Verification in the Hierarchical Development of Reactive Systems

Inauguraldissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von
Dipl.-Ing.(FH) Frank Salger
aus Reutlingen

Mannheim, 2001

Dekan:	Professor Dr. Guido Moerkotte, Universität Mannheim
Referentin:	Professor Dr. Mila Majster-Cederbaum, Universität Mannheim
Korreferent:	Professor Dr. Franz Stetter, Universität Mannheim

Tag der mündlichen Prüfung: 21. Mai 2001

Contents

List of Figures

1 Introduction

"Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in a given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning."
C. A. R. Hoare

"The notion of program verification appears to trade upon an equivocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility."
J. H. Fetzer

To communicate means to interchange information. A *language* is a collection of strings (or sentences), built up from basic letters according to particular rules. To communicate such strings is uninteresting unless these strings refer to objects of particular worlds or *domains*. Languages are named according to the objects their strings refer to. We talk about *natural languages* if we mean languages whose strings refer to objects of the "real, physical world". For example, English is a natural language: The string "dog", consisting of the three basic letters "d", "o" and "g", most commonly refers to an animal with four legs which can bark.

On the other hand, a *programming language* can be viewed as a means to communicate algorithmic ideas (programs) between people and machines. For example, the program P given by the text

$$readln(x);\ y := x + 1;\ writeln(y);$$

can be seen as an algorithmic representation of the successor function $succ : \mathbb{N} \to \mathbb{N},\ x \mapsto x + 1$. Based on our intuitive interpretation of the basic program-letters[1], the program P is thus related to an object of a *mathematical domain*. The successor

[1]For example, the intuitive interpretation of the program-letter "+" would be the binary addition function.

function *succ* can thus be viewed as the meaning of the program P. When used in order to provide interpretations (meanings) for programming languages, mathematical domains are called *semantic domains*, *semantic models* or simply *semantics* of the according programming language. On the other hand, the programs (program texts) that constitute the programming language are called the *syntax* of the programming language. Defining the semantics of a programming language amounts to choose a semantic domain and to systematically connect the syntax with this semantic domain, that is, to systematically "link" every program with one element of the semantic domain considered. A program is then said to denote the (semantic) object it is linked to or to be the *denotation* of this object.

The successor function above could also be represented by a formula φ of a suitable *logic*[2]. Both, the program P and the formula φ can be conceived as descriptions of the successor function. The difference between these two descriptions is, that the program P is given in a way, "understandable" and thus executable by a *machine*. The program P is sometimes called an *implementation* of the successor function whereas the formula φ is called a *specification* of this function. Some computer scientists prefer to use the phrase "specification" for programs as well, differing from logical specifications only in the level of abstraction (see for example [3, 126]). The program P is then often called an *executable specification* as it resides at a lower (machine oriented) level of abstraction than the formula φ.

Programs are usually developed with the intention to delegate work to machines. Instead of calculating a given problem "by hand", one develops a program and, by executing the program on a machine, aims to obtain solutions to the problem. However, the outputs returned by the machine should be "correct", that is, the outputs should actually be solutions to the problem. Ideally, one would like to have a *guarantee* that the machine *always* calculates the *correct* solutions to the problem. Such a guarantee however, could only be achieved on the basis of rigorous mathematical reasoning. As will be seen in the next section, giving a mathematical proof that a program, executed on a physically existing machine, always yields the correct answers is condemned to be impossible.

[2]For example by a formula of first-order predicate logic.

1.1 What Can be Achieved by Formal Verification

"[...], from a realistic perspective rather then a formalist, there seems to be a divide between the concrete physical objects that populate the physical world and the abstract objects about which we reason in mathematics."
J. Barwise

Physical machines versus abstractions of physical machines. Being an element of the "real, physical world", a *physical machine* (and thus the execution of a program on this machine) is subject to any physical influence exerted by its environment, whether these are lightning strikes, air humidity or whatsoever. The complexity of the sum of these influences entirely exceeds the limits of mathematical representation.

In order to address this problem, we can look for a means to abstract from this mathematically intractable complexity exhibited by physical machines. This can be done in several ways:

- One can mimic physical machines by abstract machines which are mathematical objects, equipped with components such as states, actions (the execution of which might cause state changes), and maybe others. For example, *transition systems* [114] are abstract machines which are often used to *model* physical machines: Programs for a physical machine are divided into "blocks" which are then abstractly represented by elementary actions of a transition system. Abstract machines often arise in a particular type of semantics, called *operational semantics* [172]. This type of semantics abstractly reflects what would happen when programs were executed on the according physical machine.

- Alternatively, one can relate programs to mathematical objects such as sets, functions and operators. This gives rise to so called *denotational semantics* (see for example [184]). Programs are interpreted as functions from initial states to final states. As opposed to operational semantics, the concept of intermediate computation steps is not employed.

- Using *axiomatic semantics*, one relates programs to assertions of a suitable logical framework. The *Hoare-logic* [101] for sequential while-programs is probably the most famous such semantics. There, an assertion of the form $\{p\}S\{q\}$

means, that the program S satisfies a property q if it terminates, provided it started in a state where the property p holt. The program S can also be viewed as a *predicate transformer* that maps a (final state) predicate q to the weakest (initial state) predicate p such that the Hoare-assertion $\{p\}S\{q\}$ is valid [64].

The choice of an appropriate semantic domain will depend on those aspects of physical machines one is interested to model: Whereas operational semantics closer reflect what actually happens while executing a program on a physical machine and are thus closer to the "intuitive semantics" of a programming language, they might not be abstract enough for some purposes. While denotational semantics are more abstract then their operational counterparts, the mathematical theory which underpins them might be more difficult to access. An important feature of denotational semantics (which is not always enjoyed by operational semantics) is, that they are *compositional*. This means that the semantics of a program P is determined by the semantics of the constituent parts of P. Finally, axiomatic semantics are well suited for reasoning about properties of programs.

Regardless of the relative advantages and drawbacks of the types of semantics discussed above, they share one important feature: If being defined with care, they can provide unambiguous and consistent interpretations of many interesting programming languages[3]. This is a necessary prerequisite for any meaningful reasoning about programs. Still, it is important to keep in mind the difference between the physical machine or the *physical system* and the mathematical object used to model it:

> "[...] it is only the *model* which is verified; how well this verification serves as a predictor for the behaviour of the computer *system* depends upon the extend to which the model accounts for behaviours of the computer, including its interactions with all the software which it runs, as well as other parameters such as temperature and gamma rays."
> [124, page 3]

[3]This is in contrast to natural languages: The string "dog" usually refers to the animal mentioned earlier but is sometimes also used to address a bad fellow. Hence, statements containing the word "dog" might be interpreted differently according to varying social contexts.

The world of our minds versus the world of formal specifications. The intuition of a problem whatsoever always resides in our minds whereas a formal (hence mathematically expressed) specification of the problem necessarily lives in the mathematical world. A provably correct bridge connecting these two worlds cannot exist. For example, think of a problem which is presented to an engineer by another person or through a requirement analysis. The first thing he (a women might do better in what follows) will do, is to try to access the problem itself. However, he might come up with an intuition of the problem which does not match the actual problem. In this case, any formal specification proposed by the engineer is doomed to be wrong. If his intention only captures parts of the problem, he will end up with an *under-specification* (as explained in [142, page 378]).

Note that even if the engineer succeeds in gaining the right intuition of the problem, he might not possess enough expertise in the specification formalism in order to correctly formalize his intuition[4].

In summary, what can be achieved at most is a guarantee (a mathematical proof) that a model of a physical system correctly solves a model of that problem we intend the physical system to solve. The activity of providing such a guarantee is called *formal verification* or simply *verification*. Verification has been severely criticized for being based on models of physical machines whence it does not allow to assert anything about the actual physical machines themselves [80]. The aforementioned article has stipulated an important and controversial discussion on the relative dangers and benefits of "verification" [157, 171] and the reader is encouraged to consult [13] for an objective discussion of these topics.

Being aware of the aforementioned "world-gaps" is important in order to understand what can be achieved by formal verification. However, yet another fundamental problem lurks in the mathematical world itself: The work of A. Church on *effective calculability* [39] and of A. M. Turing on *computability* [200] (see also the work of K. Gödel [79]) lead to the famous *Church-Turing-Thesis* (see for example [104, page 166]). A consequence of this thesis is, that there exist mathematically definable problems which cannot be solved by the most powerful machines we can imagine.

[4]Of course, the same problem arises in the development of programs: An engineer might possess an appropriate algorithmic solution to the problem in his mind but fail to capture his solution by say, a program in C++ or PASCAL.

The Church-Turing-Thesis implies, that there even exist mathematically definable statements about abstractions of physical machines that can neither be proved nor disproved in a purely formal and thus mechanical fashion. Hence, for particular properties, an automatic "verification-machine" can never succeed in showing that some other machine enjoys these properties[5]. These results have had a particularly strong impact to the field of *computer aided verification* [46]. Fortunately, many physical machines can be modelled by abstract machines for which computer aided verification and automatic verification is feasible.

In summary, the objective of any verification activity cannot be to show the correctness of physical systems. Even for particular abstract models of physical systems, verification remains infeasible.

