1. The three SuIs implement a hybrid approach with

immediate as well as deferred handling portions.

System	Immediate Handling	Deferred Handling	Classification
Linux	IPIs, Global Timer	Local Timer, NIC Interrupt	hybrid
	all other I/O Interrupts		
OpenBSD	IPIs, Timer, I/O Interrupts	NIC Interrupt	hybrid
Pistachio	Timer, IPI	All other Interrupts	hybrid

Table 7.1: Evaluation immediate / deferred handling

However, the weighting differs: OpenBSD handles the vast majority of interrupts immediately while in Pistachio, deferred handling is implemented for nearly all interrupts. The cross-CPU mailbox and the timer interrupt are the only ones to be completely treated as they occur.
Linux conceptually strikes a balance. The portion of interrupts with fixed assignment to immediate or deferred handling is very small; all other routines can be freely implemented in either way. For deferred handling, the concept of tasklets is to be used.
One fact that all investigated systems (and many more like OpenSolaris) have in common is the immediate handling of the timer interruption[1]. By this, a maximal accuracy of the timing facility and an optimal precision of the scheduler can be gained. The timer interrupt is the only one that is designed to be priority-compliant. In general, priority compliance is not a design objective in GPOS (see Section 5.4.4).
Figure 7.1 shows the level of deferral for the three systems. In contrast to an idealized IHF (see Section 5.4.1), the positioning of the systems on the slider is static.

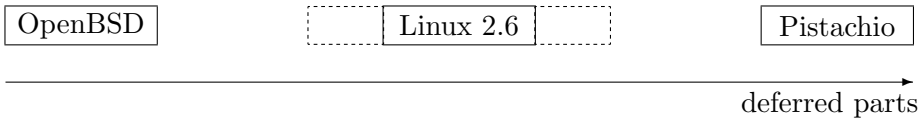


Figure 7.1: Amount of deferred interrupt handling

When deferred handling is used in a system, it must still be determined how the workload is fragmented into the immediate and deferred parts. Pistachio sources out most of the workload to the deferred handlers. Only the hardware acknowledgements and the specification of the handler is carried out immediately. In contrast, OpenBSD and Linux generally immediately perform a considerable amount of work (though in Linux, this also depends on the concrete implementation of interrupt service routines).
Comparing the three SuI IHFs with an idealized IHF with on-demand slider position, it becomes clear that a system which focusses on deferred handling and minimal immediate portions is closest to the idealized situation: when the workload on the system is high, a large part of the workload is postponed. When the system is idle, the immediate and deferred part of handling an interrupt is directly carried out converging on the completely immediate handling in the idealized IHF. Thus, Pistachio with its high level of deferral offers the best basis for a highly reactive system with respect to both load situations.

[1] at least as far as the determination of the current time and the (lazy) invocation of the scheduling routine are concerned

Linux has evolved in the direction of having more deferred handling to gain the flexibility and reactivity of μ-kernel systems such as Pistachio. This fact is highly appreciated with regard to the applicability of Linux for a MOSRTOS system.

Deferred handlers also differ in "how far" they are postponed: are they subject to scheduling or are they always executed prior to the transition back to user space? Table 7.2 shows the evaluation results of this indicator according to appendix A.2.

System	Prior to Scheduling	Subject to Scheduling
Linux	first 10 current tasklets	all other
OpenBSD	NIC interrupt	none
Pistachio	none	all

Table 7.2: Evaluation of deferred handlers: prior or subject to scheduling

In Linux, the deferral to scheduling depends on the number of pending tasklets to be treated. This number does not only comprise deferred handler routines but also other tasklets, e.g. created by software timers. When scheduling is applied, i.e. for the handlers exceeding the limit of 10, a common per-CPU activity is used for all those handlers (the kernel thread `ksoftirqd/<cpu>`). In OpenBSD, the only interrupt with deferred handler is always invoked directly when returning from the immediate interrupt handling while in Pistachio, all deferred handlers are scheduled.
As the CE in a MOSRTOS is designed to schedule activities in user mode, it can be used to take over the scheduling of the deferred handlers if they are not prior to scheduling. Due to the paradigm of decoupling control and the fact that the CE possesses more accurate information on the current requirements of the multimedia application, controlling the deferred handlers is the most beneficial way to achieve soft real-time.
To gain this, task priorities have to be set accordingly: the priorities of the IRQ handlers have to be decreased while the CE scheduler gets the highest priority. Thus, the CE scheduler can use its sophisticated heuristics to prioritize the deferred handlers according to the current application scenario.
From this it also follows that it is not useful to implement a MOSRTOS based on a hard real-time OS: in such an OS, deferred handling is unwanted so that CE scheduling cannot be applied on any deferred handler parts. The requirements of reactivity (MOSRTOS) and priority compliance / immediate handling (hard real-time OS) are inconsistent with each other.
In OpenBSD, the deferred handling is always prior to scheduling so the CE scheduling cannot be exploited here. As a consequence, the CE concept will completely lack the important feature of influencing deferred interrupt handling when used with OpenBSD.
In Linux, it cannot be known beforehand whether a certain deferred handler will be subject to scheduling. But even if a handler is, it will be treated by the common kernel thread `ksoftirqd/<cpu>`, i.e. the CE cannot influence the execution order of handler routines. This would only be possible if the tasklet mechanism was adjusted and a per-handler kernel thread was created. Such a modification would be a severe intervention in the fine-tuned balancing of SoftIRQs.

Only Pistachio allows for exploitation of the CE mechanism without major adjustments in the OS code. A separate module can be implemented to realize the priority changes for handlers and CE scheduler. With this, the CE handler scheduling as described above can be thoroughly used. The custom scheduler concept developed by Wieder [Wie07] could be easily adapted for this purpose.

System	Creation of deferred handling routines
Linux	creation of local kernel daemon `ksoftirqd/<cpu>` outside the model (boot time)
OpenBSD	no handling routines created at all
Pistachio	creation of handler thread contexts outside the model (boot time)

Table 7.3: Evaluation of deferred handlers: creation

In Linux and Pistachio, the handler contexts are created at boot time as indicated in Table 7.3. In contrast to minimal or embedded systems, in GPOS there is no resource shortage. Therefore it is reasonable to create all contexts at boot time so that they are available without delay at any time. There is even no need for on-demand creation since a static system setup is assumed, i.e. there are only predefined, static interrupt sources.

All in all, from the architectural point of view, the μ-kernel system Pistachio which provides the most flexible interrupt handling is best suited for implementing a MOSRTOS.

7.2 Analysis of Determinism and Response Behavior

Reactivity as a major property of a MOSRTOS is not only influenced by architectural properties, it also depends significantly on the intermission paths: not only the length of a path is of interest, but also where it is synchronized and thus might have to wait, where and how it can be interrupted and how long interrupt handling as such can last when multiple interruptions occur. These properties are covered by the strongly connected indicators path length, interruptibility and recursion depth.

The minimal path lengths, i.e. the baseline effort that must be spent at a minimum when handling an interruption, are achieved when immediate handlers are executed without being delayed or disrupted. Here, Linux and OpenBSD are very much alike as shown in Table 7.4 condensing the data from appendix A.3.
However, the more relevant figure is the worst case path length – which differs considerably between Linux and OpenBSD. The inflexible handling of software interrupts in OpenBSD results in large discrepancies between best and worst case: the handling of software interrupts is directly coupled to the handling of system priorities so that multiple calls of soft interrupt handling are likely to occur. Moreover, the coarse-grained synchronization primitives exacerbate the situation: a global kernel lock must be requested every time synchronization is needed. This results in numerous cycles and synchronization points (i.e. red edges in the PAG).

System	Inter Proc. Interrupts	I/O Device Interrupts	Timer Interrupts
Linux	Min: (8,0,0,0) Max: (11,1,2,0)	Min: (11,0,0,0) Max: (14,1,2,0)	Min: (15,0,0,0) Max: (18,1,2,0)
OpenBSD	Min: (4,0,0,0) Max: (10,4,11,1)	Min: (12,0,0,0) Max: (27,14,37,5)	Min: (13,0,0,0) Max: (34,10,26,4)
Pistachio	Min: (2,0,0,0) Max: (7,0,0,0)	Min: (3,0,0,0) Max: (8,0,0,0)	Min: (3,0,0,0) Max: (7,0,0,0)

Table 7.4: evaluation of interruption path lengths, notation: (numerical path length, cycles, acc. cycle length, red edges)

Although in Linux, additional cycles and prolonged paths are present in the worst case figures as well, the influence of synchronization, a severe drawback for real-time, is minimal. While the big kernel lock was still used in the early kernel versions that were SMP-capable, sophisticated fine-grained synchronization primitives such as per-CPU variables (cp. Section 4.3.2) were implemented during the development of the Linux 2.x kernel. Thus, quantities of cycles and synchronization points were significantly reduced.

In Pistachio, all paths[2] are considerably shorter than in Linux and OpenBSD. Moreover, there are no cycles or kernel synchronization points at all. Thus, the handling of an interrupt is completed in a minimum of system steps compared with the other SuIs. Best and worst case figures do not even differ much so that Pistachio also turns out to be the most load-tolerant system investigated.

In general, short path lengths contribute to the reactivity of a system. Yet, the interruptibility of such paths influences reactivity as well: a system that is highly interruptible allows the direct handling of the interrupting event on the one hand but on the other delays the finalization of the interrupted handler – and vice versa. Therefore, a relation between the path length and the interruptibility must be established to make a more sophisticated judgement.

System	Minimal Interruptibility	Maximal Interruptibility	Modal Value
Linux	Timer Interrupts: 0	Deferred I/O Interrupts: 3	1
OpenBSD	Inter Proc. Interrupts: 0	Timer Interrupts: 11	6
Pistachio	Inter Proc. Interrupts: 0	Deferred I/O Interrupts: 6	6

Table 7.5: Evaluation of interruptibility

The figures in Table 7.5 summarize maximal, minimal and modal number of interruption points as derived from the full interruptibility matrices in appendix A.4. The latter figure is the number of interruption points with the most occurrences in all paths.
Reactivity is constructed from two distinct facets: To judge how fast the handling of a new interrupt can be started, the ratio of path length and interruption points is evaluated – the expected number of system steps before the handling can be kicked off depends on

[2]Note that the Pistachio figures accumulate kernel and user space paths.

the interruptible and non-interruptible portions of a path. The second facet is the overall delay in handling, i.e. how long it takes to finalize interrupt handling once it is started. Here, the absolute number of interruption points matters, as well as the recursion depth as discussed later.
As a matter of fact, the two facets conflict: for the first one, higher interruptibility is preferable, for the second, interruptions prolong the completion of the handling and are thus undesirable. In general, short path lengths mitigate this tradeoff as they allow the swift resumption of an interrupted path. Linux addresses this tradeoff by giving the decision of interruptibility (voluntary kernel preemption) to the user at compile time.

In Linux, the absolute number of interruption points is low for most paths including immediate handlers. This is so mainly because the lowest level of preemption was assumed for this model. In OpenBSD, paths can be interrupted more often, but are also expected to be longer so that long non-interruptible sequences can occur. Moreover, the fact that the high path lengths cause an interrupted execution to wait a long time to be resumed is assumed to cause a much higher delay than having to wait for an interruption slot to start execution. Therefore, the reactivity of OpenBSD is identified to be weaker than that of Linux with its fewer interruption points but also shorter paths.
Pistachio unites both: nearly all states can be interrupted so that treatment of an interrupt is started quickly. At the same time, all paths are sufficiently short so that every interrupted path can be resumed with short delay. This is the optimal combination of properties.
The overall delay depends on the actual nesting of interruptions: any interruptible path currently executed can be left to process a more recent interrupt leading to a recursive descent. The maximal depth of such a recursion depends on the concrete system implementation. Table 7.6 lists the maximal recursion depths without the exceptions.

System	Maximal Recursion Depth	
Linux	2 timer + 3 IPI + 6 I/O devices	= 11
OpenBSD	1 syscall + 1 IPI + 1 timer + 6 I/O devices	= 9
Pistachio	1 IPI + 1 timer + 6 I/O devices	= 8

Table 7.6: Maximal recursion depths

The different values represent different implementation strategies. In Linux, each IPI comes with its own interrupt vector which simplifies treatment of mutual interruptions. This is possible as the number of IPIs is small. In OpenBSD, there is only one IPI vector with branching towards the respective handler routine. The mutual interruptibility is software-based and thus costly, but the best solution for the large number of similar IPIs, e.g. to alleviate the initialization of interrupt vectors.

The overall delay of an interrupted path depends on the particular path lengths and the actual level of recursion. While the maximum recursion level as given above depends statically on a system's implementation, scenarios and the actual overall paths resulting from them have to be taken into account to derive further statements on the real-time behavior of a system.

The main idea in a CE system is to exploit information about the topology and thus the structure of multimedia scenarios. If a mapping of these known application scenarios to interrupt scenarios existed, it would even be possible to derive the actual recursion depth – and from that the overall path length – for each scenario beforehand. The maximal recursion depth is the upper boundary for this value.
Since application scenario scheduling and buffer planning are mostly static in a CE system, a one-to-one mapping between application scenarios and interrupt load scenarios as defined in Definition 4.1 can be established as discussed in Section 8.2. A concrete CE specification is needed for this.

The response behavior of a system however can be evaluated without taking these scenarios into account. The figures derived clearly show that Pistachio combines the most favorable properties once more, followed by Linux.

7.3 Analysis of Reliability

To model check the reliability indicators, infinite handling and losing interrupts, the models have to be converted into a format readable by an established model checking tool. This translation is to be done based on the transformation rule set presented in Section 6.3.4.1. Two approaches can be taken:

1. Implementation of a parser exploiting the rule set to automatically generate suitable code from the *ESF* model

2. Manual on-the-fly application of the rule set to the *ESF* models to create code

For the first approach, it would be necessary to have a specific format for *ESF* models. Today, there is no editor capable of handling *ESF* and of creating textual model files (e.g. based on XMI). The specification and implementation of this would be a separate work (see Section 8.2) that could then serve as a basis for the development of a parser.
For now, manual on-the-fly creation of the models is done. As the models are not too large, this can be done quickly and efficiently. The target of the translation is the input language for the model checker NuSMV (see Section 6.3.2) which performs symbolic model checking. For conventional on-the-fly model checking, the *ESF* models are too complex. In the following, the translation of the *ESF* models into a NuSMV-compliant format will be presented.

For the two indicators under investigation, parallelism within a system is not relevant: inter-CPU synchronization is taken into account as the model checker anyway unwinds all possible situations including values of conditions. This allows concentrating on one CPU. In case there is a prolonged path on a distinguished CPU[3], this one is taken.

Since malicious kernel code is disregarded, the only exception that could occur in kernel mode is the page fault. However, as kernel parts are never swapped, this very exception cannot be raised in kernel mode so it can be excluded from further considerations as

[3]In the case of the timer interrupt, some housekeeping tasks are only executed on the primary processor.

well. Therefore, the exception submachine is not translated into the SMV models. Only the OpenBSD SMV model embodies the system call handling since system calls are not implemented by means of software generated interrupts in Linux and Pistachio. Due to the fact that the according interrupt vector for system calls in OpenBSD is 128, its interrupt priority ensures that it can be disrupted by any other interrupt but not dismiss any of the regarded IKCPs itself.
The set P_{conf} is thus the set of basic states within one component, i.e. one CPU submodel. This set of basic states is defined in the section `VAR` of the NuSMV file. This part contains all variable declarations of any kind. The specification of the states and any other enumeration type as the two other subsets P_{imp} and P_{exp} are declared in this portion of the code as given in the following SMV pseudo-code example:

```
MODULE main

VAR
  -- P_{CONF}
  state : {<in_STATE_1>, <in_STATE_2>, ..., <in_STATE_x>};

  -- P_{IMP}
  pc    : {<pc_1>, <pc_2>, ... , <pc_y>};

  --P_{EXP}
  cond_1 : boolean;
  cond_2 : boolean;
```

The relations Δ_{conf}, Δ_{imp} and Δ_{exp} are then implemented as described in Section 6.3.4.1 based on the step relation (`next`) of the SMV model checker.

```
ASSIGN
  -- Delta_{P_{CONF}}
  init(state) := in_STATE_x;
  next(state) := case
                  <cond> : in_STATE_x;
                  ...
                 esac;

  -- Delta_{P_{IMP}}
  init(pc)   := pc_NONE;
  next(pc)   := case
                  <cond> : pc_y;
                  ...
                 esac;

  -- Delta_{P_{EXP}}
  init(cond_1):= 0;
  next(pc)   := case
                  <cond> : 1;
                  ...
                 esac;
```

In SMV, all atomic propositions are defined separately, i.e. without any relation to other atomic propositions. However, the APs defined for the Kripke representation of *ESF* are connected: they all rely on the step relation between system configurations. Thus, it is necessary to synchronize the `next` relations of P_{imp} and P_{exp} with the `next` relation for states which represents the actual *ESF* step relation.
This synchronization is achieved by sophisticated guarding, i.e. similar conditions for stepping further in the SMV model for all three sets of propositions. The following code shows an example of where the next basic state and the next path condition are set analogously, both depending on the current state and path event.

```
ASSIGN
  -- Delta_{P_{CONF}}
  init(state) := in_STATE_x;
  next(state) := case
                   (state = in_STATE_x & pe = pe_y) : in_STATE_y;
                   (state = in_STATE_x & pe = pe_z) : in_STATE_z;
                  esac;

  -- Delta_{P_{IMP}}
  init(pc)   := pc_NONE;
  next(pc)   := case
                   (state = in_STATE_x & pe = pe_y) : pc_Y;
                   (state = in_STATE_x & pe = pe_z) : pc_Z;
                  esac;
```

Without this parallelism, the model would "fall apart" in terms of generation of the Kripke states, i.e. the states generated and their relations would no longer reflect the chosen statechart semantics. Note that enforcing this synchronous stepping would also be a challenge when implementing a parser for *ESF* models to be translated into SMV.
The fairness constraints are implemented as justice constraints [CCJ+05] using the SMV keyword `JUSTICE`. The example below shows the fairness condition "path event x appears an infinite number of times" for a model with two different interrupts that can occur, i.e. a maximum nesting level of 2. As a path event can only occur again when the previous one of this type was processed (i.e. there is no recursion level with the corresponding path condition pending), the fairness constraint can be rewritten as follows:

```
JUSTICE pcs[0]!=pc_X & pcs[1]!=pc_X;
```

Finally, the indicators are given using the keyword `LTLSPEC`:

```
LTLSPEC
  G(state = in_STATE_START -> F(state = in_STATE_FINAL))
```

Variables that are of no enumeration type were defined as boolean where possible. As NuSMV internally uses boolean representations for all variables, this greatly reduces the complexity of model checking compared with, for example, integer variables that are internally represented bitwise to comply with the boolean format. This simplification applies for example for the counter of pending SoftIRQs: in the model, it is only noted whether there are soft interrupts pending.

Table 7.7 lists the characteristic figures for the three SMV models.

System	BDD Nodes	All States	Reachable States
Linux	2679229	$1.22664 \cdot 10^{13}$	$2.87907 \cdot 10^{6}$
OpenBSD	1035634	$5.0251 \cdot 10^{14}$	625860
Pistachio	1195862	$1.7408 \cdot 10^{11}$	3215

Table 7.7: Model characteristics in NuSMV

In Table 7.8, the results of the analysis are presented.

System	Infinite Handling	Lost Interrupts
Linux	no	none
OpenBSD	possible only for syscalls	possible only for syscalls
Pistachio	not in kernel mode	none

Table 7.8: Model checking results

In Linux, infinite handling is impossible: under the given fairness constraints, any IKCP is terminated in a finite number of steps, no matter where and how often it is interrupted. Furthermore, all interrupts are finally treated in their dedicated handling states.
In OpenBSD, this is not the case: when the first interruption treated was a syscall, the interrupt handling does not necessarily terminate. Due to the low priority of the syscall itself, its handling can always be disrupted by any other interrupt. As the handling of interrupts is always terminated before resuming syscall handling, all interrupts are free to occur again. The following example shows such a case where the IHF alternates between syscall and interrupt handling without terminating.

```
// Example: Infinite Handling in OpenBSD
// handling of syscall pe_SYS
-> state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM
-> state = in_IKCP-SYS-SET_LOCK-LOCK
-> state = in_IKCP-SYS-SET_LOCK-SPLX
-> state = in_IKCP-SYS-DO_SYSCALL-EXEC_SYSCALL -> Input: 1.5 <-
// interrupted by pe_INT_LAPIC
// Loop starts
-> state = in_IKCP-SYS-REM_LOCK-REMOVE
-> state = in_IKCP-INT-CREATE_FRAME
-> state = in_IKCP-INT-LAPIC_TIMER-EOI
-> state = in_IKCP-INT-LAPIC_TIMER-SET_LOCK-LOCK
-> state = in_IKCP-INT-LAPIC_TIMER-SET_LOCK-SPLX
-> state = in_IKCP-INT-LAPIC_TIMER-TSC_ACT
-> state = in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-TIMER
-> state = in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-SCHEDULE
-> state = in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-SETSOFTCLOCK
-> state = in_IKCP-INT-LAPIC_TIMER-REM_LOCK-REMOVE
-> state = in_IKCP-INT-LAPIC_TIMER-REM_LOCK-SPLX
-> state = in_IKCP-INT-XDORETI
// timer handling completed, return to syscall handling
```

```
// Loop: again interrupted by pe_INT_LAPIC
-> state = in_IKCP-SYS-REM_LOCK-REMOVE
```

For all other interrupts, no infinite handling can occur. Moreover, all handler states of interrupts are reached, so that interrupts cannot get lost once their processing is triggered. For syscalls, handling cannot be guaranteed due to the low priority and interruptibility of the syscall as discussed above.
Note that in systems where syscalls are not implemented as software generated interrupts, it is not mandatory to guarantee that the syscall handling is terminated. There, they are treated just as any kernel paths for which the postulated demand for completion does not apply – it only has to be guaranteed that they can be resumed.

In Pistachio, no infinite handling can happen in kernel mode. However, as the deferred handling is executed in user mode, a case similar to the alternation problem in OpenBSD can occur: although the kernel mode IHF always terminates, the user space handler can be interrupted at any time. Thus, the overall handling of the interrupt is not necessarily completed as illustrated by the example below.

```
// Example: Infinite Handling in Pistachio
// handling of pe_IO_0 up to user space handler
-> state = in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK
-> state = in_KERNEL-INTERRUPT-HW_IRQ-EOI
-> state = in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_0
-> state = in_KERNEL-INTERRUPT-HW_IRQ-IRET
-> state = in_USER-HANDLERS-IRQ_THREAD
// handler is interrupted by pe_IO_5
// Loop starts here
-> state = in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK
-> state = in_KERNEL-INTERRUPT-HW_IRQ-EOI
-> state = in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_5
-> state = in_KERNEL-INTERRUPT-HW_IRQ-IRET
// resume handling of pc_IO_0
-> state = in_USER-HANDLERS-IRQ_THREAD
// interrupt pe_IO_5 occurs again, user space handling is left --> loop
-> state = in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK
```

As the dedicated handler states for the interrupts are located in kernel space, losing interrupts is not possible in Pistachio – despite the possibility of infinite handling.

When the cases of switching between kernel mode and user mode in Pistachio and between syscall handling and interrupt handling in OpenBSD are excluded from the model checking indicator, the analysis shows that there are no other occurrences of infinite handling or losing interrupts in the systems.
In OpenBSD, the possibly infinite handling of syscalls is induced by the software architecture: as syscalls have the lowest possible priority, it can only be guaranteed that their handling is resumed, but not that it is completed. This priority was consciously chosen in system design. In the other systems, this infinite handling cannot occur as syscalls are not implemented as interrupts and the other interrupts have no fixed priorities. While OpenBSD is certainly unsuitable for hard real-time, the identified flaw can be handled in a MOSRTOS: as the scenarios, including syscalls, are known to the CE a priori, it is possible

to implement the identification of a possible infinite handling scenario beforehand. Thus, a procedure to avoid infinite handling and lost syscalls can be integrated into the CE.
This can be used to address infinite handling in Pistachio as well: the CE can be implemented to react to such a scenario beforehand to avoid the problem, e.g. by rescheduling or by incorporating a small delay to allow for completing the handling in user space. However, these identified cases are exotic situations unlikely to happen: they demand worst case scenarios with interrupts that occur within such infinitesimal time intervals that it is practically impossible for such a scenario to happen in a multimedia application on a non-malfunctioning system.
Since device interrupts cannot get lost due to the fact that their handling is not conducted, the only remaining possible cause for losing such interrupts is an overflow of the buffering circuit that stores interrupts occurring while the corresponding interrupt line is masked. When enhancing the modeling framework presented with real execution times or proper worst case execution times as discussed in Sections 6.2.1 and 8.2, a direct algebraic relation between scenarios and buffer filling level of the circuit could be defined. Establishing such a relation based on path length is of no use since the external stimuli relate to the real-world time scale.
All in all, the analysis of the data-based indicators shows that there are only some very special cases in Pistachio and OpenBSD that have to be covered by the CE implementation to prevent erroneous behavior, while for Linux, the analysis has shown that erroneous behavior cannot occur at all.

Taking all defined and investigated IHF indicators into account, it is clear that the μ-kernel system Pistachio possesses the most advantageous properties with respect to architectural and control-based indicators. The minimalist design provides the highest flexibility for implementing a MOSRTOS using a CE. Directly comparing the two non-μ-kernel systems, Linux scores better than OpenBSD. This is mainly due to the more modern kernel implementation with clear μ-kernel tendencies.
As the peril of infinite handling can be averted beforehand by the CE, there are no serious flaws concerning the data-based properties of Pistachio. Taking all this into account, Pistachio is highly suitable for the implementation of a MOSRTOS and thus the prime recommendation. Linux with its expedient properties is a reasonable practical alternative to the still experimental Pistachio system.
L4Linux[4] might be a good compromise: this system combines a μ-kernel OS with a full Linux on top of it [HHW98]. This unites the demonstrated advantages of a μ-kernel system with the widely supported and well-established Linux OS.

[4] http://os.inf.tu-dresden.de/L4/LinuxOnL4/

Chapter 8

Conclusion

8.1 Summary

In this thesis, a formal approach was conducted to derive statements on the usability of general purpose operating systems for the implementation of a multimedia-oriented soft real-time OS. The focus of the investigation was on the interrupt handling facility of such systems as this is an important determining factor of a system's soft real-time capabilities.

After specification of the basic conditions such as the hardware setup, formal models for the interrupt handling facilities of three selected GPOS – Linux, OpenBSD and Pistachio – were created. To facilitate the generation of these models, a modeling technique uniting simplicity in modeling, clearness in presentation and formal accuracy was needed. As no existing technique could satisfactorily provide these properties, a new use case tailored modeling method based on statecharts was developed: the engineering statechart formalism *ESF*. With the *ESF* and its modeling paradigms, it becomes possible to model most parts of an operating system in a simple way. Due to the formal foundations, the *ESF* models can serve as a basis for the analysis of the soft real-time indicators investigated in this thesis. Moreover, the *ESF* as such facilitates the application of best practices in software engineering such as model-driven analysis and verification to real-world operating systems and paves the way for future research and application in operating systems design and analysis.

Though the term soft real-time is widely used, it turned out that existing definitions are rather fuzzy. Therefore, a definition of real-time was presented based on provided and perceived values of a task over time.
The perceived values that relate to a complete system was then projected on the interrupt handling facility of a MOSRTOS system that uses a component extension module to handle multimedia streams. This led to the specification of indicators that determine the soft real-time capabilities of an IHF in a MOSRTOS environment.

For the evaluation of the defined indicators, indicator-specific representations of the *ESF* models were devised preserving their formal expressiveness. For the different classes of indicators – architectural, control-based and data-based – different analysis techniques were introduced: static and graph-based analysis as well as temporal logic model checking were

applied. For the latter, a complete rule set to translate all *ESF* constituents into Kripke structures, a universal formal structure suitable for temporal logic model checking, was specified.
The examination of the indicators revealed that Pistachio is most suitable for the implementation of a MOSRTOS. Nonetheless, Linux also incorporates some advantageous properties so that it is a suitable candidate, too.

8.2 Outlook

As this thesis contributed to a variety of research fields, further research options arise in different areas.