1.2 Objectives of Reactive System Design Verification

> "The goal of research in program verification it to discover techniques for mathematically describing an algorithm so that conclusions drawn from formal deduction predict the behaviour of an execution of the algorithm on a physical machine with high degree of accuracy."
> *W. R. Bevier, M. K. Smith and W. D. Young*

Abstract models of physical systems capture the abstract logical structure of a physical system, that is, they embody what might be called the "abstract essence" of a physical system. This abstract essence is called the *design of a physical system* or the system design and developing the design of a physical system is the first step towards the final realization of the physical system itself. Whereas verification does not apply to physical systems, verification is very often feasible for designs of them. The verification of system designs is called *design verification.* It is well known, that the most serious failures of physical machines are caused by conceptual errors, that is, by errors which have already been made during the *design phase* [181]. Clearly, the most efficient way of uncovering and removing this kind of errors is to eliminate them during the design phase itself and not after the realization of the physical machine when the machines complexity becomes overwhelming [25, 24]. Moreover, traditional techniques such as *testing* are not applicable during the design phase as they are

[5]That is, as soon as the involved machines can simulate *Turing machines* [200].

applied to physical machines. Design verification is thus a valuable method which can help avoiding failures of physical systems that can cause tremendous expenses [78] or even human dead [131].

With the advent of systems that operate continuously while computing subtasks *concurrently*, design verification has become most important. Such systems, called *reactive computer systems* (*RCSs* for short) [94], are often employed for safety critical tasks like, for example, air traffic control or the supervision of nuclear power plants. Owing to the concurrent execution of different subsystems and the coordination and and *synchronization* issues involved, reactive systems are much more complex than purely *sequential systems* and thus extremely difficult to comprehend and design. In order to keep the procedure of designing and verifying *RCSs*-designs intellectually tractable, complying with the following proposals seems to be necessary.

- The formalisms, employed for creating and verifying *RCSs*-designs should be simple and accessible by system designers which in general are not particularly trained in mathematical concerns of computer science [48, 146].

- A programming language should be connected with the semantic model used. *Process algebras* like, for example, *CCS* [154] or *CSP* [102, 103] can be used as Programming languages for *RCSs*-designs. The constituent elements (programs) of process algebras are called *process terms.* A process algebra should comprise a few basic programming constructs that suffice to express those basic concepts of (physical) *RCSs* that we intend to model. Process algebras allow to compose complex designs of *RCSs* out of smaller designs in a syntactic way. Employing process algebras keeps the design procedure more tractable than developing designs directly in the semantic domain. More important however, process terms can sometimes be used to finitely denote *RCSs*-designs which could only be captured by an infinite object of the underlying semantic domain. Consequently, many operations which are are algorithmically feasible on process algebras could not be accordingly automated (in an effective way) for the underlying semantic domain. A verification technique however should be fully and efficiently automatizable thereby discharging system designers from the tedious and mathematically complicated details of verification [48, 53, 146].

- Redundantly expressing a system design in different types of design formalisms is called the *dual language approach* (see, for example, [96]). Typically an imperative/operational-based design formalism (like, for example, transition systems) is combined with a descriptive/property-based formalism (like, for example, logical frameworks). Verifying a design then amounts to show the consistency of the two redundant descriptions of this design. As described in [96], the dual language approach to verification has valuable advantages: Operational design descriptions are less likely to omit required attributes of the intended design and can often directly be coded into an executable program. On the other hand, property-based design descriptions are intuitive, concise and abstract in the sense, that they minimize unnecessary details as, for example, concerns about the precise architecture of physical reactive systems. A very successful dual language approach to $RCSs$-design verification (employing operational and logical design formalisms) is called *model checking* (see [46, 48] for an introduction) and will be discussed in Section 3.2.

- As the complexity of reactive system designs becomes overwhelming very quickly, methods which allow to develop designs in a hierarchical fashion must be supported by the design formalisms employed. Such methods allow to develop a design on different levels of abstraction[6] thereby making the development procedure more transparent and thus tractable: Most likely, a developer first divides the intended (complex) design into various "sub-designs" to capture the abstract overall structure of the complete design. Subsequently, the sub-designs will be developed by enriching them step by step with details. This is the design technique usually encountered in practice and, as argued in [186, 48, 53, 146, 10], design formalisms must support such typical design styles. In process algebraic settings, *syntactic action refinement* (surveyed in [90]), SAR for short, can be used for the hierarchical development of $RCSs$-designs. Intuitively, SAR means to refine an (atomic) action α occurring in a process term P by a more complex process term Q thereby yielding a more detailed process description $P[\alpha \leadsto Q]$ where $\cdot[\alpha \leadsto Q]$ denotes the SAR-operator.

[6]Please note, that in this case "abstraction" refers to mathematical objects (designs of reactive systems) and not to the kind of abstraction which is employed to model (to design) a physical reactive system.

1.3 Contributions of this Thesis

The obvious method to obtain correct designs of *RCSs* is as follows: First, we formalize those properties we wish the intended design to exhibit, that is, we devise a specification of the intended design. Subsequently, we invest experience and guess work to develop an according design. Finally, existing verifications techniques like, for example, model checking [46, 48] can be applied in order to show that the design satisfies the specification. However, adaptation of the system and subsequent verification has to be undergone repeatedly until the design meets the specification, a time consuming task. Another method (called "automated program synthesis") uses transformational methods to construct a (a priori correct) design directly from the specification [41, 145, 174, 11], thereby avoiding the need for an explicit verification.

However, the above methods implicitly assume, that the actual specification is indeed the desired one, and that subsequent changes of it will not become necessary. During a design phase however, specifications (and hence the according designs) are most often subject to repeated adaptations actuated by changed requirements or resources. Such changes also emerge in realistic design-scenarios where a specification is arrived at by successively enriching an initial specification with details.

It would thus be desirable to extend the above mentioned approaches in the following way: Once it has been proved that a design D satisfies a specification φ (denoted $D \models \varphi$), transforming φ into a modified specification φ' should entail a transformation of D into a design D' such that $D' \models \varphi'$. This paradigm supports a stepwise and *a priori correct development of RCSs-designs.* Reversely, *re-engineering* [215] amounts to the ability to infer $D \models \varphi$ from $D' \models \varphi'$. This allows to reuse verification knowledge that has already been gained through preceding steps in a development sequence.

The main contribution of this thesis is the introduction of a technique that supports a priori correct *RCSs*-design development and re-engineering. The technique is based on dual language approaches to verification: Designs of *RCSs* are expressed in an operational fashion by means of a *TCSP*-like process algebra equipped with a transition system semantics (see, for example, [162]). Specifications are formalized within a particular temporal logic, the *Modal Mu-Calculus* ($\mu\mathcal{L}$ for short) [119]. This logic is well suited for specifying designs of *RCSs* since it allows to formalize important properties of *RCSs*-designs like, for example, absence of deadlock. To support the hierarchical development of $RCSs$-designs, the method of syntactic action refinement, SAR (surveyed in [90]) is exploited. Intuitively, SAR means that actions α that occur in a process term P are refined by a more complex process term Q thereby yielding the more detailed process term $P[\alpha \rightsquigarrow Q]$. There, the process term Q is conceived as a detailed description of the abstract action α. In order to obtain the development/re-engineering-technique, SAR for formulas φ of the Modal Mu-Calculus $\mu\mathcal{L}$ will be defined in a way that conforms to SAR for $TCSP$-like process terms P (see Section 4.1). We show the validity of the assertion

$$\mathcal{T}(P) \models \varphi \text{ iff } \mathcal{T}(P[\alpha \rightsquigarrow Q]) \models \varphi[\alpha \rightsquigarrow Q] \quad (*)$$

where $\mathcal{T}(P)$ is the transition system induced by P and the operator $\cdot[\alpha \rightsquigarrow Q]$ denotes syntactic action refinement, both on process terms and formulas (see Section 4.2). The distinguishing features of this result are

- The use of SAR. This supports hierarchical development of infinite state transition systems[7]: As opposed to semantic action refinement, SAR is applied to process terms whence state spaces of transition systems do not have to be handled algorithmically to implement SAR. This allows to apply assertion $(*)$ to infinite state transition systems. The restriction to finite state transition systems would disallow the design of particular physical $RCSs$.
- Using assertion $(*)$, correctly developing (or adapting) a design with respect to adding details to the actual specification (or by changing it) boils down to "gluing" refinement operators to formulas and process terms. On the other hand, re-engineering amounts to replacing refinement operators, that is, to first "cutting away" inappropriate refinement operators (stepping backwards through a development sequence) and subsequently resuming the development procedure. This development/re-engineering-technique as illustrated by Figure 1 is developed in Section 4.2 and applied in Section 4.2.1.
- The SAR-operator implicitly supplies an *abstraction technique* that, by the syntactic nature of the SAR-operator, relates system descriptions. Again, this allows infinite state systems to be considered. Hence, if not applied in the

[7]Intuitively, a transition system is called an infinite state transition system if it has an infinite number of states (see also page 19).

context of developing/re-engineering designs of $RCSs$, assertion $(*)$ can still be useful to support model checking techniques: Many designs cannot be handled by model checking techniques due to the (huge or infinite) size of their state spaces. However, if we can find a process term P_s and a formula φ_s and establish an appropriate abstraction (induced by applications of the refinement operator), for example $P = P_s[\alpha_1 \rightsquigarrow Q_1]\ldots[\alpha_n \rightsquigarrow Q_n]$ and $\varphi = \varphi_s[\alpha_1 \rightsquigarrow Q_1]\ldots[\alpha_n \rightsquigarrow Q_n]$ then P_s might well be manageable by a model checker since the state space of P_s might be exponentially smaller then the state space of P due to the well known *state space explosion problem*[8]. We can then apply the model checker to decide $\mathcal{T}(P_s) \models \varphi_s$ and conclude via assertion $(*)$ whether $\mathcal{T}(P) \models \varphi$ holds or not. In Section 4.6, it will be discussed to what extend this abstraction technique can be fully automatized. An application of this abstraction technique can be found in Section 4.2.1.

- As the Modal Mu-Calculus subsumes many other logics [72, 58] used to specify designs of $RCSs$, we believe that our results provide a basis for similar investigations employing these logics and other semantics for concurrency.

Variations on the theme:

• Some particular conditions concerning the structure of the process terms Q have to be obeyed in order to guarantee that assertion $(*)$ is valid. These conditions can often be met by a simple and efficient restructuring of these terms. In Section 4.3, it will be shown however, that these conditions can even be dropped completely if particular fragments of $\mu\mathcal{L}$ are considered instead of the full logic. One interesting fragment of $\mu\mathcal{L}$ will be identified for which it is possible to guarantee the validity of the assertion

$$\mathcal{T}(P) \models \varphi \Rightarrow \mathcal{T}(P[\alpha \rightsquigarrow Q]) \models \varphi[\alpha \rightsquigarrow Q].$$

Further, a fragment for which the assertion

$$\mathcal{T}(P) \models \varphi \Leftarrow \mathcal{T}(P[\alpha \rightsquigarrow Q]) \models \varphi[\alpha \rightsquigarrow Q]$$

holds will be presented.

[8]A linear reduction of the number of actions in a process term might entail an exponential reduction of the underlying state space.

$$\begin{tikzcd}
\mathcal{T}(P) \arrow[d, leftrightarrow] & \models \arrow[d, Leftrightarrow] & \varphi \arrow[d, leftrightarrow] \\
\mathcal{T}(P[\alpha_1 \rightsquigarrow Q_1]) \arrow[d, leftrightarrow] & \models \arrow[d, Leftrightarrow] & \varphi[\alpha_1 \rightsquigarrow Q_1] \arrow[d, leftrightarrow] \\
\mathcal{T}(P[\alpha_1 \rightsquigarrow Q][\alpha_2 \rightsquigarrow Q_2]) & \models & \varphi[\alpha_1 \rightsquigarrow Q_1][\alpha_2 \rightsquigarrow Q_2] \\
\vdots & & \vdots
\end{tikzcd}$$

Figure 1: Developing/Re-engineering Correct Designs

• Exploiting several different levels of abstraction helps to devise complex process terms and formulas in a hierarchical, hence transparent way. However, physical *RCSs* can be constructed more conveniently from the according process terms if they are non-hierarchical. Hence, a *reduction function* (see, for example, [4, 88]) is used to collapse the different levels of abstraction that occur in a process term $P[\alpha \rightsquigarrow Q]$, yielding the process term $red(P[\alpha \rightsquigarrow Q])$. Roughly, the reduction function removes refinement operators from process expressions $P[\alpha \rightsquigarrow Q]$ by "syntactically replacing" every action α in P by the process term Q. Similarly, a *logical reduction function* is introduced to derive the non-hierarchical "low level"-specification $\mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$ from a (hierarchical) specification $\varphi[\alpha \rightsquigarrow Q]$. However, even for non-hierarchical formulas φ and process terms Q, the formula $\mathcal{R}ed(\varphi[a \rightsquigarrow Q])$ has size $O(2^{|\varphi|*|Q|})$ in the worst case (where $|\varphi|$ and $|Q|$ are the number of symbols that occur in φ and Q respectively).

The third contribution of this thesis is the introduction of a generalization of the Modal Mu-Calculus (referred to as $\mu\mathcal{L}_g$) in which such exponential blow up's does not occur (see Section 4.4). Instead of being restricted to atomic actions, modalities of $\mu\mathcal{L}_g$-formulas are generalized to contain more complex process terms, in a way similar to the logic *PDL* [82]. $\mu\mathcal{L}_g$-formulas allow to express properties of "subprocesses" of systems in a much more concise way than $\mu\mathcal{L}$-formulas. In Section 4.4, we define SAR for the logic $\mu\mathcal{L}_g$ (denoted by $\cdot[\alpha \rightsquigarrow Q]$) and show that an assertion as $(*)$ also holds for $\mu\mathcal{L}_g$-formulas. Further, the logical reduction function $\mathcal{R}ed_g$ for $\mu\mathcal{L}_g$ will be defined in a way, such that $\mathcal{R}ed_g(\varphi[a \rightsquigarrow Q])$ has size $O(|\varphi| * |Q|)$, provided φ and Q are non-hierarchical. Though many properties can be expressed more conveniently by using the logic $\mu\mathcal{L}_g$ instead of $\mu\mathcal{L}$, we give a transformation that takes $\mu\mathcal{L}_g$-formulas to logically equivalent $\mu\mathcal{L}$-formulas. This allows existing methods and tools for the Modal Mu-Calculus to be integrated into our framework.

Whereas classical methods like, for example, model checking can be used to carry out the horizontal verification task (that is, to prove or disprove $P \models \varphi$), the obtained results can be used to carry out vertical verification: The refined system $P[a \rightsquigarrow Q]$ is (a priori) correct with respect to the refined specification $\varphi[a \rightsquigarrow Q]$. This means that verification is for free if the hierarchical development of $RCSs$-designs is based on assertion $(*)$. At each stage of the development procedure, low level specifications and implementation near designs can be efficiently constructed via the reduction functions.

The obtained results are applied to a serial of examples and a more thorough case study is carried out in Section 4.2.1. Particular parts of this thesis are also available in a more condensed form (see [136, 137, 138, 139, 140]).

2 Designing Reactive Systems

Two types of computer systems can be identified (see for example [68, pp. 1048]): One class consists of so called *sequential computer systems* which typically solve scientific problems like, for example, calculate Fast Fourier Transformations, sort a list or compute a shortest path between two nodes of a graph. The computational behaviour common to all these computer systems is, that they accept input data (in some initial state), perform some calculations, and eventually terminate (in some final state) providing the according output data. *Semantics of sequential systems* can thus be defined in terms of relations between initial states and final states.

Many computer systems however, are required to operate continuously while maintaining an ongoing interaction with their environment. Such computer systems are called *reactive computer systems* (*RCSs* for short) or *reactive systems* [94] (see also [143, 144]). Representatives of this class of computer systems are operating systems, communication protocols, and control systems. Most often, *RCSs* exhibit the ability to compute various subtasks concurrently in order to comply with the requirements of the overall task. For example, the control system of a nuclear power plant must be able to quickly calculate whether some measured reactor data violates some defined safety requirements in order to shut down the reactor but at the same time continue to observe the reactor. Though it has been observed that "concurrency" and "reactivity" are system features, in some sense independent from each other [68, page 1049], it has been argued to conceive concurrent computer system as *RCSs* [68, page 1049][173]. This view is typically adopted in the literature, that is, the ability of the concurrent execution of subsystems is implicitly attributed to *RCSs*.

As opposed to sequential computer systems, termination of *RCSs* is not required but instead most often amounts to an abnormal end of the system. The definition of reasonable semantics for *RCSs* can thus not relay on relations between initial and final system states. Instead, *RCSs* can be modelled on the basis of this information about *RCSs* that can be gathered during their execution [154, page 12].

2.1 Semantic Domains for Reactive System Designs

When modelling *RCSs* one has to decide which aspects of physical *RCSs* should be considered relevant for the purposes at hand. The decision for a particular semantics has to be take carefully in order to allow the faithful reflection of the relevant aspects of physical *RCSs* by their models. To put it the other way round, the decision for a particular semantics determines which aspects of *RCSs* are negligible and can thus be abstracted by the modelling process.

A rough *classification of semantics for RCSs* can be given with respect to three parameters [159, 183]: "*behaviour versus system*", "*interleaving versus true concurrency*" and "*linear time versus branching time*". These parameters correspond to the level of abstraction at which the *RCSs* are examined. Whereas semantics of type "system"allow an explicit representation of system states, a behaviour semantics abstract from such information. "True concurrency" semantics distinguish between the concepts of concurrency and *nondeterminism* whereas an "interleaving"-semantics identifies them. Finally, a "linear time"-semantics abstracts from the choice points that might occur in computations whereas a "branching time" semantics takes informations about such points into account. Further distinctions between semantics for *RCSs* can be taken with respect to the extent to which *internal behaviour* (that is, concerns about *internal actions*) of *RCSs* is taken into account [203]. A classification of various standard semantics for *RCSs* according to the parameters "behaviour/system", "true concurrency/interleaving" and "linear time/branching time" can be found in [159, 183].

The relationships between various semantics for *RCSs* have been investigated, for example, in [202, 203, 159, 183]. In the present thesis, the semantic domain of labelled transition systems is used to model *RCSs* whence they are described more precisely in the next section.