8.2.1 Operating Systems Engineering with *ESF*

In order to elaborate the existing *ESF* models of the three SuIs, interrupt service routines for dedicated devices can be modeled. This will turn the handlers into white boxes even facilitating model-based analysis of drivers and other kernel modules. Other parts of the SuIs such as the process management facility can be particularized as well.
As a next step, the *ESF* modeling and analysis techniques can be applied to other operating systems, e.g. hard real-time systems such as QNX. This application aims at different goals: to either answer questions on real-time capabilities as done in this thesis or to pursue new objectives using the *ESF* approach as a framework.

As already discussed, the integration of timing into the models is also an issue. As *ESF* as a modeling method already allows this, the main task is to perform worst case execution time analysis of functions and calls by measuring actual OS behavior. Afterwards, additional indicators evaluating concrete timing can be defined.
Another promising approach is to streamline CE research with the *ESF* concept: an *ESF* model of the CE can be integrated into timed OS models to allow for the analysis of concrete timed scenarios. As an alternative, an approach to stochastically analyze soft real-time behavior with respect to given load profiles and scenarios could be developed.
The advantages of a clear visual representation of OS using *ESF* can be exploited when it comes to education: *ESF* can serve as a vehicle to clearly show operating systems structures and principles. To pursue this goal, traditional operating systems models such as client-server protocols and master/slave relationships can be mapped to *ESF* models. Furthermore, Linux trace visualization [Koh07] – that already supports XMI formats – and *ESF* can be brought together to provide a consistent visual notation for static and dynamic illustrations of OS.

8.2.2 Implementation of *ESF*

A graphical editor to create and modify *ESF* models is necessary to alleviate the modeling process and to allow for electronic interchange of the models. For the latter, an XMI representation of the *ESF* must be defined. One option is to enhance the open-source ArgoUML editor that already supports UML statecharts. The Netbeans XMI writer[1]

[1] http://mdr.netbeans.org/uml2mof/

used by ArgoUML can be modified to support *ESF*.
Taking these steps, a program can be designed and implemented that converts the XMI representation of an *ESF* model into the input format of a model checking tool.

8.3 Conclusion

The research conducted clearly contributes to filling a blank space in operating systems research and engineering. It also narrows the gap between best practises in software engineering and methods for analyzing operating systems. The real-time definition provided as well as the analysis of the soft real-time capabilities of concrete real-world operating systems greatly assist the design and development of a multimedia-oriented soft real-time system.

Bibliography

[AB76] James Wayne Anderson and J. C. Browne. Graph Models of Computer Systems: Application to Performance Evaluation of an Operating System. In *SIGMETRICS '76: Proceedings of the 1976 ACM SIGMETRICS Conference on Computer Performance Modeling Measurement and Evaluation*, pages 166–178, New York, NY, USA, 1976. ACM Press.

[AETP01] Rajeev Alur, Kousha Etessami, Salvatore La Torre, and Doron A. Peled. Parametric Temporal Logic for "Model Measuring". *ACM Trans. Comput. Logic*, 2(3):388–407, 2001.

[AH99] Alan Au and Gernot Heiser. L4 User Manual, 1999.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[AKY99] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating Hierarchical State Machines. In *ICAL '99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, pages 169–178, London, UK, 1999. Springer-Verlag.

[AY01] Rajeev Alur and Mihalis Yannakakis. Model Checking of Hierarchical State Machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, 2001.

[BAMP81] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The Temporal Logic of Branching Time. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 164–176, New York, NY, USA, 1981. ACM.

[Bar98] Michael Barbehenn. A Note on the Complexity of Dijkstra's Algorithm for Graphs with Weighted Vertices. *IEEE Transactions on Computers*, 47(2):263, 1998.

[BBB97] R. Baron, D. Black, and W. Bolosky. MACH Kernel Interface Manual, 1997.

[BBD+99] Tom Bienmüller, Udo Brockmeyer, Werner Damm, Gert Döhmen, Claus Eßmann, Hans-Jürgen Holberg, Hardi Hungar, Bernhard Josko, Rainer Schlör, Gunnar Wittich, Hartmut Wittke, Geoffrey Clements, John Rowlands, and Eric Sefton. Formal Verification of an Avionics Application using Abstraction

and Symbolic Model Checking. In *Proceedings of the Seventh Safety-Critical Systems Symposium*, pages 150–173. Springer-Verlag, 1999.

[BC01] Daniel Pierre Bovet and Marco Cassetti. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1. edition, 2001.

[BC06] Daniel Pierre Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3. edition, 2006.

[BCM+92] J. R. Burch, Edmund M. Clarke, Kenneth Lauchlin McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[BDW00] Tom Bienmüller, Werner Damm, and Hartmut Wittke. The STATEMATE Verification Environment - Making It Real. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 561–567, London, UK, 2000. Springer-Verlag.

[Ber73] Claude Berge. *Graphs and Hypergraphs*, volume 6 of *North - Holland Mathematical Library*. North Holland, Amsterdam, 1. edition, 1973.

[Bev89] W. R. Bevier. KIT: A Study in Operating System Verification. *IEEE Trans. Softw. Eng.*, 15(11):1382–1396, 1989.

[BG92] Gérard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[Bie97] Armin Biere. μcke - Efficient μ-Calculus Model Checking. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 468–471, London, UK, 1997. Springer-Verlag.

[Bli89] Wayne D. Blizard. Multiset Theory. *Notre Dame Journal of Formal Logic*, 30(1):36–66, 1989.

[BR04] Purandar Bhaduri and S. Ramesh. Model Checking of Statechart Models: Survey and Research Directions, July 2004.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[BS94] W. R. Bevier and L. Smith. A Mathematical Model of the Mach Kernel: Kernel Requests. Technical Report 53, Computational Logic Inc., 1994.

[Büc62] J. R. Büchi. On a Decision Method in Restricted Second Order Arithmetic. In E. Nagel, editor, *Proceedings 1960 International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[Büs03] Robert Büssow. *Model Checking Combined Z and Statechart Specifications*. PhD thesis, Technical University Berlin, 2003.

[BW95] A. Burns and A. J. Wellings. Engineering a Hard Real-Time System: From Theory to Practice. *Softw. Pract. Exper.*, 25(7):705–726, 1995.

[BW98a] Udo Brockmeyer and Gunnar Wittich. Real-Time Verification of Statemate Designs. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 537–541, London, UK, 1998. Springer-Verlag.

[BW98b] Udo Brockmeyer and Gunnar Wittich. Tamagotchis Need Not Die - Verification of STATEMENT Design. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 217–231, London, UK, 1998. Springer-Verlag.

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

[CCJ+05] Robert Cavada, Alessandro Cimatti, Charles Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, M. Roveri, and Andrei Tchaltsev. *NuSMV 2.4 User Manual*, 2005.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CES83] Edmund M. Clarke, E. Allen Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent System Using Temporal Logic Specifications: A Practical Approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 117–126, New York, NY, USA, 1983. ACM.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000.

[CH00] Edmund M. Clarke and W. Heinle. Modular translation of Statecharts to SMV. Technical Report CMU-CS-00-XXX, Carnegie Mellon University School of Computer Science, August 2000.

[Che76] Peter Pin-Shan Chen. The Entity-Relationship Model - Towards a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[Cla90] Raymond Keith Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, 1990.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 3. edition, 2005.

[CY74] Richard W. Conn and Richard H. Yamamoto. A Model Highlighting the Security of Operating Systems. In *ACM 74: Proceedings of the 1974 Annual Conference*, pages 174–179, New York, NY, USA, 1974. ACM Press.

[Day93a] Nancy Day. A Model Checker for Statecharts. Technical report, University of British Columbia, Vancouver, Canada, 1993.

[Day93b] Nancy Day. The Semantics of Statecharts in HOL. In *6th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 339–351, Vancouver, BC, August 1993. Springer-Verlag.

[DHP05] Iakov Dalinger, Mark A. Hillebrand, and Wolfgang J. Paul. On the Verification of Memory Management Mechanisms. In Dominique Borrione and Wolfgang J. Paul, editors, *CHARME*, volume 3725 of *Lecture Notes in Computer Science*, pages 301–316. Springer, 2005.

[DW05] Sven Thorsten Dietrich and Daniel Walker. The Evolution of Real-Time Linux. In *Proceeding of the seventh Real-Time Linux Workshop*, Boise, Idaho, 2005. The Real-Time Linux Foundation Inc.

[DWQQ01] Wei Dong, Ji Wang, Xuan Qi, and Zhi-Chang Qi. Model Checking UML Statecharts. In *APSEC '01: Proceedings of the Eighth Asia-Pacific Conference on Software Engineering*, page 363, Washington, DC, USA, 2001. IEEE Computer Society.

[Edw01] James Edwards. Process Algebras for Protocol Validation and Analysis. In *Proceedings of PREP 2001*, pages 1–20, Keele, April 2001. EPSRC.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time Temporal Logic. *J. ACM*, 33(1):151–178, 1986.

[Ehr88] Hartmut Ehrig. *Graph-Grammars and Their Application to Computer Science.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1988.

[Eme81] E. Allen Emerson. *Branching Time Temporal Logic and the Design of Correct Concurrent Programs.* PhD thesis, Harvard University, 1981.

[Erk05] Juha Erkkilä. Real-Time Audio Servers on BSD Unix Derivatives. Master's thesis, University of Jyväskylä, Finland, 2005.

[Eul72] Leonard Euler. *Lettres a une Princess d'Allemagne (Translated to English).* Longman in 1830, 3. edition, 1772.

[Feh93] Rainer Fehling. A Concept of Hierarchical Petri Nets with Building Blocks. In *Papers from the 12th International Conference on Applications and Theory of Petri Nets*, pages 148–168, London, UK, 1993. Springer-Verlag.

[FH01] Kay Fuhrmann and Jan-Juan Hiemer. Formal Verification of STATEMATE-Statecharts. Technical report, ESPRESS Project, 2001.

[Fil97] T. Filkorn. Applications of Formal Verification in Industrial Automation and Telecommunication. In *Proceedings of Workshop on Formal Design of Safety Critical Embedded Systems*, 1997.

[FL01] Thomas Flik and Hans Liebig. *Mikroprozessortechnik (in German)*. Springer, 6. edition, 2001.

[GHJ+77] M. G. Gouda, Y. W. Han, E. Douglas Jensen, W. D. Johnson, and R. Y. Kain. Applications of DDP Technology to BMD: Architectures and Algorithms. *Distributed Data Processing Technology*, 4, 1977.

[GHLP05] Mauro Gargano, Mark A. Hillebrand, Dirk Leinenbach, and Wolfgang J. Paul. On the Correctness of Operating System Kernels. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[GKS94] Sven Graupner, Winfried Kalfa, and Frank Schubert. Multi-Level Architecture of Object-Oriented Operating Systems. Technical Report TR-94-056, International Computer Science Institute, Berkeley, CA, 1994.

[GL91] B. O. Gallmeister and C. Lanier. Early Experience with POSIX 1003.4 and POSIX 1003.4A. In *IEEE Real-Time Systems Symposium*, pages 190–198, 1991.

[GLT80] Hartmann J. Genrich, Kurt Lautenbach, and P. S. Thiagarajan. Elements of General Net Theory. In *Proceedings of the Advanced Course on General Net Theory of Processes and Systems*, pages 21–163, London, UK, 1980. Springer-Verlag.

[Gog07] Robert Gogolok. Reverse-Engineering des Interrupt-Subsystems von OpenBSD (in German). Bachelors thesis, August 2007.

[GP93] Patrice Godefroid and Didier Pirottin. Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract). In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 438–449, London, UK, 1993. Springer-Verlag.

[Han73] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Har88] David Harel. On Visual Formalisms. *Communications of the ACM*, 31(8):514–531, 1988.

[HG88] Cornelis Huizing and Rob Gerth. On the Semantics of Reactive Systems. Technical report, Eindhoven University of Technology, 1988.

[HGdR88] Cornelis Huizing, Rob Gerth, and Willem P. de Roever. Modeling Statecharts Behaviour in a Fully Abstract Way. In *CAAP '88: Proceedings of the 13th Colloquium on Trees in Algebra and Programming*, pages 271–294, London, UK, 1988. Springer-Verlag.

[HHLS97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The Performance of Microkernel-based Systems. In *SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 66–77, New York, NY, USA, 1997. ACM Press.

[HHW98] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux, 1998.

[Hie01] Jan-Juan Hiemer. *Statecharts in CSP - Ein Prozessmodell zur Analyse von Statemate-Statecharts.* PhD thesis, Technical University Berlin, Berlin, Germany, 2001.

[HK92] David Harel and Chaim-arie Kahana. On Statecharts With Overlapping. *ACM Trans. Softw. Eng. Methodol.*, 1(4):399–421, 1992.

[HK94] Johannes Helbig and Peter Kelb. An OBDD-Representation of Statecharts. In *EDAC-ETC-EUROASIC*, pages 142–149, 1994.

[HLN+90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, R. Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *Software Engineering*, 16(4):403–414, 1990.

[HN96] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[HN04] K Hangos and E Nemeth. Multi-Scale Process Model Description by generalized Hierarchical CPN Models. Technical Report SCL-002, Hungarian Academy of Sciences, 2004.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hoh98] Michael Hohmuth. The Fiasco Kernel: Requirements Definition, 1998.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[Hol03] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 1 edition, 2003.

[HP98] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach.* McGraw-Hill, Inc., New York, NY, USA, 1998.

[HPSS87] David Harel, Amir Pnueli, J. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts. pages 54–64, 1987.

[IBDR03] A. Iqbal, A. K. Bhattacharjee, S. D. Dhodapkar, and S. Ramesh. Visual Modeling and Verification of Distributed Reactive Systems. In *SAFECOMP*, pages 22–34, 2003.

[IEE95] IEEE. IEEE Standard Hardware Description Language based on the Verilog Hardware Description Language, 1995.

[IEE01] IEEE. The Open Group Base Specifications Issue 6, IEEE Std 1003.1-2001, 2001.

[Int88] Intel. 8259A - Programmable Interrupt Controller, December 1988.

[Int94] Intel. 82C54 CHMOS Programmable Interval Timer, 1994.

[Int96] Intel. 82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC), 1996.

[Int97] Intel. MultiProcessor Specification Version 1.4, 1997.

[Int04] Intel. IA-PC HPET (High Precision Event Timers) Specification, 2004.

[Int05] Intel. IA-32 Intel Architecture Software Developers Manual - Volume 3: System Programming Guide, 2005.

[Int08a] Intel. INTEL 64 and IA-32 Architectures Software Developers Manual - Volume 1: Basic Architecture, February 2008.

[Int08b] Intel. Intel 64 and IA-32 Architectures Software Developers Manual System Programming Guide, Part 1, 2008.

[ISO89] International Organization for Standardization ISO. Information Processing Systems – Open Systems Interconnection – Estelle – A Formal Description Technique Based on an Extended State Transition Model, 1989.

[ISO91] International Organization for Standardization ISO. Information Processing Systems – Open Systems Interconnection – Guidelines for the Application of Estelle, LOTOS, and SDL, 1991.

[ISO94] International Organization for Standardization ISO. ISO 8402 Quality Management and Quality Assurance: Vocabulary, 1994.

[ISO01] International Organization for Standardization ISO/IEC. *ISO/IEC 9126. Software Engineering - Product Quality.* ISO/IEC, 2001.

[ISO05] International Organization for Standardization ISO. 9000:2005 Quality Management Systems – Fundamentals and Vocabulary, 2005.

[Jef94] Jeffrey J. Joyce and Carl-Johan H. Seger. The HOL-Voss System: Model-Checking Inside a General-Purpose Theorem-Prover. In *HUG '93: Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 185–198, London, UK, 1994. Springer-Verlag.

[Jen91] Kurt Jensen. Coloured Petri Nets: a High Level Language for System Design and Analysis. In *APN 90: Proceedings on Advances in Petri Nets 1990*, pages 342–416, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[Jen00a] E. Douglas Jensen. Utility Functions: A General Scalable Technology for Software Execution Timeliness as a Quality of Service Part 1: Motivation, 2000.

[Jen00b] E. Douglas Jensen. Utility Functions: A General Scalable Technology for Software Execution Timeliness as a Quality of Service Part 2: The Utility Function Model of Timeliness, 2000.

[JLT85] E. Douglas Jensen, Carey Douglass Locke, and Hideyuki Tokuda. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *IEEE Real-Time Systems Symposium*, pages 112–122, 1985.

[JW96] Daniel Jackson and Jeannette Wing. Lightweight Formal Methods. *Computer*, 29(4):21–22, 1996.

[KDSS07] Alexander Koenen-Dresp, Helge Scheidig, and Sebastian Schöning. Internal Technical Report on Multimedia Oriented Soft Real-Time Systems. Technical report, Saarland University, 2007.

[Kel95] Peter Kelb. *Abstraktionstechniken für automatische Verifikationsverfahren (in German)*. PhD thesis, Oldenburg University, Oldenburg, Germany, 1995.

[Koe02] Alexander Koenen. Untersuchung und Verbesserung des Zeitverhaltens aktueller Linux Kernel (in German). Master's thesis, University of the Federal Armed Forces Munich, 2002.

[Koh07] Sascha Kohn. Wiedergabe und Präsentation von tracebasierten Implementierungssichten des Linuxkerns (in German). Master's thesis, Saarland University, 2007.

[Kon96] Fabio Kon. Were Microkernels a Good Idea That Did Not Work?, 1996.

[Koy90] Ron Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Syst.*, 2(4):255–299, 1990.

[Koz97] Dexter C. Kozen. *Automata and Computability*. Springer, New York, 3. edition, 1997.

[KP91] Yonit Kesten and Amir Pnueli. Timed and Hybrid Statecharts and Their Textual Representation. In *Proceedings of the Second International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 591–620, London, UK, 1991. Springer-Verlag.

[KS05] Jiantao Kong and Karsten Schwan. KStreams: Kernel Support for Efficient Data Streaming in Proxy Servers. In *NOSSDAV '05: Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 159–164, New York, NY, USA, 2005. ACM Press.

[KT04] Gerwin Klein and Harvey Tuch. Towards Verified Virtual Memory in L4. In Konrad Slind, editor, *TPHOLs Emerging Trends '04*, page 16 pages, Park City, Utah, USA, September 2004.

[Küh06] Thomas Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling (SoSyM)*, 5(4):369–385, December 2006.

[Kwo00] Gihwon Kwon. Rewrite Rules and Operational Semantics for Model Checking UML Statecharts. In *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 528–540. Springer, 2000.

[Leo91] K. Leopere. Mach 3 Kernel Principles, 1991.

[LH00] Jork Löser and Michael Hohmuth. Omega0: A Portable Interface to Interrupt Hardware for L4 Systems, 2000.

[Lie92] Jochen Liedtke. Clans & Chiefs. In *Architektur von Rechensystemen, 12. GI/ITG-Fachtagung*, pages 294–305, London, UK, 1992. Springer-Verlag.

[Lie96a] Jochen Liedtke. L4 Reference Manual - 486, Pentium, Pentium Pro, 1996.

[Lie96b] Jochen Liedtke. Towards Real Microkernels. *Commun. ACM*, 39(9):70–77, 1996.

[Liu00] Jane W. S. W. Liu. *Real-Time Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[LKPB06] Caixue Lin, Tim Kaldewey, Anna Povzner, and Scott A. Brandt. Diverse Soft Real-Time Processing in an Integrated System. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 369–378, Washington, DC, USA, 2006. IEEE Computer Society.

[LMM99] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.

[Loc86] Carey Douglass Locke. *Best-Effort Decision-Making for Real-Time Scheduling.* PhD thesis, 1986.

[LP99a] Johan Lilius and Ivan Porres Paltor. Formalising UML State Machines for Model Checking. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 430–445. Springer, 1999.

[LP99b] Johan Lilius and Ivan Porres Paltor. vUML: a Tool for Verifying UML Models. Technical report, 1999.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LRGW95] Phillip A. Laplante, Eileen P. Rose, and Maria Gracia-Watson. An Historical Survey of Early Real-Time Computing Developments in the U.S. *Real-Time Syst.*, 8(2-3):199–213, 1995.

[LS07] Matthew Lang and Paolo A. G. Sivilotti. A Distributed Maximal Scheduler for Strong Fairness. In *DISC*, pages 358–372, 2007.

[Luc03] Michael Lucas. *Absolute OpenBSD: UNIX for the Practical Paranoid.* No Starch Press, Inc, 2003.

[Mar65] James Martin. *Programming Real-Time Computer Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1965.

[McM92a] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Pittsburgh, PA, USA, 1992.

[McM92b] Kenneth Lauchlin McMillan. The SMV System. Technical Report CMU-CS-92-131, 1992.

[Mea55] George H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Mik00] Erich Mikk. *Semantics and Verification of Statecharts.* PhD thesis, Christian Albrecht University Kiel, Kiel, Germany, 2000.

[Mil80] Robert Milner. A Calculus of Communicating Systems. volume 92 of *Lecture Notes in Computer Science.* Springer, 1980.

[Mil82] Robert Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[MIP05a] MIPS. Vol I: Introduction to the MIPS64 Architecture – Revision 2.50, 2005.

[MIP05b] MIPS. Vol II: The MIPS64 Instruction Set – Revision 2.50, 2005.

[MIP05c] MIPS. Vol III: The MIPS64 Privileged Resource Architecture – Revision 2.50, 2005.

[MLS97] Erich Mikk, Yassine Lakhnech, and Michael Siegel. Hierarchical Automata as Model for Statecharts. In *ASIAN '97: Proceedings of the Third Asian Computing Science Conference on Advances in Computing Science*, pages 181–196, London, UK, 1997. Springer-Verlag.

[MLSH98] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing Statecharts in PROMELA/SPIN. In *WIFT '98: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, page 90, Washington, DC, USA, 1998. IEEE Computer Society.

[MM06] Jim Mauro and Richard McDougall. *Solaris Internals.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2. edition, 2006.

[Mol02] Ingo Molnar. The Preemtion Patch, 2002.

[Mon03] Jean-Francois Monin. *Understanding Formal Methods.* Springer, 2003.

[Moo56] Edward F. Moore. Gedanken-Experiments on Sequential Machines. In Claude E. Shannon and J. MacCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, 1956.

[Mos97] David Mosberger. *SCOUT: A Path-Based Operating System*. PhD thesis, The University of Arizona, 1997.

[MP96] David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. *SIGOPS Oper. Syst. Rev.*, 30(SI):153–167, 1996.

[MT01] Andreas Mitschele-Thiel. *Systems Engineering with SDL: Developing Performance-Critical Communication*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[Mül08] Markus Müller. Migration of the Component Extension into the Linux Kernel. Master's thesis, Saarland University, March 2008.

[NL97] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: a Scheduler for Multimedia Applications. *SIGOPS Oper. Syst. Rev.*, 31(5):184–197, 1997.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1st edition, December 1992.

[OMG05] OMG Object Management Group. *MOF 2.0/XMI Mapping Specification, v2.1*, September 2005.

[OMG07] OMG Object Management Group. Unified Modeling Language Specification 2.1.1, February 2007.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten (in German)*. PhD thesis, 1962.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*. IEEE Press, 1977.

[Pnu81] Amir Pnueli. A Temporal Logic of Concurrent Programs. In *Theoretical Computer Science*, volume 13, pages 45–60, 1981.

[PS91] Amir Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In *TACS '91: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 244–264, London, UK, 1991. Springer-Verlag.

[PS98] Jan Philipps and Peter Scholz. Formal Verification and Hardware Design with Statecharts. In *Proceedings of the ESPRIT Working Group 8533 on Prospects for Hardware Foundations*, pages 356–389, London, UK, 1998. Springer-Verlag.

[QNX06] Software Systems QNX. QNX - A Commercial Realtime System, 2006.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[RB69] J. E. Rodrigues and Jorge E Rodriguez Bezos. A Graph Model For Parallel Computations. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1969.

[Rei87] Wolfgang Reisig. Place/Transition Systems. In *Proceedings of an Advanced Course on Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986-Part I*, pages 117–141, London, UK, 1987. Springer-Verlag.

[RHS+96] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A Aziz, S. T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a System for Verification and Synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 428–432, New Brunswick, NJ, USA, 1996. Springer Verlag.

[Rid72] William Riddle. Hierarchical Modeling of Operating System Structure and Behavior. In *ACM '72: Proceedings of the ACM annual conference*, pages 1105–1127, New York, NY, USA, 1972. ACM Press.

[RJB05] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Boston, MA, 2005.

[RJO+89] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A System Software Kernel , February 1989.

[RS59] M. O. Rabin and D. Scott. Finite Automata and their Decision Problems. *j-IBM-JRD*, pages 114–125, 1959.

[RT86] G Rozenberg and P. S. Thiagarajan. Petri Nets: Basic Notions, Structure, Behaviour. pages 585–668, 1986.

[Rus99] D. A. Rusling. *The Linux Kernel*. Published as eBook, 1999.

[Sch03] Sebastian Schöning. TwinUx@SB – eine Plattform zur Integration multimedialer und interaktiver Verarbeitung (in German). Master's thesis, Saarland University, 2003.

[Sch04] Klaus Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. SpringerVerlag, 2004.

[Sch06] Helge Scheidig. Twinux: Architecture and Realizaion - Internal Technical Report (in German). Technical report, Universität des Saarlandes, May 2006.

[Sch08] Sebastian Schöning. *Working Title: The Component Extension - Still Unpublished*. PhD thesis, Saarland University, 2008.

[SG91] Abraham Silberschatz and Peter Baer Galvin. *Operating Systems Concepts*. Addison-Wesley Publishing Company, 4. edition, 1991.

[SGG01] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[SH01] Richard O. Sinnott and Dieter Hogrefe. Finite State Machine Based: SDL. pages 55–76, 2001.

[Sha71] S. C. Shapiro. A Net Structure for Semantic Information Storage, Deduction and Retrieval. In *Proc. of the 2nd IJCAI*, pages 512–523, London, UK, 1971.

[Sim04] Jens Simon. TwinUx@SB – Eine experimentelle Implementierung von Koordinator und Multimedia-Anteil (in German). Master's thesis, Saarland University, 2004.

[SK95] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages - A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[SM07] Sebastian Schöning and Markus Müller. Providing Support for Multimedia-Oriented Applications under Linux. *Proceeding of the Ninth Real-Time Linux Workshop*, (49-59), November 2007.

[SS86] Harold S Stone and Paolo Sipala. The Average Complexity of Depth-First Search with Backtracking and Cutoff. *IBM J. Res. Dev.*, 30(3):242–258, 1986.

[SS96] C. Small and M. Seltzer. A Comparison of OS Extension Technologies. In *Proc. 1996 Annual USENIX Technical Conference*, pages 41–54, January 1996.

[SS03] Helge Scheidig and Reinhard Spurk. Motivation und Ideen für Betriebssystem S0 - Internal Technical Paper (in German). Technical report, Saarland University, February 2003.

[SSBD99] Sanjit A. Seshia, R. K. Shyamasundar, A. K. Bhattacharjee, and S. D. Dhodapkar. A Translation of Statecharts to Esterel. In *World Congress on Formal Methods (2)*, pages 983–1007. Springer, 1999.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie (in German)*. Springer-Verlag, Wien New York, 1973.

[STMW04] I. Schinz, T. Toben, C. Mrugalla, and B. Westphal. The Rhapsody UML Verification Environment. *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, 2004.

[SUN04a] Sun Microsystems Inc. SUN. UltraSPARC III cu User's Manual - Version 2.2.1, 2004.

[SUN04b] Sun Microsystems Inc. SUN. UltraSPARC IV Processor - User's Manual Supplement - Version 1.0, 2004.

[SUN05] Sun Microsystems Inc. SUN. UltraSPARC IV Processor - User's Manual Supplement - Version 1.0, 2005.

[THH01] H. Tews, Hermann Härtig, and Michael Hohmuth. VFiasco — Towards a Provably Correct μ–Kernel. Technical Report TUD-FI01-1 – January 2001, Dresden University of Technology, Department of Computer Science, 2001.

[TIM+00] Toshiba, Intel, Microsoft, Phoenix, and Hewlett-Packard. Advanced Configuration and Power Interface Specification, 2000.

[TQC04] Viet-Anh Vu Tran, Shengchao Qin, and Wei-Ngan Chin. An Automatic Mapping from Statecharts to Verilog. In *ICTAC*, pages 187–203, 2004.

[Tur87] Kenneth J Turner. Lotos – A practical Formal Description Technique for OSI. In *International Open Systems 87*, volume 1, pages 265–279. Online Publications, London, March 1987.

[vdB94] Michael von der Beeck. A Comparison of Statecharts Variants. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag.