2.1.1 Transition System Semantics

"Everything really moves continuously. But there are many kinds of machine which can profitably be *thought of* as being discrete-state machines."
A. M. Turing

Many physical systems can be appropriately modelled as objects that dynamically evolve through discrete points in time by performing particular activities. For example, the activity of preparing a cup of tea consists of boiling a kettle of water, putting some tea leaves into a mug, and subsequently pouring the hot water into the mug. In reality, the constituent activities of preparing a cup of tea exhibit a continuous nature. However, we can model them by non-interruptible, instantaneous *atomic actions.* As a consequence we abstract from the continuous nature of the constituent activities [9]. Abstracting from the duration and exact timing of activities by modelling them via atomic actions has the advantage, that the resulting model will be independent of the speed and performance of the person (or the oven) involved in preparing the tea (see also [103, page 24]). For a "computer system"-example this amounts to independence of clock rates or the rate of microinstructions per second.

As an example, we can model the activity of boiling the water by the atomic action "*boil_water*" thereby also abstracting from constituent activities of boiling the water (like, for example, filling water into the kettle). Accordingly, we can employ the atomic actions "*put_leaves*" and "*pour_water*" in order to model the activities of putting the leaves into the mug and pouring water into the mug. Putting the leaves into the mug before pouring the water into it seems to be reasonable. Also, we wont pour the water into the mug unless it is hot. Boiling the water is independent from putting the leaves into the mug. Hence, we model these two activities as concurrent activities. As we abstract from the exact timing of actions, it seems to be reasonable to model the entire concurrent activity of boiling the water and putting the leaves into the mug by an arbitrary *interleaving* [63] of the actions *boil_water* and *put_leaves*, that is, by the "action-sequences"

$$\mathit{boil\_water}; \mathit{put\_leaves} \qquad \text{and}$$

$$\mathit{put\_leaves}; \mathit{boil\_water}$$

where the semicolon stands for the successive execution of the atomic actions. Finally, if the water is hot and the leaves are within the mug, we can pour the hot water into the mug, that is, we combine the two concurrent atomic actions *put_leaves* and *boil_water* with the action *pour_water* by *sequential composition.*

[9]In the *real-time* approach to the design and analysis of reactive systems, the continuous nature of activities is taken into account (see [158]) for an overview).

The resulting model of the overall "tea preparing"-activity can then be graphically represented as shown in Figure 2.

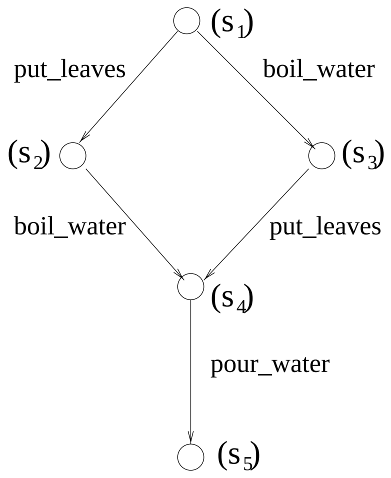

Figure 2: The "Tea Preparation"-Process

If the nodes of the structure in Figure 2 are labelled with according (state-) names, as given in the according brackets, we can employ the notion of *labelled transition systems* (*LTSs* for short) to model the activity of preparing a cup of tea.

Definition 2.1 (Labelled Transition System -LTS-)
A labelled transition system $\mathcal{T} = (s, S, Act, \rightarrow)$ *consists of*

- $s \in S$ *a starting state,*
- S*, a set of* states*,*
- Act*, a set of* atomic actions*, and*
- $\rightarrow \subseteq S \times Act \times S$*, a* transition relation*.*

$(s, \alpha, s') \in \rightarrow$ is called a transition. For (s, α, s') we also write $s \xrightarrow{\alpha} s'$. A state $t' \in S$ is called *reachable from a state* $t \in S$ if and only if there exists a sequence $t = s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_{n-1}} s_n = t'$ where $s_i \in S$ for $1 \leq i \leq n$ and $\alpha_j \in Act$ for $1 \leq j < n$. A labelled transition system $\mathcal{T} = (s, S, Act, \rightarrow)$ is called a *finite state LTS* if and only if the set $\{s' \in S \mid s' \text{ is reachable from } s\}$ of reachable states is finite and the set $\rightarrow$ of transitions is finite. Otherwise, $\mathcal{T}$ is called an *infinite state* labelled transition system. For simplicity, we let LTS also denote the set of labelled transition systems.

Example 2.2
The activity of preparing tea as illustrated in Figure 2 can be modelled by the transitions system $\mathcal{T} = (s_1, S, Act, \rightarrow)$ *where*

- $S = \{s_i | 1 \leq i \leq 5\}$,
- $Act = \{boil\_water, put\_leaves, put\_water\}$, *and*
- $\rightarrow = \{(s_1, boil\_water, s_3), (s_1, put\_leaves, s_2), (s_3, put\_leaves, s_4),$ $(s_2, boil\_water, s_4), (s_4, pour\_water, s_5)\}$. $\square$

LTSs were introduced by R. M. Keller in [114] under the name "named transition systems"[10]. In his article, Keller used them to model and analyze concurrent systems.

Finite state automata (*FSA*) are mathematical structures, omnipresent in computer science (see [170] for a detailed introduction). Historically, *FSA* have been introduced in order to model nervous activity [149, 117]. *FSA* are closely related to finite state *LTSs*, the only difference being the additional distinction of final states for *FSA*.

Various modifications of *LTSs* have been investigated in order to capture different computational aspects of *RCSs*. Examples are "concurrent transition systems" [187], "distributed transition systems" [35], "modal transition systems" [129], "timed transitions systems" [100] and "probabilistic transition systems" [130].

For some purposes, *LTSs* are too discriminating a semantics for *RCSs*. For example, it seems to be unreasonable to distinguish the transition systems $\mathcal{T}_1$ from $\mathcal{T}_2$ (as shown in Figure 3) since they exhibit the same behaviour. In order to avoid

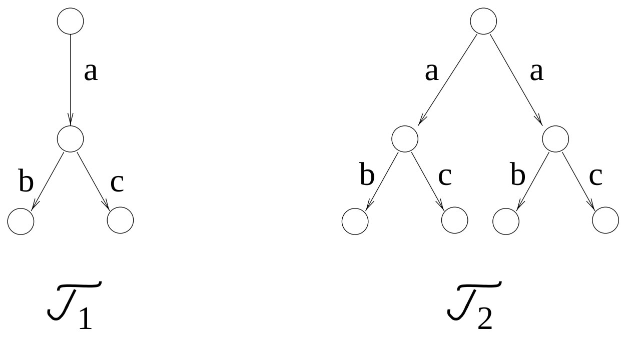

Figure 3: Two Equivalent Processes

[10]In this definition, no explicit starting state was employed.

such unreasonable distinctions, various *behavioural equivalences*[11] can be defined on *LTSs*, each of which gives rise to a particular semantic domain (the elements of such semantic domains are equivalence classes of *LTSs* rather then single *LTSs*). For this reason, *LTSs* can be considered as a fundamental unifying structure based on which many different aspects of *RCSs* can be modelled (see also [155]). Well known behavioural equivalences are, for example, *trace equivalence* [103], *failure equivalence* [30], *simulation equivalence* [166], *observational equivalence* [97] and *strong bisimulation equivalence* [154]. For a comparative study of various behavioural equivalences see [202, 203, 61].

Strong bisimulation equivalence (*SBE* for short) is the finest behavioural equivalence, that is, it identifies less *LTSs* than the other equivalences mentioned above. We now review the formal definition of *SBE*.

Definition 2.3 (Strong Bisimulation Equivalence)
For a fixed set Act of atomic actions let $\mathcal{T}_i = (s_i, S_i, Act, \rightarrow_i)$, $i = 1, 2$, be two labelled transition systems. For $s_1 \in S_1$ and $s_2 \in S_2$, a binary relation $\mathcal{S} \subseteq S_1 \times S_2$ is a strong bisimulation *for s_1 and s_2 if $(s_1, s_2) \in \mathcal{S}$ implies for all $(r, s) \in \mathcal{S}$ and $a \in Act$*

(i) Whenever $r \xrightarrow{a}_1 r'$ then, for some s', $s \xrightarrow{a}_2 s'$ and $(r', s') \in \mathcal{S}$ and

(ii) Whenever $s \xrightarrow{a}_2 s'$ then, for some r', $r \xrightarrow{a}_1 r'$ and $(r', s') \in \mathcal{S}$.

Two states s_1 and s_2 are strongly bisimular *if there exists a strong bisimulation for s_1 and s_2. Two labelled transition systems $\mathcal{T}_i = (s_i, S_i, Act, \rightarrow_i)$, $i = 1, 2$, are strongly bisimular, written $\mathcal{T}_1 \sim_b \mathcal{T}_2$, if s_1 and s_2 are strongly bisimular. The equivalence relation $\sim_b$ is called* strong bisimulation equivalence. $\square$

SBE occurs frequently in related mathematical branches and can be characterized by means of certain two-player games (see for example [191, 192, 193]), modal logics [153, 154] or axiomatic systems [154]. Further it has been abstractly defined in category-theoretic settings in order to supply a unified view on bisimulation for different semantic domains (see for example [109, 179]). *SBE* enjoys important algorithmic properties. The question whether two *LTS* are strongly bisimular is decidable even for certain infinite state *LTSs* (called context-free processes, see

[11]Also called "semantic equivalences", for example, in [202, 203].

for example [38]). In contrary, this does not hold for failure-, trace-, and simulation equivalence [93]. It is well known that deciding language equivalence of FSA is PSPACE-complete [151]. In fact, deciding whether two finite state $LTSs$ are equivalent is PSPACE-hard for any behavioural equivalence that lies between strong bisimulation equivalence and trace equivalence [177]. In contrary, a polynomial time algorithm that decides strong bisimulation equivalence of two finite state $LTSs$ was given in [110] (see also [111]) and shortly later an improved algorithm was proposed in [165]. Checking that two $LTSs$ are bisimular can also be done compositionally (see for example [92]) and (with the aid of ordered binary decision diagrams [31]) symbolically (see for example [22]).

2.1.2 Discussion

When using $LTSs$ to model $RCSs$ one is concerned with control flow aspects rather then issues about concrete data values. Clearly, this can be seen as a drawback when being concerned with the implementation of $RCSs$. On the other side, "data-abstraction" is a valuable feature when being concerned with the design of $RCSs$ (and the verification of these designs) as it puts away "unnecessary" design complexity: It has been observed that the correctness of $RCSs$ is particularly sensitive to control issues whereas data processing is most often less likely to cause fatal errors [65, 95]. A good example might be the destruction of ARIANE 5 [78]. The explosion was actually caused by an excessive floating point number. However, this overflow was caused by a part of the on board software that was unnecessarily running after take off. Yet another design error was to leave the inertial computer system unprotected from being made inoperative by an excessive value of the according variable. Finally, the specification of the inertial reference system did not specifically include the ARIANE 5 trajectory data but those of ARIANE 4: The excessive floating point number was generated due to the higher initial acceleration and a trajectory which leads to a build-up of horizontal velocity of ARIANE 5 which is five times more rapid than for ARIANE 4. Hence, the excessive floating point number was only the result of various design errors.

It has been argued that a system can most often be divided into a (safety-critical) control part and a (non-critical) functional part [145, 212, 216]. $LTSs$ are an appropriate concept to formalize such control parts. Moreover, finite state $LTSs$ are sufficient to model large classes of *RCSs* (see [68, page 1060] and [46]) such as control systems [41], communication protocols [60, 180] or digital circuits [46, 45] and automatic verification techniques can be applied to finite state *LTSs* [46, 45].

In this thesis, *LTSs* (together with strong bisimulation equivalence) are used to formalize designs of *RCSs*. According to the classification in Section 2.1 on page 17 this means that *RCSs* are modelled by a semantic domain of type "state", "interleaving" and "branching time". Apart from the discussion above, this type of semantics has been chosen for the following reasons:

- The close relation to the well known concept of finite state automata makes *LTSs* easily accessible by any computer scientist.

- Being a semantic domain of type "state", infinite behaviour can in many cases be represented by finite state *LTSs*. In general, infinite behaviour cannot be represented by finite objects of a "behaviour"-semantics like, for example, event structures. Finite state *LTSs* can be graphically represented, for example, by process graphs [202] which supports human comprehension.

- Being a semantics of type "interleaving", *LTSs* abstract away from architectural details of the *RCSs* considered. The correctness of designs will therefore be independent of the computing power or performance of the according *RCSs* [103, page 24]. *LTSs* are well suited as a design formalism for *RCSs* as they capture computational phenomena like, for example, *"deadlock"* [5, 185]. On the other hand, a "true concurrency"-semantics might be more appropriate for describing *RCSs* at an implementation-near level of abstraction (see [53] and [198, page 21]).

- Being a "branching time"-semantics, *LTSs* capture the important design concept of nondeterminism. Nondeterminism discharges a system designer from giving too many implementation details already in the design-phase and thus allows to maintain the high abstraction level, necessary for the application of verification techniques (see also [103, pp. 101]). Further, the results and techniques of this thesis can applied in conjunction with results from *automated program synthesis* (see [68, pp. 1058] for an introduction to automated program synthesis). It has been argued that temporal logics should be interpreted over "branching time" semantic models of *RCSs* when used for automated program synthesis [75, page 149].

2.2 Process Algebras: Programming with Reactive System Designs

Though the study of semantic domains for *RCSs* is useful in its own, algebraic notations, usually called *process algebras*[12] (*PAs* for short), supply additional benefits. First, *PAs* allow objects of a semantic domain to be written as concise strings called *process expressions* or *process terms*. *PAs* are thus a means to represent huge or even infinite semantic objects in a short and finite manner thus making them manageable algorithmically. Within *PAs*, reasoning about designs of *RCSs* often boils down to syntactic manipulations of process terms. For example, it is possible to axiomatize strong bisimulation equivalence by a set of algebraic laws whence the equivalence of (finite state) *RCSs*-designs can be shown by applying these laws (see for example [154]).

PAs can be viewed as programming languages for designs of *RCSs*. The syntax of *PAs* is usually generated by simple grammars which contain a number of primitives (basic process terms) and a number of structural operators. The operators can then be used to compose more complex process terms out of simpler ones according to the rules of the grammar. The primitives can then be connected to the basic objects of the semantic domain used and the structural operators induce operations on objects of the semantic domain.

The best known *PAs* are

- The *Calculus of Communicating Systems* (*CCS*) [154],
- the *Communicating Sequential Processes* (*CSP*) [102, 103],
- the *Theory of Communicating Sequential Processes* (*TCSP*) [30] and
- the *Algebra of Communicating Processes* (*ACP*) [19].

The standard semantic domains of these *PAs* are

[12] Also called "process calculi" [154] or "abstract programming languages" [198].

- $LTSs$ with bisimulation equivalence [154] for CCS,
- the "failure trace semantics" [30] for CSP and $TCSP$ and
- "process graphs" with bisimulation equivalence [19] for ACP.

The PAs mentioned above have primarily been used to model software systems (see [147, 156] where CSP was also used to model hardware systems). However, different PAs have be devised in order to model digital circuits, the best known of which might be CIRCAL [152]. Different semantic domains for the above mentioned PAs have been investigated. For example, CCS has been interpreted over Petri-nets (for example in [89, 87, 23]) and over event structures (for example in [23]). $TCSP$ has been interpreted over event structures (for example in [135, 12]) and over $LTSs$ (for example in [162]). Connections between different semantic domains for particular PAs have been studied, for example, in [12] for $TCSP$ and in [23] for CCS. Various PAs (with their standard semantics) have been related to each other, for example, in [29] (CCS with CSP) and [162] ($TCSP$ with CCS). Despite the conciseness of PAs, they often exhibit a high expressive power, for example, CCS and CSP have Turing power (see [154] and [198] respectively).

2.3 Hierarchical Development of Reactive System Designs

As was discussed in the previous chapter, PAs support the formalization of $RCSs$-designs in a modular fashion: Models of $RCSs$ can be denoted by process expressions whose subexpressions denote subcomponents of the according models. The analysis of a design can thus be done component-wise which allows to focus on particular details of the design at a time. Standard PAs thus support some kind of *horizontal modularity*. However, fixing the set of atomic actions in order to abstractly denote "real world activities" means to determine a particular level of abstraction which is then fixed once and for all. This inflexibility to change the level of abstraction is unsatisfactory from a system designers viewpoint. *Vertical modularity* is an important means to overcome this problem and to control the complexity of developing $RCSs$-designs: A complex design can first be given as concise and simple as possible and then be turned into the actually desired design by means of successive *refinement steps*. This method is referred to as the *hierarchical design* of $RCSs$.

Different approaches that aim to allow vertical modularity exist some of which will be discussed below. In doing so, we closely follow the comparative surveys [204, 90] where various approaches to vertical modularity are discussed and many references can be found.

- *Refinement Operator* versus *Hierarchy of Designs.* A refinement operator usually takes as its arguments a "target-design", a "target-primitive" and a design which is supposed to describe the target-primitive in a more precise way and which we call the "detail-design". The application of the refinement operator then renders the target-design more precise by "replacing" the occurrences of the target-primitive in the target-design by the detail-design. By analogy, the application of a refinement operator amounts to implementing the body of a procedure (the detail-design) of a subroutine in a program (the target-design) for which hitherto only the "head" (the target-primitive) had existed.

  In contrary, hierarchies of designs are based on *implementation relations* representatives of which are the behavioural equivalences, discussed in Section 2.1.1. Implementation relations are used to express that a low level design behaves "essentially in the same way" as a high level design. We refer the reader to [2] where many articles on implementation relations can be found. Ascending a hierarchy of designs is in some sense "semantic preserving". For this reason, hierarchies of designs will henceforth referred to as *semantic preserving refinement.*

- *Semantic Action Refinement* versus *Syntactic Action Refinement.* Refinement operators are most often used in settings where the target-primitives are atomic actions. One axis to distinguish refinement operator approaches is to distinguish the kind of domain on which refinement operators are defined. In semantic action refinement, the arguments of a refinement operator are objects of a semantic domain whereas process algebraic expressions constitute the arguments in syntactic action refinement.

Approaches to action refinement can be further distinguished: Whereas action refinement in *atomic action refinement* preserves atomicity (detail-designs are considered to appear non-interruptible and atomic in the target-design), atomicity is relative to the current level of abstraction in *non-atomic action refinement* (that is, actions of the target-design can interfere with the detail-design). Action refinement approaches are further distinguished according to the type of semantics that is explicitly (in semantic action refinement) or implicitly (in syntactic action refinement) involved, for example, whether a "true concurrency" or an "interleaving semantics" is used.

2.3.1 $R\Sigma$: A Process Algebra With Syntactic Action Refinement

In this section we fix the process algebraic framework that is used to develop reactive systems. Let $Act := \{a, b, \ldots\}$ be a fixed countable set of *(atomic) actions* and $Var_{Act} := \{v_1, v_2, \ldots\}$ a fixed countable set of distinguished *action variables* which will be used as "place-holders" for process terms in what follows (see Definition 4.5). We require $Act \cap Var_{Act} = \emptyset$.