[Ven80] J Venn. On the Diagramatic and Mechanical Representation of Propositions and Reasonings, 1880.

[W3C03] W3C. Scalable Vector Graphics (SVG) 1.1 Specification, January 2003.

[Wan98] Jiacun Wang. *Timed Petri Nets: Theory and Application.* Kluwer Acadamic Publishers, USA, October 1998.

[WDQ02] Ji Wang, Wei Dong, and Zhi-Chang Qi. Slicing Hierarchical Automata for Model Checking UML Statecharts. In *ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods*, pages 435–446, London, UK, 2002. Springer-Verlag.

[WG00] D. L. Weaver and T. Germont. *The SPARC Architecture Manual.* Prentice Hall, Santa Clara (CA), 2000.

[Wie03] R. J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate and the UML.* Morgan Kaufmann Publishers, 2003.

[Wie07] Alexander Wieder. Modelling the IRQ Subsystem of L4Ka::Pistacchio with the ESF. Bachelor's Thesis, August 2007.

[XQ05] Baowen Xu and Junyan Qian. Model Checking for Statecharts. In *Proceedings of the Ninth IASTED Intern. Conf. Software Engineering and Applications*, pages 183–187. ACTA, November 2005.

[Zur97] Richard Zurawski. Petri Net Models, Functionality and Functional Abstractions, and Applications to the Design of Automated Manufacturing Systems. In *Emerging Technologies and Factory Automation Proceedings*, pages 339–346, 1997.

Appendix A

Analysis Data

This first appendix contains all detailed analysis data about the three systems under investigation that is summarized and interpreted in Chapter 7. Note that some of the data is derived directly from the system models as in [Gog07] and [Wie07]. The full models can be found in the file archive, see appendix D.

A.1 Immediate vs. Deferred Interrupt Handling

The possible interruption handling states according to the axioms defined in Section 6.1.1 are listed for the three SuIs.

A.1.1 Linux

In Linux, most interrupt service routines can be implemented either way. Thus, only the ones that have a fixed assignment are elaborated. The following four interrupt handlers are always executed immediately:

$$\begin{aligned}
\mathfrak{hdl}(DEVT_{x,INT,IPI,0}) &= \mathsf{DO_RESCHED} = \mathfrak{hdl}^*(DEVT_{x,INT,IPI,0}) \\
\mathfrak{hdl}(DEVT_{x,INT,IPI,1}) &= \mathsf{DO_CALL_FUN} = \mathfrak{hdl}^*(DEVT_{x,INT,IPI,1}) \\
\mathfrak{hdl}(DEVT_{x,INT,IPI,2}) &= \mathsf{DO_INV_TLB} = \mathfrak{hdl}^*(DEVT_{x,INT,IPI,2}) \\
\mathfrak{hdl}(DEVT_{x,INT,GLOB_TIMER}) &= \mathsf{GT_ISR} = \mathfrak{hdl}^*(DEVT_{x,INT,GLOB_TIMER})
\end{aligned}$$

The following two service routines are necessarily executed in a deferred manner:

$$\begin{aligned}
\mathfrak{hdl}(DEVT_{x,INT,LOC_TIMER}) &= \mathsf{LT_ISR} \neq \\
\mathfrak{hdl}^*(DEVT_{x,INT,LOC_TIMER}) &= \{\mathsf{LT_ISR}, \mathsf{RUN_SOFTIRQ}\}
\end{aligned}$$

$$\begin{aligned}
\mathfrak{hdl}(DEVT_{x,INT,I/O,0=NIC}) &= \mathsf{UNDISRUPT_0} \neq \\
\mathfrak{hdl}^*(DEVT_{x,INT,I/O,0=NIC}) &= \{\mathsf{UNDISRUPT_0}, \mathsf{RUN_SOFTIRQ}\}
\end{aligned}$$

A.1.2 OpenBSD

Note that the latter parts of state identifiers given with their full hierarchical names, such as STATE.DESCENDENT are not included in the model extracts in this thesis, but are

part of [Gog07]. In OpenBSD, nearly all interrupt handlers are mandatorily immediate:

$$\begin{aligned}
\mathfrak{hdl}(DEVT_{x,INT,IPI,HLT}) &= \mathsf{IPI.HALT} &= \mathfrak{hdl}^*(DEVT_{x,INT,IPI,HLT})\\
\mathfrak{hdl}(DEVT_{x,INT,IPI,SET}) &= \mathsf{IPI.IPI_MICROSET} &= \mathfrak{hdl}^*(DEVT_{x,INT,IPI,SET})\\
\mathfrak{hdl}(DEVT_{x,INT,IPI,TLB}) &= \mathsf{IPI.TLB_SHOOTDOWN} &= \mathfrak{hdl}^*(DEVT_{x,INT,IPI,TLB})\\
\mathfrak{hdl}(DEVT_{x,INT,IPI,FLUSH}) &= \mathsf{IPI.FLUSH_FPU} &= \mathfrak{hdl}^*(DEVT_{x,INT,IPI,FLUSH})\\
\mathfrak{hdl}(DEVT_{x,INT,IPI,SYNC}) &= \mathsf{IPI.SYNCH_FPU} &= \mathfrak{hdl}^*(DEVT_{x,INT,IPI,SYNCH})\\
\mathfrak{hdl}(DEVT_{x,INT,IPI,DB}) &= \mathsf{IPI.IPI_DB} &= \mathfrak{hdl}^*(DEVT_{x,INT,IPI,DB})\\
\mathfrak{hdl}(DEVT_{x,INT,TIME}) &= \mathsf{LAPIC_TIMER} &= \mathfrak{hdl}^*(DEVT_{x,INT,TIME})\\
\mathfrak{hdl}(DEVT_{x,INT,I/O,HDD}) &= \mathsf{IOAPIC_HDL} &= \mathfrak{hdl}^*(DEVT_{x,INT,I/O,HDD})\\
\mathfrak{hdl}(DEVT_{x,INT,I/O,MPEG}) &= \mathsf{IOAPIC_HDL} &= \mathfrak{hdl}^*(DEVT_{x,INT,I/O,MPEG})\\
\mathfrak{hdl}(DEVT_{x,INT,I/O,HID}) &= \mathsf{IOAPIC_HDL} &= \mathfrak{hdl}^*(DEVT_{x,INT,I/O,HID})\\
\mathfrak{hdl}(DEVT_{x,INT,I/O,SND}) &= \mathsf{IOAPIC_HDL} &= \mathfrak{hdl}^*(DEVT_{x,INT,I/O,SND})\\
\mathfrak{hdl}(DEVT_{x,INT,I/O,KB}) &= \mathsf{IOAPIC_HDL} &= \mathfrak{hdl}^*(DEVT_{x,INT,I/O,KB})
\end{aligned}$$

The only exceptional case is the network card service routine as shown by the following evaluated axiom:

$$\begin{aligned}
\mathfrak{hdl}(DEVT_{x,INT,NIC}) &= \mathsf{IOAPIC_HDL} &\neq\\
\mathfrak{hdl}^*(DEVT_{x,INT,NIC}) &= \{\mathsf{IOAPIC_HDL}, \mathsf{XDORETI.SOFWTARE_INTERRUPT}\}
\end{aligned}$$

A.1.3 Pistachio

In the μ-kernel system Pistachio, only two system-internal interrupts are handled immediately:

$$\begin{aligned}
\mathfrak{hdl}(DEVT_{x,INT,TIME}) &= \mathsf{TIMER_HANDLER} &= \mathfrak{hdl}^*(DEVT_{x,INT,TIME})\\
\mathfrak{hdl}(DEVT_{x,INT,IPI}) &= \mathsf{PROCESS_XPCU_MAILBOX} &= \mathfrak{hdl}^*(DEVT_{x,INT,IPI})
\end{aligned}$$

All other six interrupt handlers, which are the service routines for the peripheral I/O devices, are handled in a deferred way:

$$\begin{aligned}
\mathfrak{hdl}(DEVT_{x,INT,I/O,HDD}) &= \mathsf{HW_IRQ_HANDLER} &\neq\\
\mathfrak{hdl}^*(DEVT_{x,INT,I/O,HDD}) &= \{\mathsf{HW_IRQ_HANDLER}, \mathsf{HDL_THREAD}\}
\end{aligned}$$

$$\begin{aligned}
\mathfrak{hdl}(DEVT_{x,INT,I/O,MPEG}) &= \mathsf{HW_IRQ_HANDLER} &\neq\\
\mathfrak{hdl}^*(DEVT_{x,INT,I/O,MPEG}) &= \{\mathsf{HW_IRQ_HANDLER}, \mathsf{HDL_THREAD}\}
\end{aligned}$$

$$\begin{aligned}
\mathfrak{hdl}(DEVT_{x,INT,I/O,HID}) &= \mathsf{HW_IRQ_HANDLER} &\neq\\
\mathfrak{hdl}^*(DEVT_{x,INT,I/O,HID}) &= \{\mathsf{HW_IRQ_HANDLER}, \mathsf{HDL_THREAD}\}
\end{aligned}$$

$$\begin{aligned}
\mathfrak{hdl}(DEVT_{x,INT,I/O,SND}) &= \mathsf{HW_IRQ_HANDLER} &\neq \\
\mathfrak{hdl}^*(DEVT_{x,INT,I/O,SND}) &= \{\mathsf{HW_IRQ_HANDLER}, \mathsf{HDL_THREAD}\} &
\end{aligned}$$

$$\begin{aligned}
\mathfrak{hdl}(DEVT_{x,INT,I/O,NIC}) &= \mathsf{HW_IRQ_HANDLER} &\neq \\
\mathfrak{hdl}^*(DEVT_{x,INT,I/O,NIC}) &= \{\mathsf{HW_IRQ_HANDLER}, \mathsf{HDL_THREAD}\} &
\end{aligned}$$

$$\begin{aligned}
\mathfrak{hdl}(DEVT_{x,INT,I/O,KB}) &= \mathsf{HW_IRQ_HANDLER} &\neq \\
\mathfrak{hdl}^*(DEVT_{x,INT,I/O,KB}) &= \{\mathsf{HW_IRQ_HANDLER}, \mathsf{HDL_THREAD}\} &
\end{aligned}$$

A.2 Interrupt Handlers Subject to Scheduling

The severeness of deferral, i.e. the question whether the deferred handler is subject to the system scheduling is also analyzed by means of axioms. This section provides the evaluation of the axioms defined in Section 6.1.3.

A.2.1 Linux

The only service routine that can be subject to scheduling is the timer handler. Note that if other routines are implemented in a deferred way, they can be subject to scheduling as well.

$$\mathfrak{hdl}^*(DEVT_{x,INT,I/O,NIC}) \qquad \ni \mathsf{TIMER\_HANDLER} \in KT$$

A.2.2 OpenBSD

In OpenBSD, there are no handlers exposed to the overall system's scheduler at all.

A.2.3 Pistachio

Since there is no kernel model handling in Pistachio other than for the two immediate handlers, all deferred handlers are subject to the system scheduling:

$$\begin{aligned}
&\mathfrak{hdl}^*(DEVT_{x,INT,I/O,HDD}) && \ni \mathsf{TIMER_HANDLER} \in U \\
&\mathfrak{hdl}^*(DEVT_{x,INT,I/O,MPEG}) && \ni \mathsf{TIMER_HANDLER} \in U \\
&\mathfrak{hdl}^*(DEVT_{x,INT,I/O,HID}) && \ni \mathsf{TIMER_HANDLER} \in U \\
&\mathfrak{hdl}^*(DEVT_{x,INT,I/O,SND}) && \ni \mathsf{TIMER_HANDLER} \in U \\
&\mathfrak{hdl}^*(DEVT_{x,INT,I/O,NIC}) && \ni \mathsf{TIMER_HANDLER} \in U \\
&\mathfrak{hdl}^*(DEVT_{x,INT,I/O,KB}) && \ni \mathsf{TIMER_HANDLER} \in U
\end{aligned}$$

A.3 Response Behavior - PAGs and Paths

The path analysis graphs as given by Definition 6.3 for the three SuIs are elaborated in the following section. Since complete hierarchical names are used, the identifiers are quite verbose. Sometimes, when the hierarchical prefix is unambiguous, `[...]` abbreviates the identifier for typesetting reasons.

A.3.1 Linux

The Linux PAG contains all interruption paths, also those only accessible when exceptions are raised.

```
V_PAG_LINUX = {START, IKCP.EXC.CPU_HANDLING , IKCP.EXC.SAVE_REGISTERS,
IKCP.EXC.HD.FIXUP, IKCP.EXC.HD.GEN_SIG, IKCP.EXC.PANIC, IKCP.EXC.DF.DF_FIXUP,
IKCP.EXC.DF.DF_PANIC, IKCP.EXC.RETURN_EXC, IKCP.EXC.CPU_RETURN, IKCP.EXC.HD.CHECK_CTX,
IKCP.INT.CPU_HANDLING, IKCP.INT.SAVE_REGISTERS, IKCP.INT.IPI.SAVE_GPR,
IKCP.INT.IPI.DO_RESCHED, IKCP.INT.IPI.DO_CALL_FUN, IKCP.INT.IPI.DO_INV_TLB,
IKCP.INT.IO_ISR.DO_INT, IKCP.INT.IO_ISR.UNDISRUPT_0, IKCP.INT.IO_ISR.DISRUPT_0,
IKCP.INT.IO_ISR.ACT_TASK_0, IKCP.INT.IO_ISR.UNDISRUPT_1, IKCP.INT.IO_ISR.DISRUPT_1,
IKCP.INT.IO_ISR.ACT_TASK_1, IKCP.INT.IO_ISR.UNDISRUPT_2, IKCP.INT.IO_ISR.DISRUPT_2,
IKCP.INT.IO_ISR.ACT_TASK_2, IKCP.INT.IO_ISR.UNDISRUPT_3, IKCP.INT.IO_ISR.DISRUPT_3,
IKCP.INT.IO_ISR.ACT_TASK_3, IKCP.INT.IO_ISR.UNDISRUPT_4, IKCP.INT.IO_ISR.DISRUPT_4,
IKCP.INT.IO_ISR.ACT_TASK_4, IKCP.INT.IO_ISR.UNDISRUPT_5, IKCP.INT.IO_ISR.DISRUPT_5,
IKCP.INT.IO_ISR.ACT_TASK_5, IKCP.INT.IO_ISR.EXIT_INT, IKCP.INT.LT_ISR.DO_INT,
IKCP.INT.LT_ISR.PROF_TICK, IKCP.INT.LT_ISR.UPD_PROC_TIMES.ACCOUNTING.UPD_TIME,
IKCP.INT.LT_ISR.UPD_PROC_TIMES.ACCOUNTING.SENDSIG,
IKCP.INT.LT_ISR.UPD_PROC_TIMES.ACCOUNTING.IT_TIMER,
IKCP.INT.LT_ISR.UPD_PROC_TIMES.TIMER_SOFTIRQ,
IKCP.INT.LT_ISR.UPD_PROC_TIMES.TICK.BALANCE,
IKCP.INT.LT_ISR.UPD_PROC_TIMES.TICK.SET_RESCHED, IKCP.INT.LT_ISR.EXIT_INT,
IKCP.INT.GT_ISR.DO_INT, IKCP.INT.GT_ISR.CHECK_LOST, IKCP.INT.GT_ISR.JIFFIES,
IKCP.INT.GT_ISR.UPD_TIME, IKCP.INT.GT_ISR.EXIT_INT, IKCP.INT.SOFT.DISABLE_DEF_FUN,
IKCP.INT.SOFT.CHECK_SOFTIRQ, IKCP.INT.SOFT.RUN_SOFTIRQ, IKCP.INT.SOFT.SIG_KSOFTIRQD,
IKCP.INT.RETURN_INT, IKCP.INT.CPU_RETURN,
IKCP.SYSCALL.SYSENTER, IKCP.SYSCALL.SYSC, IKCP.SYSCALL.SYSEXIT,
IKCP.SCHEDULE, FINAL}

E_PAG_LINUX = {(START,IKCP.EXC.CPU_HANDLING),
(IKCP.EXC.CPU_HANDLING, IKCP.EXC.SAVE_REGISTERS),
(IKCP.EXC.SAVE_REGISTERS, IKCP.EXC.HD.CHECK_CTX),
(IKCP.EXC.HD.CHECK_CTX, IKCP.EXC.HD.FIXUP),
(IKCP.EXC.HD.CHECK_CTX, IKCP.EXC.HD.PANIC),
(IKCP.EXC.HD.FIXUP, IKCP.EXC.HD.GEN_SIG),
(IKCP.EXC.HD.GEN_SIG, IKCP.EXC.RETURN_EXC),
(IKCP.EXC.SAVE_REGISTERS, IKCP.EXC.DF.DF_FIXUP),
(IKCP.EXC.DF.DF_FIXUP, IKCP.EXC.RETURN_EXC),
(IKCP.EXC.DF.DF_FIXUP, IKCP.EXC.DF.DF_PANIC),
(IKCP.EXC.RETURN_EXC, IKCP.EXC.CPU_RETURN),
(IKCP.EXC.CPU_RETURN, IKCP.SCHEDULE),
(START, IKCP.INT.CPU_HANDLING),
(IKCP.INT.CPU_HANDLING, IKCP.INT.SAVE_REGISTERS),
(IKCP.INT.SAVE_REGISTERS, IKCP.INT.IPI.SAVE_GPR),
```

```
(IKCP.INT.IPI.SAVE_GPR, IKCP.INT.IPI.DO_RESCHED),
(IKCP.INT.IPI.SAVE_GPR, IKCP.INT.IPI.DO_CALL_FUN),
(IKCP.INT.IPI.SAVE_GPR, IKCP.INT.IPI.DO_INV_TLB),
(IKCP.INT.IPI.DO_RESCHED, IKCP.INT.SOFT.DISABLE_DEF_FUN),
(IKCP.INT.IPI.DO_CALL_FUN, IKCP.INT.SOFT.DISABLE_DEF_FUN),
(IKCP.INT.IPI.DO_INV_TLB, IKCP.INT.SOFT.DISABLE_DEF_FUN),
(IKCP.INT.IPI.DO_RESCHED, IKCP.INT.RETURN_INT),
(IKCP.INT.IPI.DO_CALL_FUN, IKCP.INT.RETURN_INT),
(IKCP.INT.IPI.DO_INV_TLB, IKCP.INT.RETURN_INT),
(IKCP.INT.SAVE_REGISTERS, IKCP.INT.IO_ISR.DO_INT),
(IKCP.INT.IO_ISR.DO_INT, IKCP.INT.IO_ISR.UNDISRUPT_0),
(IKCP.INT.IO_ISR.DO_INT, IKCP.INT.IO_ISR.UNDISRUPT_1),
(IKCP.INT.IO_ISR.DO_INT, IKCP.INT.IO_ISR.UNDISRUPT_2),
(IKCP.INT.IO_ISR.DO_INT, IKCP.INT.IO_ISR.UNDISRUPT_3),
(IKCP.INT.IO_ISR.DO_INT, IKCP.INT.IO_ISR.UNDISRUPT_4),
(IKCP.INT.IO_ISR.DO_INT, IKCP.INT.IO_ISR.UNDISRUPT_5),
(IKCP.INT.IO_ISR.UNDISRUPT_0, IKCP.INT.IO_ISR.DISRUPT_0),
(IKCP.INT.IO_ISR.UNDISRUPT_1, IKCP.INT.IO_ISR.DISRUPT_1),
(IKCP.INT.IO_ISR.UNDISRUPT_2, IKCP.INT.IO_ISR.DISRUPT_2),
(IKCP.INT.IO_ISR.UNDISRUPT_3, IKCP.INT.IO_ISR.DISRUPT_3),
(IKCP.INT.IO_ISR.UNDISRUPT_4, IKCP.INT.IO_ISR.DISRUPT_4),
(IKCP.INT.IO_ISR.UNDISRUPT_5, IKCP.INT.IO_ISR.DISRUPT_5),
(IKCP.INT.IO_ISR.DISRUPT_0, IKCP.INT.IO_ISR.ACT_TASK_0),
(IKCP.INT.IO_ISR.DISRUPT_1, IKCP.INT.IO_ISR.ACT_TASK_1),
(IKCP.INT.IO_ISR.DISRUPT_2, IKCP.INT.IO_ISR.ACT_TASK_2),
(IKCP.INT.IO_ISR.DISRUPT_3, IKCP.INT.IO_ISR.ACT_TASK_3),
(IKCP.INT.IO_ISR.DISRUPT_4, IKCP.INT.IO_ISR.ACT_TASK_4),
(IKCP.INT.IO_ISR.DISRUPT_5, IKCP.INT.IO_ISR.ACT_TASK_5),
(IKCP.INT.IO_ISR.ACT_TASK_0, IKCP.INT.IO_ISR.EXIT_INT),
(IKCP.INT.IO_ISR.ACT_TASK_1, IKCP.INT.IO_ISR.EXIT_INT),
(IKCP.INT.IO_ISR.ACT_TASK_2, IKCP.INT.IO_ISR.EXIT_INT),
(IKCP.INT.IO_ISR.ACT_TASK_3, IKCP.INT.IO_ISR.EXIT_INT),
(IKCP.INT.IO_ISR.ACT_TASK_4, IKCP.INT.IO_ISR.EXIT_INT),
(IKCP.INT.IO_ISR.ACT_TASK_5, IKCP.INT.IO_ISR.EXIT_INT),
(IKCP.INT.IO_ISR.EXIT_INT, IKCP.INT.SOFT.DISABLE_DEF_FUN),
(IKCP.INT.IO_ISR.EXIT_INT, IKCP.INT.RETURN_INT),
(IKCP.INT.SAVE_REGISTERS, IKCP.INT.LT_ISR.DO_INT),
(IKCP.INT.LT_ISR.DO_INT, IKCP.INT.LT_ISR.PROF_TICK),
(IKCP.INT.LT_ISR.PROF_TICK, IKCP.INT.LT_ISR.UPD_PROC_TIMES.ACCOUNTING.UPD_TIME),
(IKCP.[...].ACCOUNTING.UPD_TIME, IKCP.[...].ACCOUNTING.SENDSIG),
(IKCP.[...].ACCOUNTING.SENDSIG, IKCP.[...].ACCOUNTING.IT_TIMER),
(IKCP.[...].ACCOUNTING.IT_TIMER,IKCP.INT.LT_ISR.UPD_PROC_TIMES.TIMER_SOFTIRQ),
(IKCP.INT.LT_ISR.UPD_PROC_TIMES.TIMER_SOFTIRQ, IKCP.[...].TICK.BALANCE),
(IKCP.[...].TICK.BALANCE, IKCP.[...].TICK.SET_RESCHED),
(IKCP.[...].TICK.SET_RESCHED, IKCP.INT.LT_ISR.EXIT_INT),
(IKCP.INT.LT_ISR.EXIT_INT, IKCP.INT.SOFT.DISABLE_DEF_FUN),
(IKCP.INT.LT_ISR.EXIT_INT, IKCP.INT.RETURN_INT),
(IKCP.INT.SAVE_REGISTERS, IKCP.INT.GT_ISR.DO_INT),
(IKCP.INT.GT_ISR.DO_INT, IKCP.INT.GT_ISR.CHECK_LOST),
(IKCP.INT.GT_ISR.CHECK_LOST, IKCP.INT.GT_ISR.JIFFIES),
(IKCP.INT.GT_ISR.JIFFIES, IKCP.INT.GT_ISR.UPD_TIME),
(IKCP.INT.GT_ISR.UPD_TIME, IKCP.INT.GT_ISR.EXIT_INT),
```

```
(IKCP.INT.GT_ISR.EXIT_INT, IKCP.INT.RETURN_INT),
(IKCP.INT.GT_ISR.EXIT_INT, IKCP.INT.SOFT.DISABLE_DEF_FUN),
(IKCP.INT.SOFT.DISABLE_DEF_FUN, IKCP.INT.SOFT.CHECK_SOFTIRQ),
(IKCP.INT.SOFT.CHECK_SOFTIRQ, IKCP.INT.SOFT.SIG_KSOFTIRQD),
(IKCP.INT.SOFT.CHECK_SOFTIRQ, IKCP.INT.SOFT.RUN_SOFTIRQ),
(IKCP.INT.SOFT.RUN_SOFTIRQ, IKCP.INT.SOFT.CHECK_SOFTIRQ),
(IKCP.INT.SOFT.SIG_KSOFTIRQD,IKCP.INT.RETURN_INT),
(IKCP.INT.RETURN_INT, IKCP.INT.CPU_RETURN),
(IKCP.INT.RETURN_INT, IKCP.SCHEDULE),
(START, IKCP.SYSCALL.SYSENTER),
(IKCP.SYSCALL.SYSENTER, IKCP.SYSCALL.SYSC),
(IKCP.SYSCALL.SYSC, IKCP.SYSCALL.SYSEXIT),
(IKCP.SYSCALL.SYSEXIT, SCHEDULE),
(IKCP.SCHEDULE, FINAL)}
```

Since all edges of the Linux PAG are black, the color function is not provided explicitly. Table A.1 lists the all path length tuples for any possible interrupt. It is directly derived from the Linux PAG by applying the path definition 6.4.