Remark 2.4
In what follows, we let $\alpha, \beta, \gamma, \ldots$ range over the set $\mathcal{A} := Act \cup Var_{Act}$, the elements of which are called (atomic) performances. □

We let $Idf := \{x, y, \ldots\}$ be a fixed set of *identifiers*. As usual the process expression 0 is used to denote a process which is unable to perform any atomic performance. Two languages are used to build up process expressions of the form $P[\alpha \rightsquigarrow Q]$. The language $R\Delta$ supplies the terms Q whereas the language $R\Sigma$ provides the terms P.

Definition 2.5 (The Process Algebras $R\Delta$, Δ, $R\Sigma$ and Σ)
Let $R\Delta$ be the language of process terms generated by the grammar

$$Q ::= \ \alpha \mid (Q + Q) \mid (Q; Q) \mid Q[\alpha \rightsquigarrow Q].$$

Let $R\Sigma$ be the language of process expressions generated by the grammar

$$P ::= 0 \mid \alpha \mid x \mid (P + P) \mid (P; P) \mid (P \|_A P) \mid fix(x = P) \mid P[\alpha \rightsquigarrow Q]$$

where $\cdot[\alpha \rightsquigarrow Q]$ is the syntactic action refinement operator, $Q \in R\Delta$ and $A \subseteq \mathcal{A}$. Let Σ, Δ be the languages of process expressions generated by the grammars for $R\Sigma$, $R\Delta$ respectively, without the rule $P ::= P[\alpha \rightsquigarrow Q]$. These two languages will subsequently be used to define logical substitution (see Definition 4.5). □

Intuitively, the operators of the language $R\Sigma$ can be conceived as follows:

- 0: Denotes a terminated process that cannot execute any performance.
- α: Stands for the system that can execute the performance α thereby evolving to the terminated process.
- x: Is used to evaluate process terms of the form $fix(x = P)$ (see below). In isolation, x behaves like the terminated process.
- $(P_1 + P_2)$: Denotes the system that can nondeterministically execute the subsystem P_1 or the subsystem P_2.
- $(P_1; P_2)$: Stands for the system that can execute the subsystem P_1 and, upon successful termination of P_1, proceeds to the execution of the subsystem P_2.
- $(P_1 \|_A P_2)$: Denotes the system that can execute the subsystems P_1 and P_2 concurrently (by interleaving the performances of P_1 and P_2). Performances that occur in the synchronization set A have to be executed synchronously.
- $fix(x = P)$: Denotes the system that executes the subsystem P recursively.
- $P[\alpha \rightsquigarrow Q]$: Stands for the system that replaces the execution of a performance α by the execution of the subsystem Q every time the subsystem P performs α. This operator allows to hierarchically design reactive systems.

In the presence of the sequential composition operator ";" it is common to use a special predicate $\surd$ (see, for example, [4]) to evaluate the semantics of the sequential composition operator ";". Let $\surd \subset \Sigma$ be the least set which contains the term 0 and is closed under the rules $(E \in \surd$ and $F \in \surd) \Rightarrow (E\ op\ F) \in \surd$ where $op \in \{\|_A, +, ;\}$ and $(E \in \surd) \Rightarrow fix(x = E) \in \surd$. An occurrence of an identifier $x \in Idf$ is called *free* in a process expression $P \in R\Sigma$ iff it does not occur within a subterm of the form $fix(x = Q)$. An occurrence of x is called *bound* iff it is not free. In what follows we only consider $R\Sigma$-expressions P in which all identifiers which occur free in P are distinct from all identifiers which occur bound in P. This can easily be achieved by consistent renaming of bound identifiers. An identifier x is *guarded* in a term $P \in R\Sigma$ iff each free occurrence of x only occurs in subexpressions F where F lies in a subexpression $(E; F)$ such that $E \notin \surd$. A term $P \in R\Sigma$ is called *guarded* iff in each subexpression $fix(x = Q)$ of P the identifier x is guarded in Q. For a language $L \subseteq R\Sigma$ we define the set of guarded $R\Sigma$-expressions by $GL := \{P \in L \mid P \text{ is guarded}\}$.

As in [88] we define a function which gives the set of performances of a process expression.

Definition 2.6 (Performances of process expressions)
Let $P, P_1, P_2 \in R\Sigma$ and $Q \in R\Delta$ be process expressions. The function $\xi : R\Sigma \to 2^{\mathcal{A}}$ is defined by

$$\xi(*) := \emptyset \text{ where } * \in \{0\} \cup Idf \;, \qquad \xi(\alpha) := \{\alpha\} \;,$$

$$\xi((P_1 \; op \; P_2)) := \xi(P_1) \cup \xi(P_2) \text{ where } op \in \{+, ;, \|_A\} \;, \qquad \xi(fix(x = P)) := \xi(P) \;,$$

$$\xi(P[\alpha \rightsquigarrow Q]) := \begin{cases} \xi(P) \setminus \{\alpha\} \cup \xi(Q) & \text{if } \alpha \in \xi(P) \\ \xi(P) & \text{else} \end{cases}$$

□

The set of synchronization performances of a process expression $P \in R\Sigma$ is given by the following function.

Definition 2.7 (Synchronization performances of process expressions)
Let $P, P_1, P_2 \in R\Sigma$ and $Q \in R\Delta$ be process expressions. The function $\chi : R\Sigma \to 2^{\mathcal{A}}$ is defined by

$$\chi(*) := \emptyset \text{ where } * \in \{0\} \cup Idf \cup \mathcal{A} \;, \qquad \chi(fix(x = P)) := \chi(P) \;,$$

$$\chi((P_1 \; op \; P_2)) := \chi(P_1) \cup \chi(P_2) \text{ where } op \in \{+, ;\}$$

$$\chi((P_1 \|_A P_2)) := \chi(P_1) \cup \chi(P_2) \cup A \;,$$

$$\chi(P[\alpha \rightsquigarrow Q]) := \begin{cases} \chi(P) \setminus \{\alpha\} \cup \xi(Q) & \text{if } \alpha \in \chi(P) \\ \chi(P) & \text{else} \end{cases}$$

□

Let the *alphabet* of a process expression $P \in R\Sigma$ be defined by $alph(P) := \xi(P) \cup \chi(P)$. For $Q \in R\Delta$ we have $alph(Q) = \xi(Q)$. Below, we define some important properties of process expressions which will be employed later when the main result is proven.

Definition 2.8 (Alphabet-disjointness)

- *A process expression* $P_1 \in R\Sigma$ *is called* ξ-disjoint *from a process expression* $P_2 \in R\Sigma$ *iff* $\xi(P_1) \cap \xi(P_2) = \emptyset$.

- *A process expression* $P_1 \in R\Sigma$ *is called* $\chi\xi$-disjoint *from a process expression* $P_2 \in R\Sigma$ *iff* $\chi(P_1) \cap \xi(P_2) = \emptyset$.

- *A process expression* $P \in R\Sigma$ *is called* alphabet-disjoint *from a process expression* $Q \in R\Delta$ *iff* P *is* ξ*-disjoint and* $\chi\xi$*-disjoint from* Q*, that is,* $alph(P) \cap alph(Q) = \emptyset$. □

Definition 2.9 (Unique synchronization, distinctness)

- *A process expression* $P \in R\Sigma$ *is called* uniquely synchronized *iff for all terms* $(P_1 \|_A P_2)$ *which occur in* P*,* $A = \chi(P_i)$ *holds for* $i = 1, 2$*. For a language* $L \subseteq R\Sigma$ *we define the uniquely synchronized fragment of* L *by*

$$UL := \{P \in L \mid P \textit{ is uniquely synchronized}\}.$$

- *A process expression* $Q \in R\Delta$ *is called* distinct *iff for all subexpressions of the form* $(Q_1; Q_2)$*,* $(Q_1 + Q_2)$ *and* $Q_1[\alpha \rightsquigarrow Q_2]$ *that occur in* Q *we have that* $\xi(Q_1) \cap \xi(Q_2) = \emptyset$. □

Lemma 2.10

Let $P = (P_1 \|_A P_2) \in U\Sigma$ *be a process expression. Then we have that* $A = \chi(P)$.

Proof:

- $A \subseteq \chi(P)$. Follows immediately from the definition of the function $\chi : \Sigma \to 2^{\mathcal{A}}$.

- $\chi(P) \subseteq A$. Assume $A \subset \chi(P)$, i.e.

$$\exists \alpha \in \mathcal{A}(\alpha \in \chi(P) \text{ and } \alpha \notin A)$$

Then $\alpha \in \chi(P_1)$ or $\alpha \in \chi(P_2)$ since $\alpha \in \chi(P) = A \cup \chi(P_1) \cup \chi(P_2)$ by definition and $\alpha \notin A$ by the assumption. W.l.o.g. assume $\alpha \in \chi(P_1)$. Since P is uniquely synchronized we have $A = \chi(P_1)$ by Definition 2.9 whence we must have $\alpha \in A$. Contradiction. ■

We now proceed to the definition of performance refinement. As in [4, 88] we use a *reduction function red* : $R\Sigma \to \Sigma$ which removes the occurrence of all refinement operators in a process expression (see Definition 2.15). The reduction function is based on an operation of syntactic substitution. We adapt the definition of [88] for our purposes.

Definition 2.11 (Syntactic Substitution for Σ)
Let $P, P_1, P_2 \in \Sigma$ and $Q \in \Delta$ be process expressions. Syntactic substitution*, denoted $(P)\{Q/\alpha\}$ is defined as follows:*

$$(*)\{Q/\alpha\} := * \ \textit{where} \ * \in \{0\} \cup Idf \ , \qquad (\alpha)\{Q/\beta\} := \begin{cases} Q & \textit{if } \alpha = \beta \\ \alpha & \textit{otherwise} \end{cases} ,$$

$$((P_1 \ op \ P_2))\{Q/\alpha\} := ((P_1)\{Q/\alpha\} \ op \ (P_2)\{Q/\alpha\}) \ \textit{where} \ op \in \{+, ;\} \ ,$$

$$((P_1 \parallel_A P_2))\{Q/\alpha\} := \begin{cases} ((P_1)\{Q/\alpha\} \parallel_{A\setminus\{\alpha\}\cup\xi(Q)} (P_2)\{Q/\alpha\}) & \textit{if } \alpha \in A \\ ((P_1)\{Q/\alpha\} \parallel_A (P_2)\{Q/\alpha\}) & \textit{if } \alpha \notin A \end{cases} ,$$

$$(fix(x = P))\{Q/\alpha\} := fix(x = (P)\{Q/\alpha\})$$ □

Remark 2.12
To avoid excessive use of brackets we sometimes use the notation $P\{Q/\alpha\}$ instead of $(P)\{Q/\alpha\}$ if the context avoids ambiguity. □

The following remark shows that several nested applications of the substitution operation can be reduced to only one such application.

Remark 2.13
Let $P \in \Sigma$ and $Q_1, Q_2 \in \Delta$ be process expressions and $\gamma_1, \gamma_2 \in \mathcal{A}$. Further let $* \in \{;, +\}$. If $\gamma_1, \gamma_2 \notin \xi((Q_1 * Q_2)) \cup alph(P)$ and $\gamma_1 \neq \gamma_2$ then

$$(((P)\{(\gamma_1 * \gamma_2)/\alpha\})\{Q_2/\gamma_2\})\{Q_1/\gamma_1\} = (P)\{(Q_1 * Q_2)/\alpha\}.$$ □

The proof of the above remark is by induction on the structure of $P \in \Sigma$. The following lemma shows that the set of performances and the set of synchronization performances of a term $(P)\{Q/\alpha\} \in \Sigma$ can be directly calculated from the terms P, Q and α.

Lemma 2.14

Let $P \in \Sigma$, $Q \in \Delta$ be process expressions and $\alpha \in \mathcal{A}$ be a performance. Then we have

$$1)\ \xi((P)\{Q/\alpha\}) = \begin{cases} \xi(P) \setminus \{\alpha\} \cup \xi(Q) & \text{if } \alpha \in \xi(P) \\ \xi(P) & \text{else} \end{cases}$$

$$2)\ \chi((P)\{Q/\alpha\}) = \begin{cases} \chi(P) \setminus \{\alpha\} \cup \xi(Q) & \text{if } \alpha \in \chi(P) \\ \chi(P) & \text{else} \end{cases}$$

Proof: The proof is by induction on the structure $P \in \Sigma$. ■

Definition 2.15 (Reduction function for $R\Sigma$)

Let $P, P_1, P_2 \in R\Sigma$ and $Q \in R\Delta$ be process expressions. The reduction function (for process expressions) *$red : R\Sigma \to \Sigma$ is defined as follows:*

$$red(*) := * \text{ for } * \in \{0\} \cup Idf \cup \mathcal{A}\ , \quad red(fix(x = P)) := fix(x = red(P))\ ,$$

$$red((P_1\ op\ P_2)) := (red(P_1)\ op\ red(P_2)) \text{ where } op \in \{+, ;, \|_A\}\ ,$$

$$red(P[\alpha \rightsquigarrow Q]) := (red(P))\{red(Q)/\alpha\}$$ □

We illustrate the reduction function by the following example.

Example 2.16

Consider the process expression $P = ((\alpha;\beta)\|_{\{\alpha\}}\alpha)[\alpha \rightsquigarrow (\alpha_1 + \alpha_2)]$. Then we have that $red(P) = (((\alpha_1 + \alpha_2);\beta)\|_{\{\alpha_1,\alpha_2\}}(\alpha_1 + \alpha_2))$. □

Remark 2.17 states that one application of the reduction function is sufficient to remove all refinement operators occurring in a process expression.

Remark 2.17

Let $P \in R\Sigma$ and $Q \in R\Delta$. Then $red(P[\alpha \rightsquigarrow Q]) = red(red(P)[\alpha \rightsquigarrow Q])$. □

The following lemma states that the set of performances and the set of synchronization performances of process expressions are invariant under the application of the reduction function.

Lemma 2.18

Let $P \in R\Sigma$ be a process expression. Then we have

1) $\xi(P) = \xi(red(P))$

2) $\chi(P) = \chi(red(P))$

Proof: The proof is by induction on the structure of $P \in R\Sigma$ using Lemma 2.14. ■

Lemma 2.19
Let $P \in \Sigma$, $Q_1, Q_2 \in R\Delta$ be process expressions and $\gamma_1, \gamma_2 \in \mathcal{A}$ such that $\gamma_1 \neq \gamma_2$. Further let $* \in \{\ ;\ ,\ +\ \}$. If $\gamma_1, \gamma_2 \notin alph(P) \cup \xi((Q_1 * Q_2))$ then we have that $red(((P[\alpha \rightsquigarrow (\gamma_1 * \gamma_2)])[\gamma_2 \rightsquigarrow Q_2])[\gamma_1 \rightsquigarrow Q_1]) = red(P[\alpha \rightsquigarrow (Q_1 * Q_2)])$.

Proof: Follows from Remark 2.13 and Remark 2.17. ■

The next lemma shows how the reduction function distributes over the parallel composition operator.

Lemma 2.20
Let $P_1, P_2 \in R\Sigma$ and $Q \in R\Delta$ be process expressions.

1) *If $\alpha \notin A$ then* $red((P_1 \parallel_A P_2)[\alpha \rightsquigarrow Q]) =$
$$(red(P_1[\alpha \rightsquigarrow Q]) \parallel_A red(P_2[\alpha \rightsquigarrow Q]))$$

2) *If $\alpha \in A$ then* $red((P_1 \parallel_A P_2)[\alpha \rightsquigarrow Q]) =$
$$(red(P_1[\alpha \rightsquigarrow Q]) \parallel_{A\setminus\{\alpha\}\cup\xi(Q)} red(P_2[\alpha \rightsquigarrow Q]))$$

Proof: Follows from Definition 2.11, Definition 2.15 and Lemma 2.18. ■

The operational semantics of the language Σ is defined as follows (see also [162]).

Definition 2.21 (Operational semantics for Σ)
Let $P, Q \in \Sigma$ be process expressions.

$$\frac{}{\alpha \xrightarrow{\alpha} 0} \qquad \frac{P \xrightarrow{\alpha} P'}{(P+Q) \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{(P+Q) \xrightarrow{\alpha} Q'}$$

$$\frac{Q \xrightarrow{\alpha} Q'}{(P;Q) \xrightarrow{\alpha} Q'} \ \textit{if } P \in \surd \qquad \frac{P \xrightarrow{\alpha} P'}{(P;Q) \xrightarrow{\alpha} (P';Q)}$$

$$\frac{P \xrightarrow{\alpha} P'}{(P \parallel_A Q) \xrightarrow{\alpha} (P' \parallel_A Q)} \ \textit{if } \alpha \notin A \qquad \frac{Q \xrightarrow{\alpha} Q'}{(P \parallel_A Q) \xrightarrow{\alpha} (P \parallel_A Q')} \ \textit{if } \alpha \notin A$$

$$\frac{P[fix(x{=}P)/x] \xrightarrow{\alpha} Q}{fix(x{=}P) \xrightarrow{\alpha} Q} \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\alpha} Q'}{(P \parallel_A Q) \xrightarrow{\alpha} (P' \parallel_A Q')} \ \textit{if } \alpha \in A$$

$\square$

A process expression P determines a *labelled transition system with termination*, that is, a tuple $\mathcal{T}(P) = (P, \Sigma, \mathcal{A}, \rightarrow, \surd)$ where $P \in \Sigma$ is the initial state and $\rightarrow \subseteq \Sigma \times \mathcal{A} \times \Sigma$ is the set of transitions, derived from the operational semantics. $\surd$ is the set of terminated states as defined before.

To supply semantics for terms $P \in R\Sigma$ we define $\mathcal{T}(P) := \mathcal{T}(red(P))$. This expresses the philosophy that the behaviour of the process P is considered to be identical to that of the process $red(P)$ (see also [4]). Intuitively this is justified by the observation that information about α is distributed over several levels of abstraction in the term $P[\alpha \rightsquigarrow Q]$, that is, it can be considered as a 'coded' version of the term $red(P[\alpha \rightsquigarrow Q])$ where the different abstraction levels have been collapsed. In what follows we sometimes identify the term P with the transition system $\mathcal{T}(P)$ if the context avoids ambiguity.