Disruptive Event	Handler state (v_{ana})	Min Length	Max Length
$DEVT_{x,INT,IPI,0}$	$\mathfrak{hdl}(DEVT_{x,INT,IPI,0}) =$ DO_RESCHED	(8,0,0,0)	(12,1,2,0)
$DEVT_{x,INT,IPI,0}$	$\mathfrak{hdl}(DEVT_{x,INT,IPI,1}) =$ DO_CALL_FUN	(8,0,0,0)	(11,1,2,0)
$DEVT_{x,INT,IPI,0}$	$\mathfrak{hdl}(DEVT_{x,INT,IPI,2}) =$ DO_INV_TLB	(8,0,0,0)	(11,1,2,0)
$DEVT_{x,INT,IO,HDD}$	$\mathfrak{hdl}(DEVT_{x,INT,I/O,HDD}) =$ UNDISRUPT_0	(11,0,0,0)	(14,1,2,0)
$DEVT_{x,INT,IO,MPEG}$	$\mathfrak{hdl}(DEVT_{x,INT,I/O,MPEG}) =$ UNDISRUPT_1	(11,0,0,0)	(14,1,2,0)
$DEVT_{x,INT,IO,HID}$	$\mathfrak{hdl}(DEVT_{x,INT,I/O,HID}) =$ UNDISRUPT_2	(11,0,0,0)	(14,1,2,0)
$DEVT_{x,INT,IO,SND}$	$\mathfrak{hdl}(DEVT_{x,INT,I/O,SND}) =$ UNDISRUPT_3	(11,0,0,0)	(14,1,2,0)
$DEVT_{x,INT,IO,NIC}$	$\mathfrak{hdl}(DEVT_{x,INT,I/O,NIC}) =$ UNDISRUPT_4	(11,0,0,0)	(14,1,2,0)
$DEVT_{x,INT,IO,KB}$	$\mathfrak{hdl}(DEVT_{x,INT,I/O,KB}) =$ UNDISRUPT_5	(11,0,0,0)[1]	(14,1,2,0)
$DEVT_{x,INT,LOC\_TIMER}$	$\mathfrak{hdl}(DEVT_{x,INT,GLOB\_TIMER}) =$ LT_ISR	(15,0,0,0)	(18,1,2,0)
$DEVT_{x,INT,GLOB\_TIMER}$	$\mathfrak{hdl}(DEVT_{x,INT,LOC\_TIMER}) =$ GT_ISR	11,0,0,0)	(14,1,2,0)

Table A.1: Path lengths of DEVTs in Linux

A.3.2 OpenBSD

The OpenBSD PAG is based on the entire model from [Gog07], not on the excerpts that are provided in Section 4.4.4. Due to this, German terms and abbreviations are used as identifiers rather than English ones. Submachines that are included in several super states, such as SPLX or KERNEL_LOCK_SETZEN do not have all possible prefixes since they are named unambiguously anyway.

```
V_PAG_OPENBSD = {START, UNTR_BEH.SYSTEMAUFRUF.SYS_VOR, UNTR_BEH.SYSTEMAUFRUF.SYS_ERL,
UNTR_BEH.SYSTEMAUFRUF.SYS_NACH, UNTR_BEH.SYSTEMAUFRUF.USERRET,
UNTR_BEH.EXCEPTION_BEHANDLUNG.DNA, UNTR_BEH.EXCEPTION_BEHANDLUNG.FPU,
UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.USR, UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.PAGE,
UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.SPEZIAL, UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.PRUEFE,
UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.PANIC, UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.TRAPSIGNAL,
UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.USERRET,
UNTR_BEH.INTERRUPT_BEHANDLUNG.FRAME_ERSTELLEN,
UNTR_BEH.INTERRUPT_BEHANDLUNG.IO_APIC_INTERRUPT.GRP.INTERRUPT_HANDLER_ERMITTELN,
UNTR_BEH.INTERRUPT_BEHANDLUNG.IO_APIC_INTERRUPT.GRP.STRAY_INTERRUPT,
UNTR_BEH.INTERRUPT_BEHANDLUNG.IO_APIC_INTERRUPT.GRP.NAECHSTER_INTERRUPT_HANDLER,
UNTR_BEH.INTERRUPT_BEHANDLUNG.IO_APIC_INTERRUPT.GRP.STATISTIKEN_AKTUALISIEREN,
UNTR_BEH.INTERRUPT_BEHANDLUNG.IO_APIC_INTERRUPT.IO_APIC_HDL,
UNTR_BEH.INTERRUPT_BEHANDLUNG.IO_APIC_INTERRUPT.EOI,
UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.EOI, UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.IPI_HALT,
UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.IPI_MICROSET, UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.FLUSH_FPU,
UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.TLB_SHOOTDOWN, UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.SYNC_FPU,
UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.IPI_DB,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.EOI,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.TSC_AKT,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.HARDCLOCK.GRP_0.INTERVALZEITGEBER_ITIMER_VIRTUAL,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.HARDCLOCK.GRP_0.INTERVALZEITGEBER_ITIMER_PROF,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.HARDCLOCK.T_ADD_VIR,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.HARDCLOCK.T_ADD_PROF,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.HARDCLOCK.STAT_AKT,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.HARDCLOCK.PRPRIO,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.HARDCLOCK.RR_AKT,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.HARDCLOCK.SPCF_SEENRR,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.HARDCLOCK.RESCHED,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.HARDCLOCK.T_W_AKT,
UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.HARDCLOCK.SETSOFTCLOCK,
UNTR_BEH.INTERRUPT_BEHANDLUNG.SOFTWARE_INTERRUPT_IPI.CPL_SETZEN,
UNTR_BEH.INTERRUPT_BEHANDLUNG.SOFTWARE_INTERRUPT_IPI.DEREG,
UNTR_BEH.INTERRUPT_BEHANDLUNG.SOFTWARE_INTERRUPT_IPI.EOI,
UNTR_BEH.INTERRUPT_NACHBEHANDLUNG.SW_INTR_NACH.UEBERPREUFUNG_SW_INTR,
UNTR_BEH.INTERRUPT_NACHBEHANDLUNG.SW_INTR_NACH.SW_B_L,
UNTR_BEH.INTERRUPT_NACHBEHANDLUNG.SW_INTR_NACH.CPL_SETZEN,
SOFTWARE_INTERRUPT.TTY_BEH, SOFTWARE_INTERRUPT.SOFTCLOCK,SOFTWARE_INTERRUPT.SOFTNET,
KERNEL_LOCK_SETZEN.LOCK_SETZEN.VERSUCH_LOCK_BEKOMMEN,
KERNEL_LOCK_SETZEN.LOCK_SETZEN.ZAEHLER_INKREMENTIEREN,
KERNEL_LOCK_ENTFERNEN.ENTFERNEN.ZAEHLER_DEKREMENTIEREN,
KERNEL_LOCK_ENTFERNEN.ENTFERNEN.LOCK_ENTFERNEN,
SPLX.CPL_SETZEN, SPLX.PRUEF,FINAL}
```

```
E_PAG_OPENBSD = {START, UNTR_BEH.SYSTEMAUFRUF.SYS_VOR),
(SYS_VOR, [...].VERSUCH_LOCK_BEKOMMEN),
(SYS_VOR, [...].ZAEHLER_INKREMENTIEREN),
([...].VERSUCH_LOCK_BEKOMMEN, [...].ZAEHLER_INKREMENTIEREN),
([...].VERSUCH_LOCK_BEKOMMEN, [...].VERSUCH_LOCK_BEKOMMEN),
([...].VERSUCH_LOCK_BEKOMMEN, SPLX.CPL_SETZEN),
([...].ZAEHLER_INKREMENTIEREN, SPLX.CPL_SETZEN),
(SPLX.CPL_SETZEN, SPLX.PRUEF),
(SPLX.PRUEF,UNTR_BEH.SYSTEMAUFRUF.SYS_ERL),
(SPLX.PRUEF,[...].VERSUCH_LOCK_BEKOMMEN),
(SPLX.PRUEF,SOFTWARE_INTERRUPT.TTY_BEH),
(SPLX.PRUEF,SOFTWARE_INTERRUPT.SOFTCLOCK),
(SPLX.PRUEF,SOFTWARE_INTERRUPT.SOFTNET),
(SOFTWARE_INTERRUPT.TTY_BEH, [...].ZAEHLER_DEKREMENTIEREN),
(SOFTWARE_INTERRUPT.SOFTCLOCK, [...].ZAEHLER_DEKREMENTIEREN),
(SOFTWARE_INTERRUPT.SOFTNET,[...].ZAEHLER_DEKREMENTIEREN),
([...].ZAEHLER_DEKREMENTIEREN),[...].LOCK_ENTFERNEN),
([...].ZAEHLER_DEKREMENTIEREN), SPLX.CPL_SETZEN),
(UNTR_BEH.SYSTEMAUFRUF.SYS_ERL,[...].LOCK_ENTFERNEN),
(SPLX.PRUEF,UNTR_BEH.SYSTEMAUFRUF.SYS_NACH),
(UNTR_BEH.SYSTEMAUFRUF.SYS_NACH, UNTR_BEH.SYSTEMAUFRUF.USERRET),
(UNTR_BEH.SYSTEMAUFRUF.USERRET,FINAL),
(START, UNTR_BEH.EXCEPTION_BEHANDLUNG.DNA),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.DNA, FINAL),
(START, UNTR_BEH.EXCEPTION_BEHANDLUNG.FPU),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.FPU, FINAL),
(START,UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.USR),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.USR, UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.PRUEFE),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.PRUEFE, FINAL),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.PRUEFE, UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.PANIC),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.USR, UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.PAGE),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.USR,UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.SPEZIAL),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.USR, [...].VERSUCH_LOCK_BEKOMMEN),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.PAGE, FINAL),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.SPEZIAL, FINAL),
(SPLX.PRUEF,UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.TRAPSIGNAL),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.TRAPSIGNAL,[...].LOCK_ENTFERNEN),
(SPLX.PRUEF,UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.USERRET),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.TRAP.USERRET, FINAL),
(START, UNTR_BEH.INTERRUPT_BEHANDLUNG.FRAME_ERSTELLEN),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.FRAME_ERSTELLEN, [...].GRP.INTERRUPT_HANDLER_ERMITTELN),
([...].GRP.INTERRUPT_HANDLER_ERMITTELN, [...].STRAY_INTERRUPT),
([...].STRAY_INTERRUPT, UNTR_BEH.INTERRUPT_BEHANDLUNG.IO_APIC_INTERRUPT.EOI),
([...].GRP.INTERRUPT_HANDLER_ERMITTELN, [...].VERSUCH_LOCK_BEKOMMEN),
(SPLX.PRUEF, UNTR_BEH.INTERRUPT_BEHANDLUNG.IO_APIC_INTERRUPT.IO_APIC_HDL),
([...].IO_APIC_HDL, [...].STATISTIKEN_AKTUALISIEREN),
([...].STATISTIKEN_AKTUALISIEREN, [...].LOCK_ENTFERNEN),
(SPLX.PRUEF, [...].NAECHSTER_INTERRUPT_HANDLER),
([...].NAECHSTER_INTERRUPT_HANDLER,[...].VERSUCH_LOCK_BEKOMMEN),
([...].NAECHSTER_INTERRUPT_HANDLER,UNTR_BEH.INTERRUPT_BEHANDLUNG.IO_APIC_INTERRUPT.EOI),
([...].IO_APIC_INTERRUPT.EOI, [...].SW_INTR_NACH.UEBERPREUFUNG_SW_INTR),
(START, UNTR_BEH.INTERRUPT_BEHANDLUNG.FRAME_ERSTELLEN),
```

```
(UNTR_BEH.INTERRUPT_BEHANDLUNG.FRAME_ERSTELLEN,UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.EOI),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.EOI,UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.IPI_HALT),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.EOI,UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.IPI_MICROSET),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.IPI_MICROSET, [...].UEBERPREUFUNG_SW_INTR),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.EOI,UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.FLUSH_FPU),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.FLUSH_FPU, [...].UEBERPREUFUNG_SW_INTR),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.EOI,UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.TLB_SHOOTDOWN),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.TLB_SHOOTDOWN, [...].UEBERPREUFUNG_SW_INTR),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.EOI, UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.SYNC_FPU),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.SYNC_FPU, [...].UEBERPREUFUNG_SW_INTR),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.EOI, UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.IPI_DB),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.IPI.IPI_DB, [...].UEBERPREUFUNG_SW_INTR),
([...].FRAME_ERSTELLEN, UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.EOI),
(UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.EOI, [...].VERSUCH_LOCK_BEKOMMEN),
(SPLX.PRUEF, UNTR_BEH.INTERRUPT_BEHANDLUNG.LAPIC_TIMER.TSC_AKT),
([...].LAPIC_TIMER.TSC_AKT, [...].HARDCLOCK.GRP_0.INTERVALZEITGEBER_ITIMER_VIRTUAL),
([...].INTERVALZEITGEBER_ITIMER_VIRTUAL, [...].HARDCLOCK.T_ADD_VIR),
([...].INTERVALZEITGEBER_ITIMER_VIRTUAL, [...].INTERVALZEITGEBER_ITIMER_PROF),
([...].HARDCLOCK.T_ADD_VIR, [...].INTERVALZEITGEBER_ITIMER_PROF),
([...].INTERVALZEITGEBER_ITIMER_PROF, [...].LAPIC_TIMER.HARDCLOCK.T_ADD_PROF),
([...].INTERVALZEITGEBER_ITIMER_PROF, [...].LAPIC_TIMER.HARDCLOCK.STAT_AKT ),
([...].LAPIC_TIMER.HARDCLOCK.T_ADD_PROF, [...].LAPIC_TIMER.HARDCLOCK.STAT_AKT ),
([...].LAPIC_TIMER.HARDCLOCK.STAT_AKT, [...].LAPIC_TIMER.HARDCLOCK.PRPRIO),
([...].LAPIC_TIMER.HARDCLOCK.PRPRIO, [...].LAPIC_TIMER.HARDCLOCK.RR_AKT),
([...].LAPIC_TIMER.HARDCLOCK.RR_AKT, [...].LOCK_ENTFERNEN),
([...].LAPIC_TIMER.HARDCLOCK.RR_AKT, [...].LAPIC_TIMER.HARDCLOCK.RESCHED),
([...].LAPIC_TIMER.HARDCLOCK.RESCHED, [...].LOCK_ENTFERNEN),
([...].LAPIC_TIMER.HARDCLOCK.RR_AKT, [...].LAPIC_TIMER.HARDCLOCK.SPCF_SEENRR),
([...].LAPIC_TIMER.HARDCLOCK.SPCF_SEENRR, [...].LAPIC_TIMER.HARDCLOCK.RESCHED),
([...].LAPIC_TIMER.HARDCLOCK.RESCHED, [...].LAPIC_TIMER.HARDCLOCK.T_W_AKT),
([...].LAPIC_TIMER.HARDCLOCK.T_W_AKT, [...].LOCK_ENTFERNEN),
([...].LAPIC_TIMER.HARDCLOCK.T_W_AKT, [...].LAPIC_TIMER.HARDCLOCK.SETSOFTCLOCK),
([...].LAPIC_TIMER.HARDCLOCK.SETSOFTCLOCK, [...].LOCK_ENTFERNEN),
([...].SW_INTR_NACH.UEBERPREUFUNG_SW_INTR, [...].SW_INTR_NACH.SW_B_L),
([...].SW_INTR_NACH.UEBERPREUFUNG_SW_INTR, FINAL),
([...].SW_INTR_NACH.SW_B_L, [...].SW_INTR_NACH.UEBERPREUFUNG_SW_INTR),
([...].SW_INTR_NACH.SW_B_L, [...].SW_INTR_NACH.CPL_SETZEN),
([...].SW_INTR_NACH.CPL_SETZEN, [...].VERSUCH_LOCK_BEKOMMEN),
(SPLX.PRUEF, [...].SW_INTR_NACH.UEBERPREUFUNG_SW_INTR)}

C_PAG_OPENSBD = {
([...].VERSUCH_LOCK_BEKOMMEN, [...].ZAEHLER_INKREMENTIEREN) -> RED
```

Since OpenBSD uses coarse-grained synchronization primitives affecting the IKCP, there is one red edge in the PAG. Table A.2 lists the all path length tuples for any interrupt that is handled by OpenBSD.

Disruptive Event	Handler state (v_{ana})	Min Length	Max Length
$DEVT_{x,INT,IPI,0}$	$\mathfrak{hdl}(DEVT_{x,INT,IPI,0}) =$ IPI_MICROSET	(4,0,0,0)	(10,4,11,1)
$DEVT_{x,INT,IPI,1}$	$\mathfrak{hdl}(DEVT_{x,INT,IPI,1}) =$ TLB_SHOOTDOWN	(4,0,0,0)	(10,4,11,1)
$DEVT_{x,INT,IPI,2}$	$\mathfrak{hdl}(DEVT_{x,INT,IPI,2}) =$ FLUSH_FPU	(4,0,0,0)	(10,4,11,1)
$DEVT_{x,INT,IPI,3}$	$\mathfrak{hdl}(DEVT_{x,INT,IPI,3}) =$ SYNCH_FPU	(4,0,0,0)	(10,4,11,1)
$DEVT_{x,INT,IPI,4}$	$\mathfrak{hdl}(DEVT_{x,INT,IPI,4}) =$ IPI_DB	(4,0,0,0)	(10,4,11,1)
$DEVT_{x,INT,IO,HDD}$	$\mathfrak{hdl}(DEVT_{x,INT,I/O,HDD}) =$ IOAPIC_HDL_0	(12,0,0,0)	(27,14,37,5)
$DEVT_{x,INT,IO,MPEG}$	$\mathfrak{hdl}(DEVT_{x,INT,I/O,MPEG}) =$ IOAPIC_HDL_1	(12,0,0,0)	(27,14,37,5)
$DEVT_{x,INT,IO,HID}$	$\mathfrak{hdl}(DEVT_{x,INT,I/O,HID}) =$ IOAPIC_HDL_2	(12,0,0,0)	(27,14,37,5)
$DEVT_{x,INT,IO,SND}$	$\mathfrak{hdl}(DEVT_{x,INT,I/O,SND}) =$ IOAPIC_HDL_3	(12,0,0,0)	(27,14,37,5)
$DEVT_{x,INT,IO,NIC}$	$\mathfrak{hdl}(DEVT_{x,INT,I/O,NIC}) =$ IOAPIC_HDL_4	(12,0,0,0)	(27,14,37,5)
$DEVT_{x,INT,IO,KB}$	$\mathfrak{hdl}(DEVT_{x,INT,I/O,KB}) =$ IOAPIC_HDL_5	(12,0,0,0)	(27,14,37,5)
$DEVT_{x,INT,TIMER}$	$\mathfrak{hdl}(DEVT_{x,INT,TIMER}) =$ LAPIC_TIMER	(13,0,0,0)	(34,10,26,4)

Table A.2: Path lengths of DEVTs in OpenBSD

A.3.3 Pistachio

Since Pistachio is a μ-kernel system, a kernel PAG and a user level PAG are constructed. The path lengths are then derived by simply adding the kernel and the user path lengths. Note that small letters from the original identifiers are replaced by capital ones and white spaces by _.

```
V_PAG_PISTACHIO_KERNEL = {START, KERNEL_LEVEL.IRQ.TIMER_HANDLER.SEND_EOI_APIC,
KERNEL_LEVEL.IRQ.TIMER_HANDLER.HANDLE_TIMER_INTERRUPT.UPDATE_GLOBAL_TIMER,
[...].HANDLE_TIMER_INTERRUPT.PROCESS_XCPU_MAILBOX.HANDLE_XCPU_REQUEST,
KERNEL_LEVEL.IRQ.TIMER_HANDLER.HANDLE_TIMER_INTERRUPT.SCHEDULER.UPDATE_TOTAL_QUANTUM,
[...].IRQ.TIMER_HANDLER.HANDLE_TIMER_INTERRUPT.SCHEDULER.TOTAL_QUANTUM_EXPIRED,
KERNEL_LEVEL.IRQ.TIMER_HANDLER.HANDLE_TIMER_INTERRUPT.SCHEDULER.END_OF_TIMESLICE,
KERNEL_LEVEL.IRQ.TIMER_HANDLER.HANDLE_TIMER_INTERRUPT.SCHEDULER.RESCHEDULE,
KERNEL_LEVEL.IRQ.TIMER_HANDLER.HANDLE_TIMER_INTERRUPT.SCHEDULER.WAKEUP_PREEMPTION,
KERNEL_LEVEL.IRQ.TIMER_HANDLER.HANDLE_TIMER_INTERRUPT.SCHEDULER.SWITCH_TO_WAKEUP,
KERNEL_LEVEL.IRQ.TIMER_HANDLER.IRET, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.SW_MASK,
KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.MARK_PENDING, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.EOI_1,
KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.EOI_2, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.IRET,
KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.HANDLE_HW_IRQ.DO_IRQTHREAD_IPC,
KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.HANDLE_HW_IRQ.DO_IPC,
KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.HANDLE_HW_IRQ.SEND_IRQ_TO_ANOTHER_CPU,
KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.HANDLE_HW_IRQ.SWITCH_TO_HANDLER,
KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.HANDLE_HW_IRQ.ENQUEUE_HANDLER,
KERNEL_LEVEL.IRQ.PROCESS_XCPU_MAILBOX.HANDLE_XCPU_REQUEST,FINAL}

E_PAG_PISTACHIO_KERNEL = {(START,KERNEL_LEVEL.IRQ.TIMER_HANDLER.SEND_EOI_APIC),
([...].TIMER_HANDLER.SEND_EOI_APIC, [...].PROCESS_XCPU_MAILBOX.HANDLE_XCPU_REQUEST),
([...].SEND_EOI_APIC,[...].HANDLE_TIMER_INTERRUPT.UPDATE_GLOBAL_TIMER),
([...].UPDATE_GLOBAL_TIMER,[...].PROCESS_XCPU_MAILBOX.HANDLE_XCPU_REQUEST),
([...].PROCESS_XCPU_MAILBOX.HANDLE_XCPU_REQUEST, KERNEL_LEVEL.IRQ.TIMER_HANDLER.IRET),
([...].PROCESS_XCPU_MAILBOX.HANDLE_XCPU_REQUEST, SCHEDULER.WAKEUP_PREEMPTION),
([...].SCHEDULER.WAKEUP_PREEMPTION, [...].SCHEDULER.SWITCH_TO_WAKEUP),
([...].SCHEDULER.SWITCH_TO_WAKEUP, KERNEL_LEVEL.IRQ.TIMER_HANDLER.IRET),
([...].PROCESS_XCPU_MAILBOX.HANDLE_XCPU_REQUEST, [...].SCHEDULER.UPDATE_TOTAL_QUANTUM),
([...].SCHEDULER.UPDATE_TOTAL_QUANTUM, [...].SCHEDULER.TOTAL_QUANTUM_EXPIRED),
([...].SCHEDULER.UPDATE_TOTAL_QUANTUM, [...].SCHEDULER.END_OF_TIMESLICE),
([...].SCHEDULER.TOTAL_QUANTUM_EXPIRED, [...].SCHEDULER.END_OF_TIMESLICE),
([...].PROCESS_XCPU_MAILBOX.HANDLE_XCPU_REQUEST, [...].SCHEDULER.END_OF_TIMESLICE),
([...].SCHEDULER.END_OF_TIMESLICE, [...].SCHEDULER.RESCHEDULE),
([...].SCHEDULER.RESCHEDULE, KERNEL_LEVEL.IRQ.TIMER_HANDLER.IRET),
(KERNEL_LEVEL.IRQ.TIMER_HANDLER.IRET, FINAL),
(START, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.SW_MASK),
(START, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.MARK_PENDING)
(START, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.EOI_2),
(KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.SW_MASK, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.EOI_1)
(KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.MARK_PENDING, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.EOI_2),
(KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.EOI_2, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.IRET),
([...].HW_IRQ_HANDLER.EOI_1, [...].HANDLE_HW_IRQ.DO_IRQTHREAD_IPC),
([...].HW_IRQ_HANDLER.EOI_1, [...].HANDLE_HW_IRQ.DO_IPC),
([...].HW_IRQ_HANDLER.EOI_1, [...].HANDLE_HW_IRQ.SEND_IRQ_TO_ANOTHER_CPU),
([...].HANDLE_HW_IRQ.DO_IRQTHREAD_IPC, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.IRET),
```

```
([...].HANDLE_HW_IRQ.SEND_IRQ_TO_ANOTHER_CPU, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.IRET),
([...].HANDLE_HW_IRQ.DO_IPC, [...].HANDLE_HW_IRQ.SWITCH_TO_HANDLER),
([...].HANDLE_HW_IRQ.DO_IPC, [...].HANDLE_HW_IRQ.ENQUEUE_HANDLER),
([...].HANDLE_HW_IRQ.SWITCH_TO_HANDLER, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.IRET),
([...].HANDLE_HW_IRQ.ENQUEUE_HANDLER, KERNEL_LEVEL.IRQ.HW_IRQ_HANDLER.IRET),
(START,KERNEL_LEVEL.IRQ.PROCESS_XCPU_MAILBOX.HANDLE_XCPU_REQUEST),
(KERNEL_LEVEL.IRQ.PROCESS_XCPU_MAILBOX.HANDLE_XCPU_REQUEST,FINAL)}

V_PAG_PISTACHIO_USER = {START, USER_LEVEL.USER_THREAD, USER_LEVEL.IDLE_THREAD,
USER_LEVEL.HW_IRQ_HANDLER_THREAD.HANDLE_IRQ, USER_LEVEL.HW_IRQ_HANDLER_THREAD,FINAL}

E_PAG_PISTACHIO_USER = {(START, USER_LEVEL.USER_THREAD),
(START, USER_LEVEL.IDLE_THREAD),
(START, USER_LEVEL.HW_IRQ_HANDLER_THREAD.HANDLE_IRQ),
(START, USER_LEVEL.HW_IRQ_HANDLER_THREAD),
(USER_LEVEL.USER_THREAD, FINAL),
(USER_LEVEL.IDLE_THREAD, FINAL),
(USER_LEVEL.HW_IRQ_HANDLER_THREAD.HANDLE_IRQ, FINAL),
(USER_LEVEL.HW_IRQ_HANDLER_THREAD, FINAL),}
```

Table A.3 lists the accumulated path lengths for the μ-kernel operating system Pistachio.

Disruptive Event	Handler state (v_{ana})	Min Length	Max Length
$DEVT_{x,INT,IPI}$	$\mathfrak{hdl}(DEVT_{x,INT,IPI})$ = PROCESS_XCPU_MAILBOX	(2,0,0,0)	(7,0,0,0)
$DEVT_{x,INT,IO,HDD}$	$\mathfrak{hdl} * (DEVT_{x,INT,I/O,HDD}) \supset$ HLD_HW_IRQ HANDLE_IRQ	(2,0,0,0) + (1,0,0,0)	(7,0,0,0) + (1,0,0,0)
$DEVT_{x,INT,IO,MPEG}$	$\mathfrak{hdl} * (DEVT_{x,INT,I/O,HDD}) \supset$ HLD_HW_IRQ HANDLE_IRQ	(2,0,0,0) + (1,0,0,0)	(7,0,0,0) + (1,0,0,0)
$DEVT_{x,INT,IO,HID}$	$\mathfrak{hdl} * (DEVT_{x,INT,I/O,HDD}) \supset$ HLD_HW_IRQ HANDLE_IRQ	(2,0,0,0) + (1,0,0,0)	(7,0,0,0) + (1,0,0,0)
$DEVT_{x,INT,IO,SND}$	$\mathfrak{hdl} * (DEVT_{x,INT,I/O,HDD}) \supset$ HLD_HW_IRQ HANDLE_IRQ	(2,0,0,0) + (1,0,0,0)	(7,0,0,0) + (1,0,0,0)
$DEVT_{x,INT,IO,NIC}$	$\mathfrak{hdl} * (DEVT_{x,INT,I/O,HDD}) \supset$ HLD_HW_IRQ HANDLE_IRQ	(2,0,0,0) + (1,0,0,0)	(7,0,0,0) + (1,0,0,0)
$DEVT_{x,INT,IO,KB}$	$\mathfrak{hdl} * (DEVT_{x,INT,I/O,HDD}) \supset$ HLD_HW_IRQ HANDLE_IRQ	(2,0,0,0) + (1,0,0,0)	(7,0,0,0) + (1,0,0,0)
$DEVT_{x,INT,TIMER}$	$\mathfrak{hdl}(DEVT_{x,INT,TIMER})$ = TIMER_HANDLER	(3,0,0,0)	(7,0,0,0)

Table A.3: Path lengths of DEVTs in Pistachio

A.4 Interruptibility

Based on the three path analysis graphs in Section A.3, the IAGs and the two multisets $Int(path)$ and $Inf(e)$ are elaborated in this section. Since all three SuIs use a slightly different categorization of disruptive events (cp. Section 4.4), the $\mathfrak{wgt}$-function is also detailed on a per-SuI level.

A.4.1 Linux

The concrete $\mathfrak{wgt}$ function for the Linux PAG and the set of disruptive events as handled in Linux is the following:

$\mathfrak{wgt}(DEVT_{x,INT,IPI,0}) \stackrel{\text{def}}{=} 2$
$\mathfrak{wgt}(DEVT_{x,INT,IPI,1}) \stackrel{\text{def}}{=} 3$
$\mathfrak{wgt}(DEVT_{x,INT,IPI,2}) \stackrel{\text{def}}{=} 4$
$\mathfrak{wgt}(DEVT_{x,INT,IO,HDD}) \stackrel{\text{def}}{=} 5$
$\mathfrak{wgt}(DEVT_{x,INT,IO,MPEG}) \stackrel{\text{def}}{=} 6$
$\mathfrak{wgt}(DEVT_{x,INT,IO,HID}) \stackrel{\text{def}}{=} 7$
$\mathfrak{wgt}(DEVT_{x,INT,IO,SND}) \stackrel{\text{def}}{=} 8$
$\mathfrak{wgt}(DEVT_{x,INT,IO,NIC}) \stackrel{\text{def}}{=} 9$
$\mathfrak{wgt}(DEVT_{x,INT,IO,KB}) \stackrel{\text{def}}{=} 10$
$\mathfrak{wgt}(x, INT, LOC\_TIMER) \stackrel{\text{def}}{=} 11$
$\mathfrak{wgt}(DEVT_{x,INT,GLOB\_TIMER}) \stackrel{\text{def}}{=} 12$

For the sake of brevity, a summed-up notation is given for the set of green edges and the weighting of an edge: `((PAG_STATE, DUMMY), C = { x,y })` means that there are two green edges from `STATE` to `DUMMY`, one with the weight `x` and one with `y`:

```
{(IKCP.EXC.HD.CHECK_CTX, DUMMY),         C = {2,3,4,5,6,7,8,9,10,11,12}),
(IKCP.EXC.HD.FIXUP, DUMMY),              C = {2,3,4,5,6,7,8,9,10,11,12}),
(IKCP.EXC.HD.GEN_SIG, DUMMY),            C = {2,3,4,5,6,7,8,9,10,11,12}),
(IKCP.INT.IPI.DO_RESCHED, DUMMY),        C = {  3,4,5,6,7,8,9,10,11,12}),
(IKCP.INT.IPI.DO_CALL_FUN, DUMMY),       C = {2,  4,5,6,7,8,9,10,11,12}),
(IKCP.INT.IPI.DO_INV_TLB, DUMMY),        C = {2,3,  5,6,7,8,9,10,11,12}),
(IKCP.INT.IO_ISR.DISRUPT_0, DUMMY),      C = {2,3,4,  6,7,8,9,10,11,12}),
(IKCP.INT.IO_ISR.ACT_TASK_0, DUMMY),     C = {2,3,4,  6,7,8,9,10,11,12}),
(IKCP.INT.IO_ISR.DISRUPT_1, DUMMY),      C = {2,3,4,5,  7,8,9,10,11,12}),
(IKCP.INT.IO_ISR.ACT_TASK_1, DUMMY),     C = {2,3,4,5,  7,8,9,10,11,12}),
(IKCP.INT.IO_ISR.DISRUPT_2, DUMMY),      C = {2,3,4,5,6,  8,9,10,11,12}),
(IKCP.INT.IO_ISR.ACT_TASK_2, DUMMY),     C = {2,3,4,5,6,  8,9,10,11,12}),
(IKCP.INT.IO_ISR.DISRUPT_3, DUMMY),      C = {2,3,4,5,6,7,  9,10,11,12}),
(IKCP.INT.IO_ISR.ACT_TASK_3, DUMMY),     C = {2,3,4,5,6,7,  9,10,11,12}),
(IKCP.INT.IO_ISR.DISRUPT_4, DUMMY),      C = {2,3,4,5,6,7,8,  10,11,12}),
(IKCP.INT.IO_ISR.ACT_TASK_4, DUMMY),     C = {2,3,4,5,6,7,8,  10,11,12}),
(IKCP.INT.IO_ISR.DISRUPT_5, DUMMY),      C = {2,3,4,5,6,7,8,9,   11,12}),
(IKCP.INT.IO_ISR.ACT_TASK_5, DUMMY),     C = {2,3,4,5,6,7,8,9,   11,12}),
(IKCP.INT.SOFT.CHECK_SOFTIRQ, DUMMY),    C = {2,3,4,5,6,7,8,9,10,11,12}),
(IKCP.INT.SOFT.RUN_SOFTIRQ, DUMMY),      C = {2,3,4,5,6,7,8,9,10,11,12})}
```

The two multisets $Int(path)$ and $Inf(e)$ for Linux are now easily derived from the IAG as a 11×11 matrix. Each row represents the path according to the 11 disruptive path events (weights 2-12), each column represents one of the 11 disruptive path events themselves. The matrix elements then show how often a certain disruption can occur.

$$\Gamma_{Linux} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 3 & 3 & 3 & 3 & 3 & 3 & 2 & 3 & 3 & 3 & 3 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

A.4.2 OpenBSD

The concrete $\mathfrak{wgt}$ function for the OpenBSD IAG slightly differs from the one defined for Linux:

$\mathfrak{wgt}(DEVT_{x,INT,IPI,0}) \stackrel{\text{def}}{=} 2$ $\quad$ $\mathfrak{wgt}(DEVT_{x,INT,IPI,1}) \stackrel{\text{def}}{=} 3$

$\mathfrak{wgt}(DEVT_{x,INT,IPI,2}) \stackrel{\text{def}}{=} 4$ $\quad$ $\mathfrak{wgt}(DEVT_{x,INT,IPI,3}) \stackrel{\text{def}}{=} 5$

$\mathfrak{wgt}(DEVT_{x,INT,IPI,4}) \stackrel{\text{def}}{=} 6$ $\quad$ $\mathfrak{wgt}(DEVT_{x,INT,IO,HDD}) \stackrel{\text{def}}{=} 7$

$\mathfrak{wgt}(DEVT_{x,INT,IO,MPEG}) \stackrel{\text{def}}{=} 8$ $\quad$ $\mathfrak{wgt}(DEVT_{x,INT,IO,HID}) \stackrel{\text{def}}{=} 9$

$\mathfrak{wgt}(DEVT_{x,INT,IO,SND}) \stackrel{\text{def}}{=} 10$ $\quad$ $\mathfrak{wgt}(DEVT_{x,INT,IO,NIC}) \stackrel{\text{def}}{=} 11$

$\mathfrak{wgt}(DEVT_{x,INT,IO,KB}) \stackrel{\text{def}}{=} 12$ $\quad$ $\mathfrak{wgt}(x, INT, TIMER) \stackrel{\text{def}}{=} 13$

The set of green edges and their weights in OpenBSD are defined as follows. The same notation as for the Linux IAG is used.

```
{(UNTR_BEH.SYSTEMAUFRUF.SYS_VOR, DUMMY),     C = {2,3,4,5,6,7,8,9,10,11,12,13}),
(UNTR_BEH.SYSTEMAUFRUF.SYS_NACH, DUMMY),     C = {2,3,4,5,6,7,8,9,10,11,12,13}),
(UNTR_BEH.SYSTEMAUFRUF.USERRET, DUMMY),      C = {2,3,4,5,6,7,8,9,10,11,12,13}),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.DNA, DUMMY),  C = {2,3,4,5,6,7,8,9,10,11,12,13}),
(UNTR_BEH.EXCEPTION_BEHANDLUNG.FPU, DUMMY),  C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].TRAP.USR, DUMMY),                     C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].TRAP.PAGE, DUMMY),                    C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].TRAP.SPEZIAL, DUMMY),                 C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].TRAP.TRAPSIGNAL, DUMMY),              C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].TRAP.PRUEFE, DUMMY),                  C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].TRAP.USERRET , DUMMY),                C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].INTERRUPT_HANDLER_ERMITTELN , DUMMY), C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].STRAY_INTERRUPT , DUMMY),             C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].NAECHSTER_INTERRUPT_HANDLER , DUMMY), C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].STATISTIKEN_AKTUALISIEREN , DUMMY),   C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].IO_APIC_HDL_0,  DUMMY),               C = {2,3,4,5,6,  8,9,10,11,12,13}),
([...].IO_APIC_HDL_1,  DUMMY),               C = {2,3,4,5,6,7,  9,10,11,12,13}),
([...].IO_APIC_HDL_2,  DUMMY),               C = {2,3,4,5,6,7,8,  10,11,12,13}),
([...].IO_APIC_HDL_3,  DUMMY),               C = {2,3,4,5,6,7,8,9,   11,12,13}),
([...].IO_APIC_HDL_4,  DUMMY),               C = {2,3,4,5,6,7,8,9,10,   12,13}),
([...].IO_APIC_HDL_5,  DUMMY),               C = {2,3,4,5,6,7,8,9,10,11,   13}),
```

```
([...].IPI.FLUSH_FPU, DUMMY),                C = {2,3,4,5,6                  }),
([...].IPI.TLB_SHOOTDOWN, DUMMY),            C = {2,3,4,5,6                  }),
([...].IPI.SYNC_FPU, DUMMY),                 C = {2,3,4,5,6                  }),
([...].IPI.IPI_DB, DUMMY),                   C = {2,3,4,5,6                  }),
([...].LAPIC_TIMER.TSC_AKT, DUMMY),          C = {2,3,4,5,6,                13}),
([...].ZEITGEBER_ITIMER_VIRTUAL, DUMMY),     C = {2,3,4,5,6,                13}),
([...].ZEITGEBER_ITIMER_PROF, DUMMY),        C = {2,3,4,5,6,                13}),
([...].HARDCLOCK.T_ADD_VIR, DUMMY),          C = {2,3,4,5,6,                13}),
([...].HARDCLOCK.T_ADD_PROF, DUMMY),         C = {2,3,4,5,6,                13}),
([...].HARDCLOCK.STAT_AKT, DUMMY),           C = {2,3,4,5,6,                13}),
([...].HARDCLOCK.PRPRIO, DUMMY),             C = {2,3,4,5,6,                13}),
([...].HARDCLOCK.SPCF_SEENRR, DUMMY),        C = {2,3,4,5,6,                13}),
([...].HARDCLOCK.RESCHED, DUMMY),            C = {2,3,4,5,6,                13}),
([...].HARDCLOCK.T_W_AKT, DUMMY),            C = {2,3,4,5,6,                13}),
([...].HARDCLOCK.SETSOFTCLOCK, DUMMY),       C = {2,3,4,5,6,                13}),
([...].SW_INTR_NACH.SW_B_L, DUMMY),          C = {2,3,4,5,6,7,8,9,10,11,12,13}),
(SPLX.CPL_SETZEN, DUMMY),                    C = {2,3,4,5,6,7,8,9,10,11,12,13}),
(SPLX.PRUE, DUMMY),                          C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].INTERRUPT_IPI.CPL_SETZEN, DUMMY),     C = {2,3,4,5,6,7,8,9,10,11,12,13}),
([...].SOFTWARE_INTERRUPT_IPI.DEREG, DUMMY), C = {2,3,4,5,6,7,8,9,10,11,12,13}),
(SOFTWARE_INTERRUPT.TTY_BEH., DUMMY),        C = {2,3,4,5,6,7,8,9,10,11,12,13}),
(LOCK_SETZEN.VERSUCH_LOCK_BEKOMMEN, DUMMY),  C = {2,3,4,5,6,7,8,9,10,11,12,13}),
(LOCK_SETZEN.ZAEHLER_INKREMENTIEREN, DUMMY), C = {2,3,4,5,6,7,8,9,10,11,12,13}),
(ENTFERNEN.ZAEHLER_DEKREMENTIEREN, DUMMY),   C = {2,3,4,5,6,7,8,9,10,11,12,13}),
(ENTFERNEN.LOCK_ENTFERNEN, DUMMY),           C = {2,3,4,5,6,7,8,9,10,11,12,13})}
```

With the above IAG and the set of green edges, the interruptibility matrix as given for Linux is for OpenBSD as follows:

$$\Gamma_{OpenBSD} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 6 & 6 & 6 & 6 & 5 & 6 & 6 & 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 & 6 & 6 & 5 & 6 & 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 5 & 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 5 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 5 & 6 & 6 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 5 & 6 \\ 11 & 11 & 11 & 11 & 11 & 11 & 0 & 0 & 0 & 0 & 0 & 11 \end{pmatrix}$$

A.4.3 Pistachio

The weighting for the edges representing the interruptible parts of the Pistachio operating system is given as follows:

$\mathfrak{wgt}(DEVT_{x,INT,IPI}) \stackrel{\text{def}}{=} 2$ $\quad$ $\mathfrak{wgt}(DEVT_{x,INT,IO,HDD}) \stackrel{\text{def}}{=} 3$

$\mathfrak{wgt}(DEVT_{x,INT,IO,MPEG}) \stackrel{\text{def}}{=} 4$ $\quad$ $\mathfrak{wgt}(DEVT_{x,INT,IO,HID}) \stackrel{\text{def}}{=} 5$

$\mathfrak{wgt}(DEVT_{x,INT,IO,SND}) \stackrel{\text{def}}{=} 6$ $\quad$ $\mathfrak{wgt}(DEVT_{x,INT,IO,NIC}) \stackrel{\text{def}}{=} 7$

$\mathfrak{wgt}(DEVT_{x,INT,IO,KB}) \stackrel{\text{def}}{=} 8$ $\quad$ $\mathfrak{wgt}(x, INT, TIMER) \stackrel{\text{def}}{=} 9$

The set of green edges and their weights for both parts, i.e. the kernel and user level graphs are listed now for Pistachio.

```
{([...].IRQ.TIMER_HANDLER.UPDATE_GLOBAL_TIMER, DUMMY),   C = {2,3,4,5,6,7,8  }),
{([...].PROCESS_XCPU_MAILBOX.HANDLE_XCPU_REQUEST, DUMMY),C = {2,3,4,5,6,7,8  }),
{([...].SCHEDULER.UPDATE_TOTAL_QUANTUM, DUMMY),          C = {2,3,4,5,6,7,8  }),
{([...].SCHEDULER.TOTAL_QUANTUM_EXPIRED, DUMMY),         C = {2,3,4,5,6,7,8  }),
{([...].SCHEDULER.END_OF_TIMESLICE, DUMMY),              C = {2,3,4,5,6,7,8  }),
{([...].SCHEDULER.RESCHEDULE, DUMMY),                    C = {2,3,4,5,6,7,8  }),
{([...].SCHEDULER.WAKEUP_PREEMPTION, DUMMY),             C = {2,3,4,5,6,7,8  }),
{([...].SWITCH_TO_WAKEUP, DUMMY),                        C = {2,3,4,5,6,7,8  }),
{([...].HANDLE_HW_IRQ.DO_IRQTHREAD_IPC, DUMMY),          C = {2,3,4,5,6,7,8,9}),
{([...].SEND_IRQ_TO_ANOTHER_CPU, DUMMY),                 C = {2,3,4,5,6,7,8,9}),
{([...].DO_IPC, DUMMY),                                  C = {2,3,4,5,6,7,8,9}),
{([...].SWITCH_TO_HANDLER, DUMMY),                       C = {2,3,4,5,6,7,8,9}),
{([...].ENQUEUE_HANDLER, DUMMY),                         C = {2,3,4,5,6,7,8,9}),
{(USER_LEVEL.HW_IRQ_HANDLER_THREAD_0, DUMMY),            C = {2,3,4,5,6,7,8,9}),
{(USER_LEVEL.HW_IRQ_HANDLER_THREAD_1, DUMMY),            C = {2,3,4,5,6,7,8,9}),
{(USER_LEVEL.HW_IRQ_HANDLER_THREAD_2, DUMMY),            C = {2,3,4,5,6,7,8,9}),
{(USER_LEVEL.HW_IRQ_HANDLER_THREAD_3, DUMMY),            C = {2,3,4,5,6,7,8,9}),
{(USER_LEVEL.HW_IRQ_HANDLER_THREAD_4, DUMMY),            C = {2,3,4,5,6,7,8,9}),
{(USER_LEVEL.HW_IRQ_HANDLER_THREAD_5, DUMMY),            C = {2,3,4,5,6,7,8,9})}
```

The resulting interruptibility matrix for Pistachio is given as follows:

$$\Gamma_{Pistachio} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\ 8 & 8 & 8 & 8 & 8 & 8 & 8 & 0 \end{pmatrix}$$

Appendix B

SMV Models

B.1 Linux

```
MODULE main
VAR
  state : {
    -- dummy user-space state
    in_USERSPACE,
    -- scheduling state
    in_SCHEDULER,
    -- INT states
    in_INT-CPU_HANDLING, in_INT-SAVE_REGISTERS, in_INT-RETURN_INT, in_INT-CPU_RETURN,
    -- helper state to re-unite paths before decision SoftIRQ handling or not
    -- necessary since combine is directly followed by a branch connector
    in_INT-CO3,
    -- INT.IPI states
    in_INT-IPI-SAVE_GPR, in_INT-IPI-DO_RESCHED, in_INT-IPI-DO_CALL_FUN, in_INT-IPI-DO_INV_TLB,
    -- INT.LT_ISR states
    in_INT-LT_ISR-DO_INT, in_INT-LT_ISR-PROF_TICK, in_INT-LT_ISR-UPD_PROC_TIMES-ACCOUNTING-UPD_TIME,
    in_INT-LT_ISR-UPD_PROC_TIMES-ACCOUNTING-SENDSIG, in_INT-LT_ISR-UPD_PROC_TIMES-ACCOUNTING-IT_TIMER,
    in_INT-LT_ISR-UPD_PROC_TIMES-TIMER_SOFTIRQ, in_INT-LT_ISR-UPD_PROC_TIMES-TICK-BALANCE,
    in_INT-LT_ISR-UPD_PROC_TIMES-TICK-SET_RESCHED, in_INT-LT_ISR-EXIT_INT,
    -- INT.GT_ISR states
    in_INT-GT_ISR-DO_INT, in_INT-GT_ISR-CHECK_LOST, in_INT-GT_ISR-JIFFIES, in_INT-GT_ISR-UPD_TIME,
    in_INT-GT_ISR-EXIT_INT,
    -- INT.IO_ISR states
    in_INT-IO_ISR-DO_INT, in_INT-IO_ISR-UNDISRUPT_0, in_INT-IO_ISR-DISRUPT_ACT_TASK_0,
    in_INT-IO_ISR-UNDISRUPT_5, in_INT-IO_ISR-DISRUPT_ACT_TASK_5, in_INT-IO_ISR-EXIT_INT,
    -- INT.SOFT states
    in_INT-SOFT-DISABLE_DEF_FUN, in_INT-SOFT-CHECK_RUN_SOFTIRQ, in_INT-SOFT-SIG_KSOFTIRQD};

    -- path events
    pe : {pe_none, pe_INT_IPI_0, pe_INT_IPI_1, pe_INT_IPI_2, pe_INT_LOC_TIMER,
          pe_INT_GLOB_TIMER, pe_INT_IO_0, pe_INT_IO_5};

    -- counter to access the path conditions array statically
    counter : 0..6;
    pcs: array 0..6 of {pc_none, pc_INT_IPI_0, pc_INT_IPI_1, pc_INT_IPI_2, pc_INT_LOC_TIMER,
        pc_INT_GLOB_TIMER, pc_INT_IO_0, pc_INT_IO_5};

    -- current and previous path condition : constructed to avoid non-allowed dynamic access to pcs array
    pc: {pc_none, pc_INT_IPI_0, pc_INT_IPI_1, pc_INT_IPI_2, pc_INT_LOC_TIMER,
         pc_INT_GLOB_TIMER, pc_INT_IO_0, pc_INT_IO_5};
    prevpc : {pc_none, pc_INT_IPI_0, pc_INT_IPI_1, pc_INT_IPI_2, pc_INT_LOC_TIMER, pc_INT_GLOB_TIMER,
              pc_INT_IO_0, pc_INT_IO_5};

    -- last interrupted historic state
```

```
    hist : {
        state_none,
        -- INT.IPI states
        in_INT-IPI-DO_RESCHED, in_INT-IPI-DO_CALL_FUN, in_INT-IPI-DO_INV_TLB,
        -- INT.IO_ISR states
        in_INT-IO_ISR-DISRUPT_ACT_TASK_0,
        in_INT-IO_ISR-DISRUPT_ACT_TASK_5
        };

    -- indicator: has SOFT submachine been interrupted in the last run?
    submachine_soft_interrupted : boolean;

    -- condition variables
    soft_int_pending : boolean;
    tif_need_reschedule : boolean;


ASSIGN

  --
  -- \DELTA P_CONF : STATES / BASIC CONFIGURATIONS
  -- START
  --

  init(state) := in_INT-CPU_HANDLING;
  next(state) := case

    -- INT states
    state = in_INT-CPU_HANDLING : in_INT-SAVE_REGISTERS;
    state = in_INT-RETURN_INT : in_INT-CPU_RETURN;

    -- split state SP3
    state = in_INT-SAVE_REGISTERS & (pc = pc_INT_IPI_0 | pc = pc_INT_IPI_1 | pc = pc_INT_IPI_2):
            in_INT-IPI-SAVE_GPR;
    state = in_INT-SAVE_REGISTERS & (pc = pc_INT_LOC_TIMER): in_INT-LT_ISR-DO_INT;
    state = in_INT-SAVE_REGISTERS & (pc = pc_INT_GLOB_TIMER): in_INT-GT_ISR-DO_INT;
    state = in_INT-SAVE_REGISTERS & (pc = pc_INT_IO_0): in_INT-IO_ISR-DO_INT;
    state = in_INT-SAVE_REGISTERS & (pc = pc_INT_IO_5): in_INT-IO_ISR-DO_INT;

    -- resume in SOFT when was interrupted, else start in default state of SOFT
    state = in_INT-CO3 & soft_int_pending > 0 & submachine_soft_interrupted = 0:
      in_INT-SOFT-DISABLE_DEF_FUN;
    state = in_INT-CO3 & soft_int_pending > 0 & submachine_soft_interrupted = 1:
      in_INT-SOFT-CHECK_RUN_SOFTIRQ;

    state = in_INT-CO3 & soft_int_pending = 0 : in_INT-RETURN_INT;
    state = in_INT-RETURN_INT : in_INT-CPU_RETURN;

    -- history : jumping back to last left state
    state = in_INT-CPU_RETURN & hist != state_none  : hist;

    -- INT processed : go to scheduler and user space
    state = in_INT-CPU_RETURN & hist = state_none : in_SCHEDULER;
    state = in_SCHEDULER : in_USERSPACE;


    --
    -- Submachine IPI - START

    -- ... <- analogous for target states in_INT-IPI-DO_CALL_FUN and in_INT-IPI-DO_INV_TLB
    state = in_INT-IPI-SAVE_GPR & pc = pc_INT_IPI_0 : in_INT-IPI-DO_RESCHED;

    state = in_INT-IPI-DO_RESCHED & (pe = pe_none | pe = pe_INT_IPI_0 |
      (pe = pe_INT_IPI_1 & (pcs[0] = pc_INT_IPI_1 | pcs[1] = pc_INT_IPI_1 | pcs[2] = pc_INT_IPI_1 |
            pcs[3] = pc_INT_IPI_1 | pcs[4] = pc_INT_IPI_1 | pcs[5] = pc_INT_IPI_1 | pcs[6] = pc_INT_IPI_1)) |
```

```
(pe = pe_INT_IPI_2 & (pcs[0] = pc_INT_IPI_2 | pcs[1] = pc_INT_IPI_2 | pcs[2] = pc_INT_IPI_2 |
      pcs[3] = pc_INT_IPI_2 | pcs[4] = pc_INT_IPI_2 | pcs[5] = pc_INT_IPI_2 | pcs[6] = pc_INT_IPI_2)) |
(pe = pe_INT_LOC_TIMER & (pcs[0] = pc_INT_LOC_TIMER | pcs[1] = pc_INT_LOC_TIMER |
      pcs[2] = pc_INT_LOC_TIMER | pcs[3] = pc_INT_LOC_TIMER | pcs[4] = pc_INT_LOC_TIMER |
      pcs[5] = pc_INT_LOC_TIMER | pcs[6] = pc_INT_LOC_TIMER)) |
(pe = pe_INT_GLOB_TIMER & (pcs[0] = pc_INT_GLOB_TIMER | pcs[1] = pc_INT_GLOB_TIMER |
      pcs[2] = pc_INT_GLOB_TIMER | pcs[3] = pc_INT_GLOB_TIMER | pcs[4] = pc_INT_GLOB_TIMER |
      pcs[5] = pc_INT_GLOB_TIMER | pcs[6] = pc_INT_GLOB_TIMER)) |
(pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
      pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5 | pcs[5] = pc_INT_IO_5 | pcs[6] = pc_INT_IO_5)) |
(pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
      pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0 | pcs[5] = pc_INT_IO_0 | pcs[6] = pc_INT_IO_0)))
: in_INT-CO3;
-- ... ->

-- <- analogous for target states in_INT-IPI-DO_CALL_FUN and in_INT-IPI-DO_INV_TLB
-- path event handling : stepping to next recursion level
state = in_INT-IPI-DO_RESCHED & (pe = pe_INT_IPI_1 & !(pcs[0] = pc_INT_IPI_1 | pcs[1] = pc_INT_IPI_1 |
        pcs[2] = pc_INT_IPI_1 | pcs[3] = pc_INT_IPI_1 | pcs[4] = pc_INT_IPI_1 | pcs[5] = pc_INT_IPI_1 |
        pcs[6] = pc_INT_IPI_1)) : in_INT-CPU_HANDLING;
state = in_INT-IPI-DO_RESCHED & (pe = pe_INT_IPI_2 & !(pcs[0] = pc_INT_IPI_2 | pcs[1] = pc_INT_IPI_2 |
        pcs[2] = pc_INT_IPI_2 | pcs[3] = pc_INT_IPI_2 | pcs[4] = pc_INT_IPI_2 | pcs[5] = pc_INT_IPI_2 |
        pcs[6] = pc_INT_IPI_2)) : in_INT-CPU_HANDLING;
state = in_INT-IPI-DO_RESCHED & (pe = pe_INT_LOC_TIMER & !(pcs[0] = pc_INT_LOC_TIMER |
        pcs[1] = pc_INT_LOC_TIMER | pcs[2] = pc_INT_LOC_TIMER | pcs[3] = pc_INT_LOC_TIMER |
        pcs[4] = pc_INT_LOC_TIMER | pcs[5] = pc_INT_LOC_TIMER | pcs[6] = pc_INT_LOC_TIMER))
        : in_INT-CPU_HANDLING;
state = in_INT-IPI-DO_RESCHED & (pe = pe_INT_GLOB_TIMER & !(pcs[0] = pc_INT_GLOB_TIMER |
        pcs[1] = pc_INT_GLOB_TIMER | pcs[2] = pc_INT_GLOB_TIMER | pcs[3] = pc_INT_GLOB_TIMER |
        pcs[4] = pc_INT_GLOB_TIMER | pcs[5] = pc_INT_GLOB_TIMER | pcs[6] = pc_INT_GLOB_TIMER))
        : in_INT-CPU_HANDLING;
state = in_INT-IPI-DO_RESCHED & (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 |
        pcs[2] = pc_INT_IO_5 | pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5 | pcs[5] = pc_INT_IO_5 |
        pcs[6] = pc_INT_IO_5)) : in_INT-CPU_HANDLING;
state = in_INT-IPI-DO_RESCHED & (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 |
        pcs[2] = pc_INT_IO_0 | pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0 | pcs[5] = pc_INT_IO_0 |
        pcs[6] = pc_INT_IO_0)) : in_INT-CPU_HANDLING;
-- ... ->

-- Submachine IPI - END
--

--
-- Submachine LT_ISR - START
state = in_INT-LT_ISR-DO_INT : in_INT-LT_ISR-PROF_TICK;

state = in_INT-LT_ISR-PROF_TICK : in_INT-LT_ISR-UPD_PROC_TIMES-ACCOUNTING-UPD_TIME;
state = in_INT-LT_ISR-UPD_PROC_TIMES-ACCOUNTING-UPD_TIME :
  in_INT-LT_ISR-UPD_PROC_TIMES-ACCOUNTING-SENDSIG;
state = in_INT-LT_ISR-UPD_PROC_TIMES-ACCOUNTING-SENDSIG :
  in_INT-LT_ISR-UPD_PROC_TIMES-ACCOUNTING-IT_TIMER;
state = in_INT-LT_ISR-UPD_PROC_TIMES-ACCOUNTING-IT_TIMER :
  in_INT-LT_ISR-UPD_PROC_TIMES-TIMER_SOFTIRQ;
state = in_INT-LT_ISR-UPD_PROC_TIMES-TIMER_SOFTIRQ :
  in_INT-LT_ISR-UPD_PROC_TIMES-TICK-BALANCE;
state = in_INT-LT_ISR-UPD_PROC_TIMES-TICK-BALANCE :
  in_INT-LT_ISR-UPD_PROC_TIMES-TICK-SET_RESCHED;
state = in_INT-LT_ISR-UPD_PROC_TIMES-TICK-SET_RESCHED : in_INT-LT_ISR-EXIT_INT;
state = in_INT-LT_ISR-EXIT_INT : in_INT-CO3;
-- Submachine LT_ISR - END
--

--
-- Submachine GT_ISR - START
state = in_INT-GT_ISR-DO_INT : in_INT-GT_ISR-CHECK_LOST;
```

```
state = in_INT-GT_ISR-CHECK_LOST : in_INT-GT_ISR-JIFFIES;
state = in_INT-GT_ISR-JIFFIES : in_INT-GT_ISR-UPD_TIME;
state = in_INT-GT_ISR-UPD_TIME : in_INT-GT_ISR-EXIT_INT;
state = in_INT-GT_ISR-EXIT_INT : in_INT-CO3;

-- Submachine GT_ISR - END
--

--
-- Submachine IO_ISR - START
-- split state SP5
state = in_INT-IO_ISR-DO_INT & pc = pc_INT_IO_0 : in_INT-IO_ISR-UNDISRUPT_0;
state = in_INT-IO_ISR-DO_INT & pc = pc_INT_IO_5 : in_INT-IO_ISR-UNDISRUPT_5;
state = in_INT-IO_ISR-UNDISRUPT_0 : in_INT-IO_ISR-DISRUPT_ACT_TASK_0;
state = in_INT-IO_ISR-UNDISRUPT_5 : in_INT-IO_ISR-DISRUPT_ACT_TASK_5;

-- ... <- analogous for in_INT-IO_ISR-DISRUPT_ACT_TASK_5
state = in_INT-IO_ISR-DISRUPT_ACT_TASK_0 & (pe = pe_none | pe = pe_INT_IO_0 |
  (pe = pe_INT_IPI_0 & (pcs[0] = pc_INT_IPI_0 | pcs[1] = pc_INT_IPI_0 | pcs[2] = pc_INT_IPI_0 |
  pcs[3] = pc_INT_IPI_0 | pcs[4] = pc_INT_IPI_0 | pcs[5] = pc_INT_IPI_0 | pcs[6] = pc_INT_IPI_0)) |
  (pe = pe_INT_IPI_1 & (pcs[0] = pc_INT_IPI_1 | pcs[1] = pc_INT_IPI_1 | pcs[2] = pc_INT_IPI_1 |
  pcs[3] = pc_INT_IPI_1 | pcs[4] = pc_INT_IPI_1 | pcs[5] = pc_INT_IPI_1 | pcs[6] = pc_INT_IPI_1)) |
  (pe = pe_INT_IPI_2 & (pcs[0] = pc_INT_IPI_2 | pcs[1] = pc_INT_IPI_2 | pcs[2] = pc_INT_IPI_2 |
  pcs[3] = pc_INT_IPI_2 | pcs[4] = pc_INT_IPI_2 | pcs[5] = pc_INT_IPI_2 | pcs[6] = pc_INT_IPI_2)) |
  (pe = pe_INT_LOC_TIMER & (pcs[0] = pc_INT_LOC_TIMER | pcs[1] = pc_INT_LOC_TIMER |
  pcs[2] = pc_INT_LOC_TIMER | pcs[3] = pc_INT_LOC_TIMER | pcs[4] = pc_INT_LOC_TIMER |
  pcs[5] = pc_INT_LOC_TIMER | pcs[6] = pc_INT_LOC_TIMER)) |
  (pe = pe_INT_GLOB_TIMER & (pcs[0] = pc_INT_GLOB_TIMER | pcs[1] = pc_INT_GLOB_TIMER |
  pcs[2] = pc_INT_GLOB_TIMER | pcs[3] = pc_INT_GLOB_TIMER | pcs[4] = pc_INT_GLOB_TIMER |
  pcs[5] = pc_INT_GLOB_TIMER | pcs[6] = pc_INT_GLOB_TIMER)) |
  (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5 | pcs[5] = pc_INT_IO_5 | pcs[6] = pc_INT_IO_5)))
  : in_INT-IO_ISR-EXIT_INT;

state = in_INT-IO_ISR-DISRUPT_ACT_TASK_0 & (pe = pe_INT_IPI_0 & !(pcs[0] = pc_INT_IPI_0 |
  pcs[1] = pc_INT_IPI_0 | pcs[2] = pc_INT_IPI_0 | pcs[3] = pc_INT_IPI_0 | pcs[4] = pc_INT_IPI_0 |
  pcs[5] = pc_INT_IPI_0 | pcs[6] = pc_INT_IPI_0)) : in_INT-CPU_HANDLING;
state = in_INT-IO_ISR-DISRUPT_ACT_TASK_0 & (pe = pe_INT_IPI_1 & !(pcs[0] = pc_INT_IPI_1 |
  pcs[1] = pc_INT_IPI_1 | pcs[2] = pc_INT_IPI_1 | pcs[3] = pc_INT_IPI_1 | pcs[4] = pc_INT_IPI_1 |
  pcs[5] = pc_INT_IPI_1 | pcs[6] = pc_INT_IPI_1)) : in_INT-CPU_HANDLING;
state = in_INT-IO_ISR-DISRUPT_ACT_TASK_0 & (pe = pe_INT_IPI_2 & !(pcs[0] = pc_INT_IPI_2 |
  pcs[1] = pc_INT_IPI_2 | pcs[2] = pc_INT_IPI_2 | pcs[3] = pc_INT_IPI_2 | pcs[4] = pc_INT_IPI_2 |
  pcs[5] = pc_INT_IPI_2 | pcs[6] = pc_INT_IPI_2)) : in_INT-CPU_HANDLING;
state = in_INT-IO_ISR-DISRUPT_ACT_TASK_0 & (pe = pe_INT_LOC_TIMER & !(pcs[0] = pc_INT_LOC_TIMER |
  pcs[1] = pc_INT_LOC_TIMER | pcs[2] = pc_INT_LOC_TIMER | pcs[3] = pc_INT_LOC_TIMER |
  pcs[4] = pc_INT_LOC_TIMER | pcs[5] = pc_INT_LOC_TIMER | pcs[6] = pc_INT_LOC_TIMER))
  : in_INT-CPU_HANDLING;
state = in_INT-IO_ISR-DISRUPT_ACT_TASK_0 & (pe = pe_INT_GLOB_TIMER & !(pcs[0] = pc_INT_GLOB_TIMER |
  pcs[1] = pc_INT_GLOB_TIMER | pcs[2] = pc_INT_GLOB_TIMER | pcs[3] = pc_INT_GLOB_TIMER |
  pcs[4] = pc_INT_GLOB_TIMER | pcs[5] = pc_INT_GLOB_TIMER | pcs[6] = pc_INT_GLOB_TIMER))
  : in_INT-CPU_HANDLING;
state = in_INT-IO_ISR-DISRUPT_ACT_TASK_0 & (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 |
  pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 | pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5 |
  pcs[5] = pc_INT_IO_5 | pcs[6] = pc_INT_IO_5)) : in_INT-CPU_HANDLING;
-- ... ->

state = in_INT-IO_ISR-EXIT_INT : in_INT-CO3;

-- Submachine IO_ISR - END
--

--
-- Submachine SOFT - START
state = in_INT-SOFT-DISABLE_DEF_FUN : in_INT-SOFT-CHECK_RUN_SOFTIRQ;
```

```
    state = in_INT-SOFT-CHECK_RUN_SOFTIRQ & (pe = pe_none |
        (pe = pe_INT_IPI_0 & (pcs[0] = pc_INT_IPI_0 | pcs[1] = pc_INT_IPI_0 | pcs[2] = pc_INT_IPI_0 |
        pcs[3] = pc_INT_IPI_0 | pcs[4] = pc_INT_IPI_0 | pcs[5] = pc_INT_IPI_0 | pcs[6] = pc_INT_IPI_0)) |
        (pe = pe_INT_IPI_1 & (pcs[0] = pc_INT_IPI_1 | pcs[1] = pc_INT_IPI_1 | pcs[2] = pc_INT_IPI_1 |
        pcs[3] = pc_INT_IPI_1 | pcs[4] = pc_INT_IPI_1 | pcs[5] = pc_INT_IPI_1 | pcs[6] = pc_INT_IPI_1)) |
        (pe = pe_INT_IPI_2 & (pcs[0] = pc_INT_IPI_2 | pcs[1] = pc_INT_IPI_2 | pcs[2] = pc_INT_IPI_2 |
        pcs[3] = pc_INT_IPI_2 | pcs[4] = pc_INT_IPI_2 | pcs[5] = pc_INT_IPI_2 | pcs[6] = pc_INT_IPI_2)) |
        (pe = pe_INT_LOC_TIMER & (pcs[0] = pc_INT_LOC_TIMER | pcs[1] = pc_INT_LOC_TIMER |
        pcs[2] = pc_INT_LOC_TIMER | pcs[3] = pc_INT_LOC_TIMER | pcs[4] = pc_INT_LOC_TIMER |
        pcs[5] = pc_INT_LOC_TIMER | pcs[6] = pc_INT_LOC_TIMER)) |
        (pe = pe_INT_GLOB_TIMER & (pcs[0] = pc_INT_GLOB_TIMER | pcs[1] = pc_INT_GLOB_TIMER |
        pcs[2] = pc_INT_GLOB_TIMER | pcs[3] = pc_INT_GLOB_TIMER | pcs[4] = pc_INT_GLOB_TIMER |
        pcs[5] = pc_INT_GLOB_TIMER | pcs[6] = pc_INT_GLOB_TIMER)) |
        (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
        pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5 | pcs[5] = pc_INT_IO_5 | pcs[6] = pc_INT_IO_5)) |
        (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
        pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0 | pcs[5] = pc_INT_IO_0 | pcs[6] = pc_INT_IO_0)))
        & soft_int_pending =  0 : in_INT-SOFT-SIG_KSOFTIRQD;

    state = in_INT-SOFT-CHECK_RUN_SOFTIRQ & (pe = pe_INT_IPI_0 & !(pcs[0] = pc_INT_IPI_0 |
      pcs[1] = pc_INT_IPI_0 | pcs[2] = pc_INT_IPI_0 | pcs[3] = pc_INT_IPI_0 | pcs[4] = pc_INT_IPI_0 |
      pcs[5] = pc_INT_IPI_0 | pcs[6] = pc_INT_IPI_0)) : in_INT-CPU_HANDLING;
    state = in_INT-SOFT-CHECK_RUN_SOFTIRQ & (pe = pe_INT_IPI_1 & !(pcs[0] = pc_INT_IPI_1 |
      pcs[1] = pc_INT_IPI_1 | pcs[2] = pc_INT_IPI_1 | pcs[3] = pc_INT_IPI_1 | pcs[4] = pc_INT_IPI_1 |
      pcs[5] = pc_INT_IPI_1 | pcs[6] = pc_INT_IPI_1)) : in_INT-CPU_HANDLING;
    state = in_INT-SOFT-CHECK_RUN_SOFTIRQ & (pe = pe_INT_IPI_2 & !(pcs[0] = pc_INT_IPI_2 |
      pcs[1] = pc_INT_IPI_2 | pcs[2] = pc_INT_IPI_2 | pcs[3] = pc_INT_IPI_2 | pcs[4] = pc_INT_IPI_2 |
      pcs[5] = pc_INT_IPI_2 | pcs[6] = pc_INT_IPI_2)) : in_INT-CPU_HANDLING;
    state = in_INT-SOFT-CHECK_RUN_SOFTIRQ & (pe = pe_INT_LOC_TIMER & !(pcs[0] = pc_INT_LOC_TIMER |
      pcs[1] = pc_INT_LOC_TIMER | pcs[2] = pc_INT_LOC_TIMER | pcs[3] = pc_INT_LOC_TIMER |
      pcs[4] = pc_INT_LOC_TIMER | pcs[5] = pc_INT_LOC_TIMER | pcs[6] = pc_INT_LOC_TIMER))
      : in_INT-CPU_HANDLING;
    state = in_INT-SOFT-CHECK_RUN_SOFTIRQ & (pe = pe_INT_GLOB_TIMER & !(pcs[0] = pc_INT_GLOB_TIMER |
      pcs[1] = pc_INT_GLOB_TIMER | pcs[2] = pc_INT_GLOB_TIMER | pcs[3] = pc_INT_GLOB_TIMER |
      pcs[4] = pc_INT_GLOB_TIMER | pcs[5] = pc_INT_GLOB_TIMER | pcs[6] = pc_INT_GLOB_TIMER))
      : in_INT-CPU_HANDLING;
    state = in_INT-SOFT-CHECK_RUN_SOFTIRQ & (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 |
      pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 | pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0 |
      pcs[5] = pc_INT_IO_0 | pcs[6] = pc_INT_IO_0)) : in_INT-CPU_HANDLING;
    state = in_INT-SOFT-CHECK_RUN_SOFTIRQ & (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 |
      pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 | pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5 |
      pcs[5] = pc_INT_IO_5 | pcs[6] = pc_INT_IO_5)) : in_INT-CPU_HANDLING;

    state = in_INT-SOFT-SIG_KSOFTIRQD : in_INT-RETURN_INT;

    -- Submachine SOFT - END
    --

   1 : state;
  esac;

--
-- \DELTA P_CONF : STATES / BASIC CONFIGURATIONS
-- END
--

-----------------------------------------------------------------------------------------------------

--
-- \DELTA P_IMP : PATH CONDITIONS & HISTORY
-- START
--

--
-- PATH CONDITIONS
```

```
-- START
--
init(pcs[0]) := {pc_INT_IPI_0, pc_INT_IPI_1, pc_INT_IPI_2, pc_INT_LOC_TIMER, pc_INT_GLOB_TIMER,
      pc_INT_IO_0, pc_INT_IO_5};
init(pcs[1]) := pc_none;
init(pcs[2]) := pc_none;
init(pcs[3]) := pc_none;
init(pcs[4]) := pc_none;
init(pcs[5]) := pc_none;
init(pcs[6]) := pc_none;

next(pcs[0]) := case
    counter = 0: case
      state = in_INT-CPU_RETURN : pc_none;
      1 : pcs[0];
    esac;
    1: pcs[0];
  esac;

next(pcs[1]) := case
  counter = 0 : case
    -- ... <- analogous for all path events that might interrupt in_INT-IPI-DO_RESCHED,
    --        in_INT-IPI-DO_CALL_FUN and in_INT-IPI-DO_INV_TLB
    state = in_INT-IPI-DO_RESCHED & counter < 6 &
            (pe = pe_INT_IPI_1 & !(pcs[0] = pc_INT_IPI_1 | pcs[1] = pc_INT_IPI_1 | pcs[2] = pc_INT_IPI_1 |
            pcs[3] = pc_INT_IPI_1 | pcs[4] = pc_INT_IPI_1 | pcs[5] = pc_INT_IPI_1 | pcs[6] = pc_INT_IPI_1)):
            pc_INT_IPI_1;
    -- ... ->

    -- ... <- analogous for all path events that might interrupt in_INT-SOFT-CHECK_RUN_SOFTIRQ
    state = in_INT-SOFT-CHECK_RUN_SOFTIRQ & counter < 6 &
            (pe = pe_INT_IPI_0 & !(pcs[0] = pc_INT_IPI_0 | pcs[1] = pc_INT_IPI_0 | pcs[2] = pc_INT_IPI_0 |
            pcs[3] = pc_INT_IPI_0 | pcs[4] = pc_INT_IPI_0 | pcs[5] = pc_INT_IPI_0 | pcs[6] = pc_INT_IPI_0)):
            pc_INT_IPI_0;
   -- ... ->

   -- ... <- analogous for all path events that might interrupt in_INT-IO_ISR-DISRUPT_ACT_TASK_0 and
   --        in_INT-IO_ISR-DISRUPT_ACT_TASK_5
   state = in_INT-IO_ISR-DISRUPT_ACT_TASK_0 & counter < 6 &
           (pe = pe_INT_IPI_0 & !(pcs[0] = pc_INT_IPI_0 | pcs[1] = pc_INT_IPI_0 | pcs[2] = pc_INT_IPI_0 |
           pcs[3] = pc_INT_IPI_0 | pcs[4] = pc_INT_IPI_0 | pcs[5] = pc_INT_IPI_0 | pcs[6] = pc_INT_IPI_0)):
           pc_INT_IPI_0;
   -- ... ->

   1 : pcs[1];
  esac;

  counter = 1 : case
    state = in_INT-CPU_RETURN : pc_none;
    1 : pcs[1];
  esac;
 1: pcs[1];
esac;


-- analogous: pcs[2] ... pcs[6]

-- current path condition
init(pc) := pcs[0];
next(pc) := case
  -- ... <- analogous for all path events that might interrupt in_INT-IPI-DO_RESCHED,
  --        in_INT-IPI-DO_CALL_FUN and in_INT-IPI-DO_INV_TLB
  state = in_INT-IPI-DO_RESCHED & counter < 6 &
          (pe = pe_INT_IPI_1 & !(pcs[0] = pc_INT_IPI_1 | pcs[1] = pc_INT_IPI_1 | pcs[2] = pc_INT_IPI_1 |
          pcs[3] = pc_INT_IPI_1 | pcs[4] = pc_INT_IPI_1 | pcs[5] = pc_INT_IPI_1 | pcs[6] = pc_INT_IPI_1)):
```

```
        pc_INT_IPI_1;
  -- ... ->

  -- ... <- analogous for all path events that might interrupt in_INT-SOFT-CHECK_RUN_SOFTIRQ
  state = in_INT-SOFT-CHECK_RUN_SOFTIRQ & counter < 6 &
          (pe = pe_INT_IPI_0 & !(pcs[0] = pc_INT_IPI_0 | pcs[1] = pc_INT_IPI_0 | pcs[2] = pc_INT_IPI_0 |
          pcs[3] = pc_INT_IPI_0 | pcs[4] = pc_INT_IPI_0 | pcs[5] = pc_INT_IPI_0 | pcs[6] = pc_INT_IPI_0)):
          pc_INT_IPI_0;
  -- ... ->

  -- ... <- analogous for all path events that might interrupt in_INT-IO_ISR-DISRUPT_ACT_TASK_0 and
  --        in_INT-IO_ISR-DISRUPT_ACT_TASK_5
  state = in_INT-IO_ISR-DISRUPT_ACT_TASK_0 & counter < 6 &
          (pe = pe_INT_IPI_0 & !(pcs[0] = pc_INT_IPI_0 | pcs[1] = pc_INT_IPI_0 | pcs[2] = pc_INT_IPI_0 |
          pcs[3] = pc_INT_IPI_0 | pcs[4] = pc_INT_IPI_0 | pcs[5] = pc_INT_IPI_0 | pcs[6] = pc_INT_IPI_0)):
          pc_INT_IPI_0;
  -- ... ->

  -- recursive ascent : go to previous counter position since counter is decreased at the
  -- same step as this next(pc) occurs
  state = in_INT-CPU_RETURN & counter = 1 : pcs[0];
  state = in_INT-CPU_RETURN & counter = 2 : pcs[1];
  state = in_INT-CPU_RETURN & counter = 3 : pcs[2];
  state = in_INT-CPU_RETURN & counter = 4 : pcs[3];
  state = in_INT-CPU_RETURN & counter = 5 : pcs[4];
  state = in_INT-CPU_RETURN & counter = 6 : pcs[5];

  1 : pc;
esac;

-- previous path condition
init(prevpc) := pc_none;
next(prevpc) := case
  -- ... <- analogous for counters 0..4: reset pc at current counter if state is interrupted
  state = in_INT-IPI-DO_RESCHED & !(pe = pe_none | pe = pe_INT_IPI_0 |
          (pe = pe_INT_IPI_1 & (pcs[0] = pc_INT_IPI_1 | pcs[1] = pc_INT_IPI_1 | pcs[2] = pc_INT_IPI_1 |
          pcs[3] = pc_INT_IPI_1 | pcs[4] = pc_INT_IPI_1 | pcs[5] = pc_INT_IPI_1 | pcs[6] = pc_INT_IPI_1)) |
          (pe = pe_INT_IPI_2 & (pcs[0] = pc_INT_IPI_2 | pcs[1] = pc_INT_IPI_2 | pcs[2] = pc_INT_IPI_2 |
          pcs[3] = pc_INT_IPI_2 | pcs[4] = pc_INT_IPI_2 | pcs[5] = pc_INT_IPI_2 | pcs[6] = pc_INT_IPI_2)) |
          (pe = pe_INT_LOC_TIMER & (pcs[0] = pc_INT_LOC_TIMER | pcs[1] = pc_INT_LOC_TIMER |
          pcs[2] = pc_INT_LOC_TIMER | pcs[3] = pc_INT_LOC_TIMER | pcs[4] = pc_INT_LOC_TIMER |
          pcs[5] = pc_INT_LOC_TIMER | pcs[6] = pc_INT_LOC_TIMER)) |
          (pe = pe_INT_GLOB_TIMER & (pcs[0] = pc_INT_GLOB_TIMER | pcs[1] = pc_INT_GLOB_TIMER |
          pcs[2] = pc_INT_GLOB_TIMER | pcs[3] = pc_INT_GLOB_TIMER | pcs[4] = pc_INT_GLOB_TIMER |
          pcs[5] = pc_INT_GLOB_TIMER | pcs[6] = pc_INT_GLOB_TIMER)) |
          (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
          pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0 | pcs[5] = pc_INT_IO_0 | pcs[6] = pc_INT_IO_0)) |
          (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
          pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5 | pcs[5] = pc_INT_IO_5 | pcs[6] = pc_INT_IO_5 )))
          & counter = 5: pcs[5];
  -- ... ->

  -- analogous for interruptible states in_INT-IPI-DO_CALL_FUN, in_INT-IPI-INV_TLB,
  -- in_INT-SOFT-CHECK_RUN_SOFTIRQ, in_INT-IO_ISR-DISRUPT_ACT_TASK_0, in_INT-IO_ISR-DISRUPT_ACT_TASK_5
  -- for counters 0..5

  -- recursive ascent : go to previous counter position since counter is decreased at the
  -- same step as this next(pc) occurs
  state = in_INT-CPU_RETURN & counter = 1 : pc_none;
  state = in_INT-CPU_RETURN & counter = 2 : pcs[0];
  state = in_INT-CPU_RETURN & counter = 3 : pcs[1];
  state = in_INT-CPU_RETURN & counter = 4 : pcs[2];
  state = in_INT-CPU_RETURN & counter = 5 : pcs[3];
  state = in_INT-CPU_RETURN & counter = 6 : pcs[4];
```

```
  1 : prevpc;
esac;

init(counter) := 0;
next(counter) := case
  -- recursive descent
  -- ... <- analogous for all other interruptible states
  state = in_INT-IPI-DO_RESCHED & counter < 6 & !(pe = pe_none | pe = pe_INT_IPI_0 |
          (pe = pe_INT_IPI_1 & (pcs[0] = pc_INT_IPI_1 | pcs[1] = pc_INT_IPI_1 | pcs[2] = pc_INT_IPI_1 |
          pcs[3] = pc_INT_IPI_1 | pcs[4] = pc_INT_IPI_1 | pcs[5] = pc_INT_IPI_1 | pcs[6] = pc_INT_IPI_1)) |
          (pe = pe_INT_IPI_2 & (pcs[0] = pc_INT_IPI_2 | pcs[1] = pc_INT_IPI_2 | pcs[2] = pc_INT_IPI_2 |
          pcs[3] = pc_INT_IPI_2 | pcs[4] = pc_INT_IPI_2 | pcs[5] = pc_INT_IPI_2 | pcs[6] = pc_INT_IPI_2)) |
          (pe = pe_INT_LOC_TIMER & (pcs[0] = pc_INT_LOC_TIMER | pcs[1] = pc_INT_LOC_TIMER |
          pcs[2] = pc_INT_LOC_TIMER | pcs[3] = pc_INT_LOC_TIMER | pcs[4] = pc_INT_LOC_TIMER |
          pcs[5] = pc_INT_LOC_TIMER | pcs[6] = pc_INT_LOC_TIMER)) |
          (pe = pe_INT_GLOB_TIMER & (pcs[0] = pc_INT_GLOB_TIMER | pcs[1] = pc_INT_GLOB_TIMER |
          pcs[2] = pc_INT_GLOB_TIMER | pcs[3] = pc_INT_GLOB_TIMER | pcs[4] = pc_INT_GLOB_TIMER |
          pcs[5] = pc_INT_GLOB_TIMER | pcs[6] = pc_INT_GLOB_TIMER)) |
          (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
          pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0 | pcs[5] = pc_INT_IO_0 | pcs[6] = pc_INT_IO_0)) |
          (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
          pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5 | pcs[5] = pc_INT_IO_5 | pcs[6] = pc_INT_IO_5 )))
          : counter+1;
  -- ... ->

  -- recursive ascent
  state = in_INT-CPU_RETURN & counter > 0 : (counter + (-1));

  1 : counter;
esac;

--
-- PATH CONDITIONS
-- END
--

--
-- HISTORY
-- START
--

-- previous / last seen historic state
init(hist) := state_none;
next(hist) := case
      -- interruption point is unambiguous per path condition
      prevpc = pc_INT_IPI_0 : in_INT-IPI-DO_RESCHED;
      prevpc = pc_INT_IPI_1 : in_INT-IPI-DO_CALL_FUN;
      prevpc = pc_INT_IPI_2 : in_INT-IPI-DO_INV_TLB;
      prevpc = pc_INT_IO_0 : in_INT-IO_ISR-DISRUPT_ACT_TASK_0;
      prevpc = pc_INT_IO_5 : in_INT-IO_ISR-DISRUPT_ACT_TASK_5;
      1 : state_none;
  esac;

--
-- HISTORY
-- END
--

--
-- \DELTA P_IMP : PATH CONDITIONS & HISTORY
-- END
--
-----------------------------------------------------------------------------------------------------------
--
-- \DELTA P_EXP : CONDITIONS & VARIABLES
```

```
-- START
--
-- setting condition soft_int_pending
init(soft_int_pending) := 0;
next(soft_int_pending) := case
  -- decrement if RUN_SOFTIRQ steps through without being interrupted: finalize all pending softIRQs
  state = in_INT-SOFT-CHECK_RUN_SOFTIRQ & (pe = pe_none |
          (pe = pe_INT_IPI_0 & (pcs[0] = pc_INT_IPI_0 | pcs[1] = pc_INT_IPI_0 | pcs[2] = pc_INT_IPI_0 |
          pcs[3] = pc_INT_IPI_0 | pcs[4] = pc_INT_IPI_0 | pcs[5] = pc_INT_IPI_0 | pcs[6] = pc_INT_IPI_0)) |
          (pe = pe_INT_IPI_1 & (pcs[0] = pc_INT_IPI_1 | pcs[1] = pc_INT_IPI_1 | pcs[2] = pc_INT_IPI_1 |
          pcs[3] = pc_INT_IPI_1 | pcs[4] = pc_INT_IPI_1 | pcs[5] = pc_INT_IPI_1 | pcs[6] = pc_INT_IPI_1)) |
          (pe = pe_INT_IPI_2 & (pcs[0] = pc_INT_IPI_2 | pcs[1] = pc_INT_IPI_2 | pcs[2] = pc_INT_IPI_2 |
          pcs[3] = pc_INT_IPI_2 | pcs[4] = pc_INT_IPI_2 | pcs[5] = pc_INT_IPI_2 | pcs[6] = pc_INT_IPI_2)) |
          (pe = pe_INT_LOC_TIMER & (pcs[0] = pc_INT_LOC_TIMER | pcs[1] = pc_INT_LOC_TIMER |
          pcs[2] = pc_INT_LOC_TIMER | pcs[3] = pc_INT_LOC_TIMER | pcs[4] = pc_INT_LOC_TIMER |
          pcs[5] = pc_INT_LOC_TIMER | pcs[6] = pc_INT_LOC_TIMER)) |
          (pe = pe_INT_GLOB_TIMER & (pcs[0] = pc_INT_GLOB_TIMER | pcs[1] = pc_INT_GLOB_TIMER |
          pcs[2] = pc_INT_GLOB_TIMER | pcs[3] = pc_INT_GLOB_TIMER | pcs[4] = pc_INT_GLOB_TIMER |
          pcs[5] = pc_INT_GLOB_TIMER | pcs[6] = pc_INT_GLOB_TIMER)) |
          (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
          pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0 | pcs[5] = pc_INT_IO_0 | pcs[6] = pc_INT_IO_0)) |
          (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
          pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5 | pcs[5] = pc_INT_IO_5 | pcs[6] = pc_INT_IO_5 )))
          & soft_int_pending > 0 : 0;

  -- increment when DISRUPT_ACT_TASK_x steps through
  -- ... <- analogous for TASK_5
  state = in_INT-IO_ISR-DISRUPT_ACT_TASK_0 & soft_int_pending < 9 & (pe = pe_none | pe = pe_INT_IO_0 |
         (pe = pe_INT_IPI_0 & (pcs[0] = pc_INT_IPI_0 | pcs[1] = pc_INT_IPI_0 | pcs[2] = pc_INT_IPI_0 |
         pcs[3] = pc_INT_IPI_0 | pcs[4] = pc_INT_IPI_0 | pcs[5] = pc_INT_IPI_0 | pcs[6] = pc_INT_IPI_0)) |
         (pe = pe_INT_IPI_1 & (pcs[0] = pc_INT_IPI_1 | pcs[1] = pc_INT_IPI_1 | pcs[2] = pc_INT_IPI_1 |
         pcs[3] = pc_INT_IPI_1 | pcs[4] = pc_INT_IPI_1 | pcs[5] = pc_INT_IPI_1 | pcs[6] = pc_INT_IPI_1)) |
         (pe = pe_INT_IPI_2 & (pcs[0] = pc_INT_IPI_2 | pcs[1] = pc_INT_IPI_2 | pcs[2] = pc_INT_IPI_2 |
         pcs[3] = pc_INT_IPI_2 | pcs[4] = pc_INT_IPI_2 | pcs[5] = pc_INT_IPI_2 | pcs[6] = pc_INT_IPI_2)) |
         (pe = pe_INT_LOC_TIMER & (pcs[0] = pc_INT_LOC_TIMER | pcs[1] = pc_INT_LOC_TIMER |
         pcs[2] = pc_INT_LOC_TIMER | pcs[3] = pc_INT_LOC_TIMER | pcs[4] = pc_INT_LOC_TIMER |
         pcs[5] = pc_INT_LOC_TIMER | pcs[6] = pc_INT_LOC_TIMER)) |
         (pe = pe_INT_GLOB_TIMER & (pcs[0] = pc_INT_GLOB_TIMER | pcs[1] = pc_INT_GLOB_TIMER |
         pcs[2] = pc_INT_GLOB_TIMER | pcs[3] = pc_INT_GLOB_TIMER | pcs[4] = pc_INT_GLOB_TIMER |
         pcs[5] = pc_INT_GLOB_TIMER | pcs[6] = pc_INT_GLOB_TIMER)) |
         (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
         pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5 | pcs[5] = pc_INT_IO_5 | pcs[6] = pc_INT_IO_5)))
         & soft_int_pending = 0 : 1;

  -- ... ->
  1 : soft_int_pending;
esac;

init(tif_need_reschedule) := 0;
next(tif_need_reschedule) := case
    state = in_INT-LT_ISR-UPD_PROC_TIMES-TICK-SET_RESCHED : 1;
    state = in_SCHEDULER : 0;
    1 : tif_need_reschedule;
  esac;


-- history indicator for submachine SOFT
init(submachine_soft_interrupted) := 0;
next(submachine_soft_interrupted) := case
  state = in_INT-SOFT-CHECK_RUN_SOFTIRQ & (pe = pe_INT_IPI_0 & !(pcs[0] = pc_INT_IPI_0 | pcs[1] = pc_INT_IPI_0 |
          pcs[2] = pc_INT_IPI_0 | pcs[3] = pc_INT_IPI_0 | pcs[4] = pc_INT_IPI_0 | pcs[5] = pc_INT_IPI_0 |
          pcs[6] = pc_INT_IPI_0)) : 1;
  -- analogous for all other path events that could interrupt

  -- interruptible state passed --> SOFT about to terminate
```

```
    state = in_INT-SOFT-SIG_KSOFTIRQD : 0;

    1 : submachine_soft_interrupted;
  esac;

  --
  -- \DELTA P_EXP : CONDITIONS & VARIABLES
  -- END
  --
  ------------------------------------------------------------------------------------------------------------
  -- FAIRNESS CONSTRAINTS
  --
  -- ... <- analogous for all other path conditions: path event is free to occur an infinite number of times
  JUSTICE pcs[0]!=pc_INT_IPI_0 & pcs[1]!=pc_INT_IPI_0 & pcs[2]!=pc_INT_IPI_0 & pcs[3]!=pc_INT_IPI_0 &
          pcs[4]!=pc_INT_IPI_0 & pcs[5]!=pc_INT_IPI_0 & pcs[6]!=pc_INT_IPI_0;
  -- ... ->

  --
  ------------------------------------------------------------------------------------------------------------
  -- INDICATORS
  --
  -- infinite handling
  LTLSPEC
    G(state = in_INT-CPU_HANDLING -> F(state=in_USERSPACE))

  -- losing interrupts
  LTLSPEC G(state = in_INT-CPU_HANDLING & pc = in_INT_IPI_0
    -> F(state=in_INT-DO_RESCHED))
  LTLSPEC G(state = in_INT-CPU_HANDLING & pc = in_INT_IPI_1
    -> F(state=in_INT-IPI-DO_CALL_FUN))
  LTLSPEC G(state = in_INT-CPU_HANDLING & pc = in_INT_IPI_2
    -> F(state=in_INT-IPI-DO_INV_TLB))
  LTLSPEC G(state = in_INT-CPU_HANDLING & pc = in_INT_IO_0
    -> F(state=in_INT-IO_ISR-DISRUPT_ACT_TASK_0))
  LTLSPEC G(state = in_INT-CPU_HANDLING & pc = in_INT_IO_5
    -> F(state=in_INT-IO_ISR-DISRUPT_ACT_TASK_5))
  LTLSPEC G(state = in_INT-CPU_HANDLING & pc = in_INT_LOC_TIMER
    -> F(state=in_INT-LT_ISR-UPD_PROC_TIMES-TICK-SET_RESCHED))
  LTLSPEC G(state = in_INT-CPU_HANDLING & pc = in_INT_GLOB_TIMER
    -> F(state=in_INT-GT_ISR-JIFFIES))
```

B.2 OpenBSD

```
MODULE main
VAR
    state : {
        -- dummy user-space state
        in_USERSPACE,
        -- INT states
        in_IKCP-INT-CREATE_FRAME, in_IKCP-INT-XDORETI, in_IKCP-INT-IPI-EOI, in_IKCP-INT-IPI-IPI,
        -- IO_APIC states
        in_IKCP-INT-IO_APIC-GRP-DET_HANDLER, in_IKCP-INT-IO_APIC-GRP-UPDATE_STAT,
        in_IKCP-INT-IO_APIC-IOAPIC_HDL_0, in_IKCP-INT-IO_APIC-IOAPIC_HDL_5,
        in_IKCP-INT-IO_APIC-EOI, in_IKCP-INT-IO_APIC-SET_LOCK-LOCK,
        in_IKCP-INT-IO_APIC-SET_LOCK-SPLX, in_IKCP-INT-IO_APIC-REM_LOCK-REMOVE,
        in_IKCP-INT-IO_APIC-REM_LOCK-SPLX,
        -- LAPIC_TIMER states
        in_IKCP-INT-LAPIC_TIMER-EOI, in_IKCP-INT-LAPIC_TIMER-SET_LOCK-LOCK,
        in_IKCP-INT-LAPIC_TIMER-SET_LOCK-SPLX,
        in_IKCP-INT-LAPIC_TIMER-REM_LOCK-REMOVE, in_IKCP-INT-LAPIC_TIMER-REM_LOCK-SPLX,
        in_IKCP-INT-LAPIC_TIMER-TSC_ACT, in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-TIMER,
        in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-SCHEDULE,
        in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-SETSOFTCLOCK,
```

```
        -- SYSCALL states
        in_IKCP-SYS-DO_SYSCALL-GET_PARAM, in_IKCP-SYS-DO_SYSCALL-EXEC_SYSCALL,
        in_IKCP-SYS-DO_SYSCALL-USERRET, in_IKCP-SYS-SET_LOCK-LOCK, in_IKCP-SYS-SET_LOCK-SPLX,
        in_IKCP-SYS-REM_LOCK-REMOVE, in_IKCP-SYS-REM_LOCK-SPLX
        };

    pe : {pe_none, pe_INT_IPI, pe_INT_LAPIC, pe_INT_IO_0, pe_INT_IO_5, pe_SYS};

    counter : 0..4;
    pcs: array 0..4 of {pc_none, pc_INT_IPI, pc_INT_LAPIC, pc_INT_IO_0, pc_INT_IO_5, pc_SYS};

    -- current and previous pc : constructed to avoid non-allowed dynamic access to pcs array
    pc: {pc_none, pc_INT_IPI, pc_INT_LAPIC, pc_INT_IO_0, pc_INT_IO_5, pc_SYS};
    prevpc : {pc_none, pc_INT_IPI, pc_INT_LAPIC, pc_INT_IO_0, pc_INT_IO_5, pc_SYS};

    -- last interrupted historic state
    hist : {
        state_none,
        in_IKCP-INT-IPI-IPI,
        in_IKCP-INT-IO_APIC-GRP-DET_HANDLER, in_IKCP-INT-IO_APIC-GRP-UPDATE_STAT,
        in_IKCP-INT-IO_APIC-IOAPIC_HDL_0, in_IKCP-INT-IO_APIC-IOAPIC_HDL_5,
        in_IKCP-INT-IO_APIC-SET_LOCK-LOCK, in_IKCP-INT-IO_APIC-REM_LOCK-REMOVE,
        in_IKCP-INT-LAPIC_TIMER-EOI,
        in_IKCP-INT-LAPIC_TIMER-SET_LOCK-LOCK, in_IKCP-INT-LAPIC_TIMER-REM_LOCK-REMOVE,
        in_IKCP-INT-LAPIC_TIMER-TSC_ACT, in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-TIMER,
        in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-SCHEDULE, in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-SETSOFTCLOCK,
        in_IKCP-SYS-DO_SYSCALL-GET_PARAM, in_IKCP-SYS-DO_SYSCALL-EXEC_SYSCALL,
        in_IKCP-SYS-SET_LOCK-LOCK, in_IKCP-SYS-REM_LOCK-REMOVE
        };

    -- historic states: all interruptible states that can be subject to long-term history
    history: array 0..3 of {
        state_none,
        in_IKCP-INT-IPI-IPI,
        in_IKCP-INT-IO_APIC-GRP-DET_HANDLER, in_IKCP-INT-IO_APIC-GRP-UPDATE_STAT,
        in_IKCP-INT-IO_APIC-IOAPIC_HDL_0, in_IKCP-INT-IO_APIC-IOAPIC_HDL_5,
        in_IKCP-INT-IO_APIC-SET_LOCK-LOCK, in_IKCP-INT-IO_APIC-REM_LOCK-REMOVE,
        in_IKCP-INT-LAPIC_TIMER-SET_LOCK-LOCK, in_IKCP-INT-LAPIC_TIMER-REM_LOCK-REMOVE,
        in_IKCP-INT-LAPIC_TIMER-TSC_ACT, in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-TIMER,
        in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-SCHEDULE, in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-SETSOFTCLOCK,
        in_IKCP-SYS-DO_SYSCALL-GET_PARAM, in_IKCP-SYS-DO_SYSCALL-EXEC_SYSCALL,
        in_IKCP-SYS-SET_LOCK-LOCK, in_IKCP-SYS-REM_LOCK-REMOVE
    };


ASSIGN

  -- \DELTA P_CONF : STATES / BASIC CONFIGURATIONS
  -- START

  init(state) :=
          case
            -- split state SP1
            pc = pc_SYS : in_IKCP-SYS-DO_SYSCALL-GET_PARAM;
            1 : in_IKCP-INT-CREATE_FRAME;
          esac;


  next(state) := case

    -- Submachine SYS - START

    -- .. <-- analogous for all other interruptible states of the SYS submachine:
    -- in_IKCP-SYS-SET_LOCK-LOCK, in_IKCP-SYS-DO_SYSCALL-EXEC_SYSCALL,
    -- in_IKCP-SYS-REM_LOCK-REMOVE,
```

```
state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM & (pe = pe_none | pe = pe_SYS |
  (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 |
  pcs[2] = pc_INT_IO_0 | pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)) |
  (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 |
  pcs[2] = pc_INT_IO_5 | pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)) |
  (pe = pe_INT_LAPIC & (pcs[0] = pc_INT_LAPIC | pcs[1] = pc_INT_LAPIC |
  pcs[2] = pc_INT_LAPIC | pcs[3] = pc_INT_LAPIC | pcs[4] = pc_INT_LAPIC)) |
  (pe = pe_INT_IPI & (pcs[0] = pc_INT_IPI | pcs[1] = pc_INT_IPI |
  pcs[2] = pc_INT_IPI | pcs[3] = pc_INT_IPI | pcs[4] = pc_INT_IPI))) :
  in_IKCP-SYS-SET_LOCK-LOCK;

state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM & !(pe = pe_none | pe = pe_SYS |
  (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 |
  pcs[2] = pc_INT_IO_0 | pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)) |
  (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 |
  pcs[2] = pc_INT_IO_5 | pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)) |
  (pe = pe_INT_LAPIC & (pcs[0] = pc_INT_LAPIC | pcs[1] = pc_INT_LAPIC |
  pcs[2] = pc_INT_LAPIC | pcs[3] = pc_INT_LAPIC | pcs[4] = pc_INT_LAPIC)) |
  (pe = pe_INT_IPI & (pcs[0] = pc_INT_IPI | pcs[1] = pc_INT_IPI |
  pcs[2] = pc_INT_IPI | pcs[3] = pc_INT_IPI | pcs[4] = pc_INT_IPI))) :
  in_IKCP-INT-CREATE_FRAME;

-- .. -->

state = in_IKCP-SYS-SET_LOCK-SPLX : in_IKCP-SYS-DO_SYSCALL-EXEC_SYSCALL;
state = in_IKCP-SYS-REM_LOCK-SPLX : in_IKCP-SYS-DO_SYSCALL-USERRET;

-- go directly to user space because no recursive ascent is possible:
-- a syscall can't interrupt any other IKCP
state = in_IKCP-SYS-DO_SYSCALL-USERRET : in_USERSPACE;

-- Submachine SYS - END


------ Submachine INT - Interrupt Processing

-- split state SP3
state = in_IKCP-INT-CREATE_FRAME & pc = pc_INT_IPI : in_IKCP-INT-IPI-EOI;
state = in_IKCP-INT-CREATE_FRAME & (pc = pc_INT_IO_0 | pc = pc_INT_IO_5) :
        in_IKCP-INT-IO_APIC-GRP-DET_HANDLER;
state = in_IKCP-INT-CREATE_FRAME & pc = pc_INT_LAPIC : in_IKCP-INT-LAPIC_TIMER-EOI;

-- Submachine IPI - START

state = in_IKCP-INT-IPI-EOI : in_IKCP-INT-IPI-IPI;

state = in_IKCP-INT-IPI-IPI & (pe = pe_none | pe = pe_INT_IPI | pe = pe_SYS |
  (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
  pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)) |
  (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)) |
  (pe = pe_INT_LAPIC & (pcs[0] = pc_INT_LAPIC | pcs[1] = pc_INT_LAPIC | pcs[2] = pc_INT_LAPIC |
  pcs[3] = pc_INT_LAPIC | pcs[4] = pc_INT_LAPIC))) :
  in_IKCP-INT-XDORETI;

state = in_IKCP-INT-IPI-IPI & !(pe = pe_none | pe = pe_INT_IPI | pe = pe_SYS |
  (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
  pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)) |
  (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)) |
  (pe = pe_INT_LAPIC & (pcs[0] = pc_INT_LAPIC | pcs[1] = pc_INT_LAPIC | pcs[2] = pc_INT_LAPIC |
  pcs[3] = pc_INT_LAPIC | pcs[4] = pc_INT_LAPIC))) :
  in_IKCP-INT-CREATE_FRAME;

-- Submachine IPI - END
```

```
-- Submachine IO - START

-- ... <-- analogous for the other interruptible states of submachine IO_APIC:
-- in_IKCP-INT-IO_APIC-SET_LOCK-LOCK, in_IKCP-INT-IO_APIC-IOAPIC_HDL_0,
-- in_IKCP-INT-IO_APIC-IOAPIC_HDL_5, in_IKCP-INT-IO_APIC-GRP-UPDATE_STAT,
-- in_IKCP-INT-IO_APIC-REM_LOCK-REMOVE
state = in_IKCP-INT-IO_APIC-GRP-DET_HANDLER & (pe = pe_none | pe = pe_SYS |
  (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
  pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)) |
  (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)) |
  (pe = pe_INT_LAPIC & (pcs[0] = pc_INT_LAPIC | pcs[1] = pc_INT_LAPIC | pcs[2] = pc_INT_LAPIC |
  pcs[3] = pc_INT_LAPIC | pcs[4] = pc_INT_LAPIC)) |
  (pe = pe_INT_IPI & (pcs[0] = pc_INT_IPI | pcs[1] = pc_INT_IPI | pcs[2] = pc_INT_IPI |
  pcs[3] = pc_INT_IPI | pcs[4] = pc_INT_IPI))) :
  in_IKCP-INT-IO_APIC-SET_LOCK-LOCK;

state = in_IKCP-INT-IO_APIC-GRP-DET_HANDLER & !(pe = pe_none | pe = pe_SYS |
  (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
  pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)) |
  (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)) |
  (pe = pe_INT_LAPIC & (pcs[0] = pc_INT_LAPIC | pcs[1] = pc_INT_LAPIC | pcs[2] = pc_INT_LAPIC |
  pcs[3] = pc_INT_LAPIC | pcs[4] = pc_INT_LAPIC)) |
  (pe = pe_INT_IPI & !(pcs[0] = pc_INT_IPI | pcs[1] = pc_INT_IPI | pcs[2] = pc_INT_IPI |
  pcs[3] = pc_INT_IPI | pcs[4] = pc_INT_IPI))) :
  in_IKCP-INT-CREATE_FRAME;

 state = in_IKCP-INT-IO_APIC-SET_LOCK-SPLX & pc = pc_INT_IO_0 : in_IKCP-INT-IO_APIC-IOAPIC_HDL_0;
 state = in_IKCP-INT-IO_APIC-SET_LOCK-SPLX & pc = pc_INT_IO_5 : in_IKCP-INT-IO_APIC-IOAPIC_HDL_5;
 state = in_IKCP-INT-IO_APIC-REM_LOCK-SPLX : in_IKCP-INT-IO_APIC-EOI;
 state = in_IKCP-INT-IO_APIC-EOI : in_IKCP-INT-XDORETI;

 -- Submachine IO - END

 -- Submachine LAPIC_TIMER - START

 state = in_IKCP-INT-LAPIC_TIMER-EOI : in_IKCP-INT-LAPIC_TIMER-SET_LOCK-LOCK;

-- ... <-- analogous for the other interruptible states of submachine LAPIC_TIMER:
-- in_IKCP-INT-LAPIC_TIMER-TSC_ACT, in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-TIMER,
-- in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-SCHEDULE, state = in_IKCP-INT-LAPIC_TIMER-REM_LOCK-REMOVE
 state = in_IKCP-INT-LAPIC_TIMER-SET_LOCK-LOCK & (pe = pe_none | pe = pe_INT_LAPIC | pe = pe_SYS |
   (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
   pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)) |
   (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
   pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)) |
   (pe = pe_INT_IPI & (pcs[0] = pc_INT_IPI | pcs[1] = pc_INT_IPI | pcs[2] = pc_INT_IPI |
   pcs[3] = pc_INT_IPI | pcs[4] = pc_INT_IPI))) :
   in_IKCP-INT-LAPIC_TIMER-SET_LOCK-SPLX;

 state = in_IKCP-INT-LAPIC_TIMER-SET_LOCK-LOCK & !(pe = pe_none | pe = pe_INT_LAPIC | pe = pe_SYS |
   (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
   pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)) |
   (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
   pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)) |
   (pe = pe_INT_IPI & (pcs[0] = pc_INT_IPI | pcs[1] = pc_INT_IPI | pcs[2] = pc_INT_IPI |
   pcs[3] = pc_INT_IPI | pcs[4] = pc_INT_IPI))) :
   in_IKCP-INT-CREATE_FRAME;

 state = in_IKCP-INT-LAPIC_TIMER-SET_LOCK-SPLX : in_IKCP-INT-LAPIC_TIMER-TSC_ACT;
 state = in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-SETSOFTCLOCK : in_IKCP-INT-LAPIC_TIMER-REM_LOCK-REMOVE;
 state = in_IKCP-INT-LAPIC_TIMER-REM_LOCK-SPLX : in_IKCP-INT-XDORETI;
```

```
    -- Submachine LAPIC_TIMER - END

    -- history : jumping back to last left state
    state = in_IKCP-INT-XDORETI & hist != state_none : hist;

    -- no history: return to user space
    state = in_IKCP-INT-XDORETI & hist = state_none : in_USERSPACE;

    1 : state;
  esac;

--
-- \DELTA P_CONF : STATES / BASIC CONFIGURATIONS
-- END
-----------------------------------------------------------------------------------------------------------
--
-- \DELTA P_IMP : PATH CONDITIONS & HISTORY
-- START
--

-- PATH CONDITIONS
-- START
init(pcs[0]) := {pc_INT_IPI, pc_INT_LAPIC, pc_INT_IO_0, pc_INT_IO_5, pc_SYS};
init(pcs[1]) := pc_none;
init(pcs[2]) := pc_none;
init(pcs[3]) := pc_none;
init(pcs[4]) := pc_none;

next(pcs[0]) := case
          counter = 0: case
              state = in_IKCP-INT-XDORETI : pc_none;
              state = in_IKCP-SYS-DO_SYSCALL-USERRET : pc_none;
              1 : pcs[0];
              esac;
           1: pcs[0];
          esac;

-- ... <-- analogous for pcs[2] .. pcs[4]
next(pcs[1]) := case
  counter = 0 : case
    -- ... <-- analogous for all other interruptible states and their interrupting path events
    state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM &
            (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 |
            pcs[2] = pc_INT_IO_5 | pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)):
            pc_INT_IO_5;
    state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM &
            (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 |
            pcs[2] = pc_INT_IO_0 | pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)):
            pc_INT_IO_0;
    state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM &
            (pe = pe_INT_IPI & !(pcs[0] = pc_INT_IPI | pcs[1] = pc_INT_IPI |
            pcs[2] = pc_INT_IPI | pcs[3] = pc_INT_IPI | pcs[4] = pc_INT_IPI)):
            pc_INT_IPI;
    state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM &
            (pe = pe_INT_LAPIC & !(pcs[0] = pc_INT_LAPIC | pcs[1] = pc_INT_LAPIC |
            pcs[2] = pc_INT_LAPIC | pcs[3] = pc_INT_LAPIC | pcs[4] = pc_INT_LAPIC)):
            pc_INT_LAPIC;
    -- ... -->

    1 : pcs[1];
  esac;
  counter = 1 : case
    state = in_IKCP-INT-XDORETI : pc_none;
    state = in_IKCP-SYS-DO_SYSCALL-USERRET : pc_none;
    1 : pcs[1];
```

```
  esac;
  1: pcs[1];
esac;

-- current path condition
init(pc) := pcs[0];
next(pc) := case
  -- recursive descent
  -- ... <-- analogous for all other interruptible states and their interrupting path events
  state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM &
            (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 |
            pcs[2] = pc_INT_IO_5 | pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)):
            pc_INT_IO_5;
  state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM &
            (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 |
            pcs[2] = pc_INT_IO_0 | pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)):
            pc_INT_IO_0;
  state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM &
            (pe = pe_INT_IPI & !(pcs[0] = pc_INT_IPI | pcs[1] = pc_INT_IPI |
            pcs[2] = pc_INT_IPI | pcs[3] = pc_INT_IPI | pcs[4] = pc_INT_IPI)):
            pc_INT_IPI;
  state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM &
            (pe = pe_INT_LAPIC & !(pcs[0] = pc_INT_LAPIC | pcs[1] = pc_INT_LAPIC |
            pcs[2] = pc_INT_LAPIC | pcs[3] = pc_INT_LAPIC | pcs[4] = pc_INT_LAPIC)):
            pc_INT_LAPIC;
  -- ... -->

  -- recursive ascent : go to previous counter position since counter is decreased at the
  -- same step as this next(pc) occurs
  state = in_IKCP-INT-XDORETI & counter = 1 : pcs[0];
  state = in_IKCP-INT-XDORETI & counter = 2 : pcs[1];
  state = in_IKCP-INT-XDORETI & counter = 3 : pcs[2];
  state = in_IKCP-INT-XDORETI & counter = 4 : pcs[3];
  1 : pc;
esac;

-- previous path condition
init(prevpc) := pc_none;
next(prevpc) := case
  -- recursive descent
  -- ... <-- analogous for all other interruptible states and counters / array indices 2..0
  state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM & !(pe = pe_none | pe = pe_SYS |
    (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
    pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)) |
    (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
    pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)) |
    (pe = pe_INT_LAPIC & (pcs[0] = pc_INT_LAPIC | pcs[1] = pc_INT_LAPIC | pcs[2] = pc_INT_LAPIC |
    pcs[3] = pc_INT_LAPIC | pcs[4] = pc_INT_LAPIC)) |
    (pe = pe_INT_IPI & (pcs[0] = pc_INT_IPI | pcs[1] = pc_INT_IPI | pcs[2] = pc_INT_IPI |
    pcs[3] = pc_INT_IPI | pcs[4] = pc_INT_IPI)))
    & counter = 3 : pcs[3];

    -- recursive ascent
    state = in_IKCP-INT-XDORETI & counter = 1 : pc_none;
    state = in_IKCP-INT-XDORETI & counter = 2 : pcs[0];
    state = in_IKCP-INT-XDORETI & counter = 3 : pcs[1];
    state = in_IKCP-INT-XDORETI & counter = 4 : pcs[2];

    1 : prevpc;
  esac;

init(counter) := 0;
next(counter) := case
  -- ... <-- analogous for all other interruptible states
  state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM & !(pe = pe_none | pe = pe_INT_IPI | pe = pe_SYS |
```

```
    (pe = pe_INT_LAPIC & (pcs[0] = pc_INT_LAPIC | pcs[1] = pc_INT_LAPIC | pcs[2] = pc_INT_LAPIC |
    pcs[3] = pc_INT_LAPIC | pcs[4] = pc_INT_LAPIC)) |
    (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
    pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)) |
    (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
    pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)))
    & counter < 4 : counter+1;
  -- ... -->

  state = in_IKCP-INT-XDORETI & counter > 0 : (counter + (-1));
  1 : counter;
esac;

-- PATH CONDITIONS - END

-- HISTORY - START

-- previous / last seen historic state
init(hist) := history[0];
next(hist) := case
  -- ... <-- analogous for all other interruptible states
  state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM & !(pe = pe_none | pe = pe_SYS |
    (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
    pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)) |
    (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
    pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)) |
    (pe = pe_INT_LAPIC & (pcs[0] = pc_INT_LAPIC | pcs[1] = pc_INT_LAPIC | pcs[2] = pc_INT_LAPIC |
    pcs[3] = pc_INT_LAPIC | pcs[4] = pc_INT_LAPIC)) |
    (pe = pe_INT_IPI & (pcs[0] = pc_INT_IPI | pcs[1] = pc_INT_IPI | pcs[2] = pc_INT_IPI |
    pcs[3] = pc_INT_IPI | pcs[4] = pc_INT_IPI)))
    : in_IKCP-SYS-DO_SYSCALL-GET_PARAM;
  -- ... -->

  state = in_IKCP-INT-XDORETI & counter = 1 : state_none;
  state = in_IKCP-INT-XDORETI & counter = 2 : history[0];
  state = in_IKCP-INT-XDORETI & counter = 3 : history[1];
  state = in_IKCP-INT-XDORETI & counter = 4 : history[2];

  1 : hist;
esac;


init(history[0]) := state_none;
init(history[1]) := state_none;
init(history[2]) := state_none;
init(history[3]) := state_none;

-- ... <-- analogous for history[1] .. history[3]
next(history[0]) := case
  counter = 0: case
    -- ... <-- analogous for all other interruptible states
    state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM & !(pe = pe_none | pe = pe_SYS |
      (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
      pcs[3] = pc_INT_IO_0 | pcs[4] = pc_INT_IO_0)) |
      (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
      pcs[3] = pc_INT_IO_5 | pcs[4] = pc_INT_IO_5)) |
      (pe = pe_INT_LAPIC & (pcs[0] = pc_INT_LAPIC | pcs[1] = pc_INT_LAPIC | pcs[2] = pc_INT_LAPIC |
      pcs[3] = pc_INT_LAPIC | pcs[4] = pc_INT_LAPIC)) |
      (pe = pe_INT_IPI & (pcs[0] = pc_INT_IPI | pcs[1] = pc_INT_IPI | pcs[2] = pc_INT_IPI |
      pcs[3] = pc_INT_IPI | pcs[4] = pc_INT_IPI)))
      : in_IKCP-SYS-DO_SYSCALL-GET_PARAM;
    -- ... -->

    1 : history[0];
  esac;
```

```
    counter = 1: case
      state = in_IKCP-INT-XDORETI : state_none;
      state = in_IKCP-SYS-DO_SYSCALL-USERRET : state_none;
      1 : history[0];
    esac;

1: history[0];
  esac;

  -- HISTORY - END

  -- \DELTA P_IMP : PATH CONDITIONS & HISTORY
  -- END

  ---------------------------------------------------------------------------------------------------

  -- \DELTA P_EXP : CONDITIONS & VARIABLES
  -- START
  --
  -- NONE
  --
  -- \DELTA P_EXP : CONDITIONS & VARIABLES
  -- END

  ---------------------------------------------------------------------------------------------------

  -- FAIRNESS CONSTRAINTS
  --
  JUSTICE pcs[0]!=pc_INT_IPI & pcs[1]!=pc_INT_IPI & pcs[2]!=pc_INT_IPI & pcs[3]!=pc_INT_IPI;
  JUSTICE pcs[0]!=pc_INT_IO_0 & pcs[1]!=pc_INT_IO_0 & pcs[2]!=pc_INT_IO_0 & pcs[3]!=pc_INT_IO_0;
  JUSTICE pcs[0]!=pc_INT_IO_5 & pcs[1]!=pc_INT_IO_5 & pcs[2]!=pc_INT_IO_5 & pcs[3]!=pc_INT_IO_5;
  JUSTICE pcs[0]!=pc_INT_LAPIC & pcs[1]!=pc_INT_LAPIC & pcs[2]!=pc_INT_LAPIC & pcs[3]!=pc_INT_LAPIC;

  --
  ---------------------------------------------------------------------------------------------------
  -- INDICATORS
  --
  -- infinite handling
  LTLSPEC
    G(state = in_IKCP-INT-CREATE_FRAME & hist = state_none & pcs[0] != pc_SYS
        -> F(state = in_USERSPACE));
  LTLSPEC
    G(state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM & hist = state_none -> F(state = in_USERSPACE))

  -- losing interrupts
  LTLSPEC
    G(state = in_IKCP-INT-CREATE_FRAME & pc = pc_INT_IPI -> F(state = in_IKCP-INT-IPI-IPI));
  LTLSPEC
    G(state = in_IKCP-INT-CREATE_FRAME & pc = pc_INT_IO_0
    -> F(state = in_IKCP-INT-IO_APIC-IOAPIC_HDL_0));
  LTLSPEC
    G(state = in_IKCP-INT-CREATE_FRAME & pc = pc_INT_IO_5
    -> F(state = in_IKCP-INT-IO_APIC-IOAPIC_HDL_5));
  LTLSPEC
    G(state = in_IKCP-INT-CREATE_FRAME & pc = pc_INT_LAPIC ->
    F(state = in_IKCP-INT-LAPIC_TIMER-HARDCLOCK-SETSOFTCLOCK));
  LTLSPEC
    G(state = in_IKCP-SYS-DO_SYSCALL-GET_PARAM & pc = pc_SYS
    -> F(state = in_IKCP-SYS-DO_SYSCALL-EXEC_SYSCALL));
```

B.3 Pistachio

```
MODULE main
VAR
  state : {state_none,
    -- Kernel Mode
    in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI, in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST,
    in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET,
    in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK, in_KERNEL-INTERRUPT-HW_IRQ-EOI,
    in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_0, in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_5,
    in_KERNEL-INTERRUPT-HW_IRQ-IRET,
    in_KERNEL-INTERRUPT-TIMER-EOI_APIC, in_KERNEL-INTERRUPT-TIMER-IRET,
    in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-UPD_TIMER,
    in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-TOTAL_QUANTUM,
    in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-RESCHED,
    in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-XCPU_MAILBOX,
    -- User Mode
    in_USER-HANDLERS-IRQ_THREAD,
    -- dummy user-space state
    in_USERSPACE
    };

    pe : {pe_none, pe_INT_XCPU, pe_INT_TIMER, pe_INT_IO_0, pe_INT_IO_5};

    counter : 0..3;
    pcs: array 0..3 of {pc_none, pc_INT_XCPU, pc_INT_TIMER, pc_INT_IO_0, pc_INT_IO_5};

    pc: {pc_none, pc_INT_XCPU, pc_INT_TIMER, pc_INT_IO_0, pc_INT_IO_5};
    prevpc : {pc_none, pc_INT_XCPU, pc_INT_TIMER, pc_INT_IO_0, pc_INT_IO_5};

    -- last interrupted historic state
    hist : {
      state_none,
      in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST,
      in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_0, in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_5,
      in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-UPD_TIMER,
      in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-TOTAL_QUANTUM,
      in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-RESCHED,
      in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-XCPU_MAILBOX
      };

    -- historic states: all interruptible basic states that can be subject to LTH
    history: array 0..3 of {
      state_none,
      in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST,
      in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_0, in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_5,
      in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-UPD_TIMER,
      in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-TOTAL_QUANTUM,
      in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-RESCHED,
      in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-XCPU_MAILBOX
    };


ASSIGN

  -- \DELTA P_CONF : STATES / BASIC CONFIGURATIONS - START

  init(state) := case
    pc = pc_INT_XCPU : in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI;
    pc = pc_INT_TIMER : in_KERNEL-INTERRUPT-TIMER-EOI_APIC;
    pc = pc_INT_IO_0 | pc = pc_INT_IO_5 : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;
    1 : state_none;
    esac;

  next(state) := case
```

```
-- Kernel Space

-- Submachine XCPU_MAILBOX - START
--
state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI : in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST;

state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST & (pe = pe_none | pe = pe_INT_XCPU |
  (pe = pe_INT_TIMER & (pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
  pcs[3] = pc_INT_TIMER)) |
  (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
  pcs[3] = pc_INT_IO_0)) |
  (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5))) :
  in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET;

state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST &
  (pe = pe_INT_TIMER & !(pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
  pcs[3] = pc_INT_TIMER))
  : in_KERNEL-INTERRUPT-TIMER-EOI_APIC;
state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST &
  ((pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
  pcs[3] = pc_INT_IO_0)) |
  (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5)))
  : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;

state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET & hist != state_none : hist;
state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET & hist = state_none : in_USER-HANDLERS-IRQ_THREAD;

--
-- Submachine XCPU_MAILBOX - END

-- Submachine HW_IRQ - START
--
state = in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK : in_KERNEL-INTERRUPT-HW_IRQ-EOI;

state = in_KERNEL-INTERRUPT-HW_IRQ-EOI & pc = pc_INT_IO_0 : in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_0;
state = in_KERNEL-INTERRUPT-HW_IRQ-EOI & pc = pc_INT_IO_5 : in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_5;

state = in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_0 & (pe = pe_none | pe  = pe_INT_IO_0 |
  (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5)) |
  (pe = pe_INT_TIMER & (pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
  pcs[3] = pc_INT_TIMER)) |
  (pe = pe_INT_XCPU & (pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
  pcs[3] = pc_INT_XCPU))) :
  in_KERNEL-INTERRUPT-HW_IRQ-IRET;

state = in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_0 &
  (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5))
  : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;
state = in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_0 &
 (pe = pe_INT_TIMER & !(pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
  pcs[3] = pc_INT_TIMER))
  : in_KERNEL-INTERRUPT-TIMER-EOI_APIC;
state = in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_0 &
  (pe = pe_INT_XCPU & !(pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
  pcs[3] = pc_INT_XCPU))
  : in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI;

state = in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_5 & (pe = pe_none | pe  = pe_INT_IO_5 |
  (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
  pcs[3] = pc_INT_IO_0)) |
  (pe = pe_INT_TIMER & (pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
```

```
    pcs[3] = pc_INT_TIMER)) |
    (pe = pe_INT_XCPU & (pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
    pcs[3] = pc_INT_XCPU)))
    : in_KERNEL-INTERRUPT-HW_IRQ-IRET;

  state = in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_5 &
    (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
    pcs[3] = pc_INT_IO_0))
    : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;
  state = in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_5 &
    (pe = pe_INT_TIMER & !(pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
    pcs[3] = pc_INT_TIMER))
    : in_KERNEL-INTERRUPT-TIMER-EOI_APIC;
  state = in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_5 &
    (pe = pe_INT_XCPU & !(pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
    pcs[3] = pc_INT_XCPU))
    : in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI;

  state = in_KERNEL-INTERRUPT-HW_IRQ-IRET & hist != state_none : hist;
  state = in_KERNEL-INTERRUPT-HW_IRQ-IRET & hist = state_none : in_USER-HANDLERS-IRQ_THREAD;

  --
  -- Submachine HW_IRQ - END

  -- Submachine TIMER - START
  --
  state = in_KERNEL-INTERRUPT-TIMER-EOI_APIC :
    in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-UPD_TIMER;

  state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-UPD_TIMER &
    (pe = pe_none | pe = pe_INT_TIMER |
    (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
    pcs[3] = pc_INT_IO_0)) |
    (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
    pcs[3] = pc_INT_IO_5)) |
    (pe = pe_INT_XCPU & (pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
    pcs[3] = pc_INT_XCPU))) :
    in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-XCPU_MAILBOX;

  state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-UPD_TIMER &
    (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
    pcs[3] = pc_INT_IO_0))
    : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;
  state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-UPD_TIMER &
    (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
    pcs[3] = pc_INT_IO_5))
    : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;
  state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-UPD_TIMER &
    (pe = pe_INT_XCPU & !(pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
    pcs[3] = pc_INT_XCPU))
    : in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI;

  state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-XCPU_MAILBOX &
    (pe = pe_none | pe = pe_INT_TIMER |
    (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
    pcs[3] = pc_INT_IO_0)) |
    (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
    pcs[3] = pc_INT_IO_5)) |
    (pe = pe_INT_XCPU & (pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
    pcs[3] = pc_INT_XCPU)))
    : in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-TOTAL_QUANTUM;

state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-XCPU_MAILBOX  &
  (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
   pcs[3] = pc_INT_IO_0))
```

```
  : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;
state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-XCPU_MAILBOX  &
  (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5))
  : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;
state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-XCPU_MAILBOX  &
  (pe = pe_INT_XCPU & !(pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
  pcs[3] = pc_INT_XCPU))
  : in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI;


state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-TOTAL_QUANTUM &
  (pe = pe_none | pe = pe_INT_TIMER |
  (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
  pcs[3] = pc_INT_IO_0)) |
  (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5)) |
  (pe = pe_INT_XCPU & (pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
  pcs[3] = pc_INT_XCPU)))
  : in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-RESCHED;

state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-TOTAL_QUANTUM   &
  (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
  pcs[3] = pc_INT_IO_0))
  : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;

state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-TOTAL_QUANTUM   &
  (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5))
  : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;
state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-TOTAL_QUANTUM   &
  (pe = pe_INT_XCPU & !(pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
  pcs[3] = pc_INT_XCPU))
  : in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI;


state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-RESCHED &
  (pe = pe_none | pe = pe_INT_TIMER |
  (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
  pcs[3] = pc_INT_IO_0)) |
  (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5)) |
  (pe = pe_INT_XCPU & (pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
  pcs[3] = pc_INT_XCPU)))
  : in_KERNEL-INTERRUPT-TIMER-IRET;

state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-RESCHED &
  (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
  pcs[3] = pc_INT_IO_0))
  : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;
state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-RESCHED &
  (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
  pcs[3] = pc_INT_IO_5))
  : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;
state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-RESCHED &
 (pe = pe_INT_XCPU & !(pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
 pcs[3] = pc_INT_XCPU))
 : in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI;

 state = in_KERNEL-INTERRUPT-TIMER-IRET & hist != state_none: hist;
 state = in_KERNEL-INTERRUPT-TIMER-IRET & hist = state_none: in_USER-HANDLERS-IRQ_THREAD;

 --
 -- Submachine TIMER - END
```

```
  -- User Space
  --
  state = in_USER-HANDLERS-IRQ_THREAD & (pe = pe_none |
    (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
    pcs[3] = pc_INT_IO_0)) |
    (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
    pcs[3] = pc_INT_IO_5)) |
    (pe = pe_INT_TIMER & (pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
    pcs[3] = pc_INT_TIMER)) |
    (pe = pe_INT_XCPU & (pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
    pcs[3] = pc_INT_XCPU)))
    : in_USERSPACE;

  state = in_USER-HANDLERS-IRQ_THREAD &
    (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
    pcs[3] = pc_INT_IO_0))
    : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;
  state = in_USER-HANDLERS-IRQ_THREAD &
    (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
    pcs[3] = pc_INT_IO_5))
    : in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK;
  state = in_USER-HANDLERS-IRQ_THREAD &
    (pe = pe_INT_XCPU & !(pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
    pcs[3] = pc_INT_XCPU))
    : in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI;
  state = in_USER-HANDLERS-IRQ_THREAD &
    (pe = pe_INT_TIMER & !(pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
    pcs[3] = pc_INT_TIMER))
    : in_KERNEL-INTERRUPT-TIMER-EOI_APIC;

  1 : state;
esac;

--
-- \DELTA P_CONF : STATES / BASIC CONFIGURATIONS
-- END

-----------------------------------------------------------------------------------------------------

-- \DELTA P_IMP : PATH CONDITIONS & HISTORY
-- START

-- PATH CONDITIONS
-- START
--
init(pcs[0]) := {pc_INT_XCPU, pc_INT_TIMER, pc_INT_IO_0, pc_INT_IO_5};
init(pcs[1]) := pc_none;
init(pcs[2]) := pc_none;
init(pcs[3]) := pc_none;

next(pcs[0]) := case
  counter = 0: case
    state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET : pc_none;
    state = in_KERNEL-INTERRUPT-HW_IRQ-IRET : pc_none;
    state = in_KERNEL-INTERRUPT-TIMER-IRET : pc_none;

    -- ... <-- analogous for the other interrupting path events
    -- pe_INT_IO_5, pe_INT_TIMER, pe_INT_XCPU
    state =  in_USER-HANDLERS-IRQ_THREAD    &
      (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
      pcs[3] = pc_INT_IO_0)) : pc_INT_IO_0;
    -- ... -->

    1 : pcs[0];
    esac;
```

```
  1: pcs[0];
  esac;

-- ... <-- analogous for pcs[2] and pcs[3]
next(pcs[1]) := case
  counter = 0 : case

    -- ... <-- analogous for all other interruptible states and their interrupting path events
    state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST &
      (pe = pe_INT_TIMER & !(pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
      pcs[3] = pc_INT_TIMER)) : pc_INT_TIMER;
    state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST &
      (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
      pcs[3] = pc_INT_IO_0)) : pc_INT_IO_0;
    state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST &
      (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
      pcs[3] = pc_INT_IO_5)) : pc_INT_IO_5;
    -- ... -->

    1 : pcs[1];
    esac;

  counter = 1 : case
    state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET : pc_none;
    state = in_KERNEL-INTERRUPT-HW_IRQ-IRET : pc_none;
    state = in_KERNEL-INTERRUPT-TIMER-IRET : pc_none;

    -- when an interrupt is in user space handling, it does not block any field in the pcs
    state =  in_USER-HANDLERS-IRQ_THREAD    &
      (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
      pcs[3] = pc_INT_IO_0)) : pc_INT_IO_0;
    state =  in_USER-HANDLERS-IRQ_THREAD    &
      (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
      pcs[3] = pc_INT_IO_5)) : pc_INT_IO_5;
    state = in_USER-HANDLERS-IRQ_THREAD &
      (pe = pe_INT_TIMER & !(pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
      pcs[3] = pc_INT_TIMER)) : pc_INT_TIMER;
    state =  in_USER-HANDLERS-IRQ_THREAD   &
      (pe = pe_INT_XCPU & !(pcs[0] = pc_INT_XCPU | pcs[1] = pc_INT_XCPU | pcs[2] = pc_INT_XCPU |
      pcs[3] = pc_INT_XCPU)) : pc_INT_XCPU;

    1 : pcs[1];
    esac;
  1: pcs[1];
  esac;
-- ... -->


init(pc) := pcs[0];
next(pc) := case
    -- ... <-- analogous for all other interruptible states and their interrupting path events
    state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST &
      (pe = pe_INT_TIMER & !(pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
      pcs[3] = pc_INT_TIMER)) : pc_INT_TIMER;
    state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST &
      (pe = pe_INT_IO_0 & !(pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
      pcs[3] = pc_INT_IO_0)) : pc_INT_IO_0;
    state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST &
      (pe = pe_INT_IO_5 & !(pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
      pcs[3] = pc_INT_IO_5)) : pc_INT_IO_5;
    -- ... -->

     -- recursive ascent
    state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET & counter = 1 : pcs[0];
    state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET & counter = 2 : pcs[1];
```

```
    state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET & counter = 3 : pcs[2];

    state = in_KERNEL-INTERRUPT-HW_IRQ-IRET & counter = 1 : pcs[0];
    state = in_KERNEL-INTERRUPT-HW_IRQ-IRET & counter = 2 : pcs[1];
    state = in_KERNEL-INTERRUPT-HW_IRQ-IRET & counter = 3 : pcs[2];

    state = in_KERNEL-INTERRUPT-TIMER-IRET & counter = 1 : pcs[0];
    state = in_KERNEL-INTERRUPT-TIMER-IRET & counter = 2 : pcs[1];
    state = in_KERNEL-INTERRUPT-TIMER-IRET & counter = 3 : pcs[2];

    1 : pc;
  esac;

init(prevpc) := pc_none;
next(prevpc) := case
  -- ... <-- analogous for all other interruptible states and counters 0..2
  state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST & !(pe = pe_none | pe = pe_INT_XCPU |
    (pe = pe_INT_TIMER & (pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
    pcs[3] = pc_INT_TIMER)) |
    (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
    pcs[3] = pc_INT_IO_0)) |
    (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
    pcs[3] = pc_INT_IO_5)))
    & counter = 2 : pcs[2];

  -- recursive ascent
  state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET & counter = 1 : pc_none;
  state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET & counter = 2 : pcs[0];
  state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET & counter = 3 : pcs[1];

  state = in_KERNEL-INTERRUPT-HW_IRQ-IRET & counter = 1 : pc_none;
  state = in_KERNEL-INTERRUPT-HW_IRQ-IRET & counter = 2 : pcs[0];
  state = in_KERNEL-INTERRUPT-HW_IRQ-IRET & counter = 3 : pcs[1];

  state = in_KERNEL-INTERRUPT-TIMER-IRET & counter = 1 : pc_none;
  state = in_KERNEL-INTERRUPT-TIMER-IRET & counter = 2 : pcs[0];
  state = in_KERNEL-INTERRUPT-TIMER-IRET & counter = 3 : pcs[1];

  1 : prevpc;
esac;


init(counter) := 0;
next(counter) := case
  -- ... <-- analogous for all other interruptible states except user space handling
  state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST & !(pe = pe_none | pe = pe_INT_XCPU |
    (pe = pe_INT_TIMER & (pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
    pcs[3] = pc_INT_TIMER)) |
    (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
    pcs[3] = pc_INT_IO_0)) |
    (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
    pcs[3] = pc_INT_IO_5)))
    & counter < 3 : counter+1;

  -- recursive ascent
  (state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET |
   state = in_KERNEL-INTERRUPT-HW_IRQ-IRET |
   state = in_KERNEL-INTERRUPT-TIMER-IRET) & counter > 0 : (counter + (-1));

  1 : counter;
esac;

--
-- PATH CONDITIONS - END
```

```
-- HISTORY - START
--

init(hist) := history[0];
next(hist) := case
  -- ... <-- analogous for all other interruptible states except user space handling
  state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST & !(pe = pe_none | pe = pe_INT_XCPU |
    (pe = pe_INT_TIMER & (pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
    pcs[3] = pc_INT_TIMER)) |
    (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
    pcs[3] = pc_INT_IO_0)) |
    (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
    pcs[3] = pc_INT_IO_5)))
    : in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST;

  -- recursive ascent
 (state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET |
   state = in_KERNEL-INTERRUPT-HW_IRQ-IRET |
   state = in_KERNEL-INTERRUPT-TIMER-IRET) & counter = 1 : state_none;
 (state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET |
   state = in_KERNEL-INTERRUPT-HW_IRQ-IRET |
   state = in_KERNEL-INTERRUPT-TIMER-IRET) & counter = 2 : history[0];
 (state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET |
   state = in_KERNEL-INTERRUPT-HW_IRQ-IRET |
   state = in_KERNEL-INTERRUPT-TIMER-IRET) & counter = 3 : history[1];

 1 : hist;
 esac;


init(history[0]) := state_none;
init(history[1]) := state_none;
init(history[2]) := state_none;
init(history[3]) := state_none;

-- ... <-- analogous for history[1] .. history[2]
next(history[0]) := case
  counter = 0: case
    -- ... <-- analogous for all other interruptible states except user space handling
    state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST & !(pe = pe_none | pe = pe_INT_XCPU |
      (pe = pe_INT_TIMER & (pcs[0] = pc_INT_TIMER | pcs[1] = pc_INT_TIMER | pcs[2] = pc_INT_TIMER |
      pcs[3] = pc_INT_TIMER)) |
      (pe = pe_INT_IO_0 & (pcs[0] = pc_INT_IO_0 | pcs[1] = pc_INT_IO_0 | pcs[2] = pc_INT_IO_0 |
      pcs[3] = pc_INT_IO_0)) |
      (pe = pe_INT_IO_5 & (pcs[0] = pc_INT_IO_5 | pcs[1] = pc_INT_IO_5 | pcs[2] = pc_INT_IO_5 |
      pcs[3] = pc_INT_IO_5)))
      : in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST;
    -- ... -->

    1 : history[0];
    esac;

  counter = 1: case
    (state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-IRET |
     state = in_KERNEL-INTERRUPT-HW_IRQ-IRET |
     state = in_KERNEL-INTERRUPT-TIMER-IRET) : state_none;

    1 : history[0];
    esac;

  1: history[0];
  esac;

next(history[3]) := history[3];
```

```
--
-- HISTORY - END

--
-- \DELTA P_IMP : PATH CONDITIONS & HISTORY -- END

-----------------------------------------------------------------------------------------------

-- \DELTA P_EXP : CONDITIONS & VARIABLES - START
-- NONE
-- \DELTA P_EXP : CONDITIONS & VARIABLES - END

-----------------------------------------------------------------------------------------------

-- FAIRNESS CONSTRAINTS
--
JUSTICE pcs[0]!=pc_INT_XCPU & pcs[1]!=pc_INT_XCPU & pcs[2]!=pc_INT_XCPU & pcs[3]!=pc_INT_XCPU;
JUSTICE pcs[0]!=pc_INT_IO_0 & pcs[1]!=pc_INT_IO_0 & pcs[2]!=pc_INT_IO_0 & pcs[3]!=pc_INT_IO_0;
JUSTICE pcs[0]!=pc_INT_IO_5 & pcs[1]!=pc_INT_IO_5 & pcs[2]!=pc_INT_IO_5 & pcs[3]!=pc_INT_IO_5;
JUSTICE pcs[0]!=pc_INT_TIMER & pcs[1]!=pc_INT_TIMER & pcs[2]!=pc_INT_TIMER & pcs[3]!=pc_INT_TIMER;


--
-- INDICATORS
--

-- infinite handling
LTLSPEC
  G((state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI | state = in_KERNEL-INTERRUPT-TIMER-EOI_APIC |
  state = in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK) -> F(state = in_USERSPACE));

LTLSPEC
  G((state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI | state = in_KERNEL-INTERRUPT-TIMER-EOI_APIC |
  state = in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK) -> F(state = in_USER-HANDLERS-IRQ_THREAD));


-- losing interrupts
LTLSPEC
   G((state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-EOI & pc = pc_INT_XCPU)
   -> F(state = in_KERNEL-INTERRUPT-XCPU_MAILBOX-HANDLE_XCPU_REQUEST));
LTLSPEC
   G((state = in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK & pc = pc_INT_IO_0)
   -> F(state = in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_0));
LTLSPEC
   G((state = in_KERNEL-INTERRUPT-HW_IRQ-SW_MASK & pc = pc_INT_IO_5)
   -> F(state = in_KERNEL-INTERRUPT-HW_IRQ-HANDLE_HW_IRQ_5));
LTLSPEC
   G((state = in_KERNEL-INTERRUPT-TIMER-EOI_APIC & pc = pc_INT_TIMER)
   -> F(state = in_KERNEL-INTERRUPT-TIMER-HANDLE_TIMER_INTERRUPT-SCHEDULER-RESCHED));
```

Appendix C

List of Used Abbreviations

ACPI	Advanced Configuration and Power Interface
ALU	Arithmetical Logical Unit
AOCS	Attitude and Orbital Control System
AP	Atomic Proposition
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
AS	Architectural State
BCT	Basic Compound Transition
BDD	Binary Decision Diagrams
BKL	Big Kernel Lock
BSD	Berkley Software Distribution
BSP	Bootstrap Processor
CAD	Computer Aided Design
CE	Component Extension
CISC	Complex Instruction Set Computing
CCS	Calculus of Communicating Systems
CPL	Current Privilege Level
CPU	Central Processing Unit
CSP	Communication Sequential Processes
CT	Compound Transition
CTL	Branching Time Computation Tree Logic
CTL*	Computation Tree Logic
CTS	Cartesian Transition Set
CTSC	Cartesian Transition Set Connector
DEVT	Disruptive Event
DFA	Deterministic Finite Automaton
DFS	Depth First Search
DNA	Device Not Available
EB	Event Bus
EBC	Event Bus Connector
EBS	Event Bus Pseudo-State
EDT	Estelle Development Toolset

EHA	Extended Hierarchical Automaton
EOI	End Of Interrupt
ESF	Engineering Statechart Formalism
FCT	Full Compound Transition
FDR	Failures Divergences Refinement
FDT	Formal Description Technique
FIFO	First In First Out
FPE	Floating Point Errors
FPU	Floating Point Unit
FT	Full Transition
GB	GigaByte
GMD	Gesellschaft für Datenverarbeitung
GNU	GNU is Not Unix
GPOS	General Purpose Operating System
HCPN	Hierarchical Colored Petri Net
HDD	Hard Disc Drive
HID	Human Interface Device
HPET	High Precision Event Timers
HRM	Hierarchic Reactive Modules
HRT	Hard Real-Time
HWEB	Heavy-Weight Event Bus
IA	Intel Architecture
IAG	Interruptibility Analysis Graph
ICR	Interrupt Command Register
ICT	Initial Compound Transition
IDT	Interrupt Descriptor Table
IKCP	Intermission Kernel Control Path
INT	Interrupt
IOAPIC	Input Output Advanced Programmable Interrupt Controller
IPC	Inter-Process Communication
IPI	Inter-Processor Interrupt
IRQ	Interrupt Request
IRT	Interrupt Redirection Table
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
ITU-T	International Telecommunication Union
IHF	Interrupt Handling Facility
ISO	International Standardization Organization
KB	Keyboard
KCP	Kernel Control Path
KIT	Kernel for Isolated Tasks
LAPIC	Local Advanced Programmable Interrupt Controller
LCA	Lowest Common Ancestor
LOTOS	Language Of Temporal Ordering Specification
LTH	Long-Term History

LTL	Linear Time Logic
LTS	Labeled Transition System
LWEB	Light-Weight Event Bus
MC	Model Checking
MHz	MegaHertz
MIPS	Microprocessor without Interlocked Pipeline Stages
ML	Meta Language
MOF	Meta Object Facility
MOSRTOS	Multimedia-Oriented Soft Real-Time Operating System
MPEG	Moving Picture Experts Group
MTL	Metric Temporal Logic
NIC	Network Interface Card
NMR	Non-Maskable Interrupt
NFA	Non-Deterministic Finite Automaton
OBBD	Ordered Binary Decision Diagrams
PAG	Path Analysis Graph
PC	Path Condition
PCI	Peripheral Component Interconnect
PIC	Programmable Interrupt Controller
PLTL	Parametric Temporal Logic
PIT	Programmable Interval Timer
POSIX	Portable Operating System Interface
PPR	Processor Priority Register
PROMELA	Process Meta Language
P/T	Place Transition System
RISC	Reduced Instruction Set Computing
RS	Recursion Scope
RSC	Remote System Control
RT	Real-Time
RTC	Real Time Clock
OS	Operating System
SDL	Specification and Description Language
SEREXP	Sequential Extended Regular Expressions
SMP	Symmetric Multi-Processor
SMV	Symbolic Model Checker
SND	Sound Card
SOS	Structural Operational Semantics
SPARC	Scalable Processor Architecture
SRT	Soft Real-Time
STD	Symbolic Timing Diagrams
SuD	System under Development
SuI	System under Investigation
SVG	Scalable Vector Graphics
TLB	Translation Look Aside Buffer
TLMC	Temporal Logic Model Checking

TPR	Task Priority Register
TS	Transition Segment
TSC	Time Stamp Counter
TUF	Time Utility Function
UML	Unified Modeling Language
USB	Universal Serial Bus
VAL	Value Function
VIS	Visual Instruction Set
VLSI	Very-Large-Scale Integration
VHSIC	Very-Large-Scale Integration Circuits
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XOR	Exclusive Or
XSCF	Extended System Control Facility

Appendix D

Contents of the Archive

The archive belonging to this thesis contains the following files:

Dissertation/dissertation.pdf	Dissertation in PDF format
Dissertation/dissertation.ps	Dissertation in PS format
Linux/model.smv	SMV Model of Linux
OpenBSD/model.smv	SMV Model of OpenBSD
OpenBSD/Thesis.pdf	Bachelor Thesis Gogolok
Pistachio/model.smv	SMV Model of Pistachio
Pistachio/Thesis.pdf	Bachelor Thesis Wieder

It is available for download at http://www.koenen-dresp.de/research-alex/

List of Figures

List of Tables