Remark 2.22
The absence of the parallel composition operator in terms $Q \in R\Delta$ is no severe restriction. For any finite state system it is possible to replace $\|_A$ by appropriate combinations of sequential composition and binary choice operators without changing the semantics (up to strong bisimulation equivalence [154]). The exclusion of the empty process term 0 from the language $R\Delta$ means that we disallow 'forgetful refinement'[13]. As the refinement of a (terminating) action by some infinite behaviour violates the intuition [90], no expression of the form $fix(x = P)$ is allowed to occur in a term $Q \in R\Delta$. $\square$

Definition 2.23 (See [12])
Let $P, A_1, \ldots, A_n \in \Sigma$ and $x_1, \ldots, x_n \in Idf$ be pairwise distinct identifiers. The Σ-term $P[A_1/x_1, \ldots, A_n/x_n]$ arises from P by substituting each free occurrence of the identifiers $x_1, \ldots, x_n$ in P simultaneously by the terms $A_1, \ldots, A_n$. $\square$

Some elementary properties of the reduction function which allow us to relate the behaviour of P and $red(P[\alpha \rightsquigarrow Q])$ are necessary for the proof of the main theorem. In turn, the proofs of those properties make use of Lemma 2.25 which

[13]Such refinements cannot be explained by a change in the level of abstraction [201] and are usually avoided.

establishes a connection between behavioural properties of guarded processes $P \in G\Sigma$ and $P[A_1/x_1, \ldots, A_n/x_n]$.

In [12], Lemma 2.24 is used to prove Lemma 2.25.

Lemma 2.24 (See [12])
Let $P, B, A_1, \ldots A_n \in G\Sigma$ and let $x_1, \ldots, x_n, y \in Idf$ be pairwise distinct identifiers such that y does not occur free in $A_1, \ldots, A_n$. Then

$$P[A_1/x_1, \ldots, A_n/x_n, B[\tilde{A}/\tilde{x}]/y] = P[B/y][\tilde{A}/\tilde{x}]$$

Proof: By induction on the structure of $P \in G\Sigma$ (see also [12]). ■

Lemma 2.25
[See [12]] Let $P, A_1, \ldots, A_n \in G\Sigma$ and let $x_1, \ldots, x_n \in Idf$ be pairwise distinct identifiers which are guarded in P. Then we have:

If $P[A_1/x_1, \ldots, A_n/x_n] \xrightarrow{\alpha} Q$, then there exists $P' \in G\Sigma$ with
1. *$P \xrightarrow{\alpha} P'$ and*
2. *$P'[A_1/x_1, \ldots, A_n/x_n] = Q$*

Proof: In [12] the action prefixing (see, for example [154]) is used instead of an operator for sequential composition. Hence, we only show the induction steps $P = \gamma \in \mathcal{A}$ and $P = (P_1; P_2)$. Note that the semantics of our choice operator '+' resembles to the 'external choice' semantics of the operator '□' in [12].

$P \in Idf \cup \{0\}$: Trivial.
$P = \gamma \in \mathcal{A}$: Then $\gamma[\tilde{A}/\tilde{x}] = \gamma$. We choose $P' = 0$ to complete this step.

$P = (P_1; P_2)$:

Assume $(P_1; P_2)[\tilde{A}/\tilde{x}] \xrightarrow{\alpha} Q$. Then $(P_1[\tilde{A}/\tilde{x}]; P_2[\tilde{A}/\tilde{x}]) \xrightarrow{\alpha} Q$.

Case 1: $P_1[\tilde{A}/\tilde{x}] \in \surd$. Then $P_1 \in \surd$. To see this assume $P_1[\tilde{A}/\tilde{x}] \in \surd$ and $P_1 \notin \surd$. From the former we have either $P_1 \in \surd$ which gives an immediate contradiction or x_i must occur unguarded in P_1 for some $i \in \{1, \ldots, n\}$ and $A_i \in \surd$. But this is a contradiction to the conditions of the lemma since it implies that x_i occurs unguarded in P.

By the assumption and Definition 2.21 we have $P_2[\tilde{A}/\tilde{x}] \xrightarrow{\alpha} Q$ which implies

$$\exists P_2'(P_2 \xrightarrow{\alpha} P_2' \text{ and } P_2'[\tilde{A}/\tilde{x}] = Q)$$

by the induction hypothesis. Hence

$$\exists P_2'((P_1; P_2) \xrightarrow{\alpha} P_2' \text{ and } P_2'[\tilde{A}/\tilde{x}] = Q)$$

by Definition 2.21 since $P_1 \in \surd$.

Case 2: $P_1[\tilde{A}/\tilde{x}] \notin \surd$. This implies $P_1 \notin \surd$. By the assumption and Definition 2.21 we have

$$Q = (E; P_2[\tilde{A}/\tilde{x}]) \text{ where } P_1[\tilde{A}/\tilde{x}] \xrightarrow{\alpha} E$$

This implies

$$\exists P_1'(P_1 \xrightarrow{\alpha} P_1' \text{ and } P_1'[\tilde{P}/\tilde{x}] = E)$$

by the induction hypothesis. Hence

$$\exists P_1'((P_1; P_2) \xrightarrow{\alpha} (P_1'; P_2) \text{ and } P_1'[\tilde{A}/\tilde{x}] = E)$$

by Definition 2.21. Further

$$\begin{gathered}(P_1'; P_2)[\tilde{A}/\tilde{x}] = \\ (P_1'[\tilde{A}/\tilde{x}]; P_2[\tilde{A}/\tilde{x}]) = \\ (E; P_2[\tilde{A}/\tilde{x}]) = \\ Q\end{gathered}$$

whence the claim of the lemma follows by choosing $P' = (P_1'; P_2)$. ■

Some elementary properties of the function *red* are summarized in the following which allow us to relate the behaviour of P and $red(P[\alpha \rightsquigarrow Q])$. The proofs of Lemma 2.28, Lemma 2.30, Lemma 2.31 and Lemma 2.32 stated below make use of Lemma 2.25. Lemma 2.26 states that refinements behave well in the sense that they neither remove a process expression from the set $\surd$ of terminated processes nor introduce a reduced process expression to it while Lemma 2.27 states that we can first substitute the term E for every variable x in P and than refine the resulting expression instead of substituting the refined term E for every x in the refined term P.

Lemma 2.26

Let $P \in \Sigma$ and $Q \in R\Delta$. Then $P \in \surd$ iff $red(P[\alpha \rightsquigarrow Q]) \in \surd$.

Proof: Immediate. ■

Lemma 2.27

Let $P, E \in \Sigma$, $Q \in R\Delta$ and $x \in Idf$. Then

$$red(P[\alpha \rightsquigarrow Q])[red(E[\alpha \rightsquigarrow Q])/x] = red((P[E/x])[\alpha \rightsquigarrow Q])$$

Proof: By induction on the structure of $P \in \Sigma$. ■

Lemma 2.28

Let $P \in G\Sigma$ and $Q \in R\Delta$ be process expressions. Then we have

1) If $\alpha \neq \beta$ and $\beta \notin \xi(Q)$ then, for all $P' \in G\Sigma$, we have
$P \xrightarrow{\beta} P' \Rightarrow red(P[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P'[\alpha \rightsquigarrow Q])$

2) If $\beta \notin \xi(Q)$ then, for all $P' \in G\Sigma$,
$red(P[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} P' \Rightarrow \exists P''(P \xrightarrow{\beta} P''$ and $red(P''[\alpha \rightsquigarrow Q]) = P')$

3) If $P \in UG\Sigma$ and $\alpha \neq \beta$ and $(\alpha \in \chi(P) \wedge \beta \notin \chi(P)) \Rightarrow (\beta \notin \xi(Q))$ then, for all $P' \in UG\Sigma$, $P \xrightarrow{\beta} P' \Rightarrow red(P[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P'[\alpha \rightsquigarrow Q])$

Proof: The proofs of assertion 1) and assertion 3) only differ in one case for the induction step $P = (P_1 \|_A P_2)$. Hence we prove assertion 1) and assertion 3) simultaneously making a distinction only in the above mentioned case. The proof is by structural induction on $P \in G\Sigma$ ($P \in UG\Sigma$ resp.).

$\underline{P = 0,\ P = x \in Idf \text{ and } P = \gamma \text{ where } \gamma \neq \beta}$: Trivial.

$\underline{P = \beta}$:

Obvious as $P \xrightarrow{\beta} 0$, $red(P[\alpha \rightsquigarrow Q]) = \beta = P$ since $\alpha \neq \beta$ and $red(0[\alpha \rightsquigarrow Q]) = 0$.

$\underline{P = (P_1 + P_2)}$:

Assume $(P_1 + P_2) \xrightarrow{\beta} P'$ for some $P' \in G\Sigma$. By Definition 2.21 we get $P_1 \xrightarrow{\beta} P'$ or $P_2 \xrightarrow{\beta} P'$. W.l.o.g. assume the former. Then

$$red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P'[\alpha \rightsquigarrow Q])$$

by the induction hypothesis. This implies

$$(red(P_1[\alpha \rightsquigarrow Q]) + red(P_2[\alpha \rightsquigarrow Q])) \xrightarrow{\beta} red(P'[\alpha \rightsquigarrow Q])$$

As $(red(P_1[\alpha \rightsquigarrow Q]) + red(P_2[\alpha \rightsquigarrow Q])) = red((P_1 + P_2)[\alpha \rightsquigarrow Q])$ we get

$$red(P[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P'[\alpha \rightsquigarrow Q]).$$

$\underline{P = (P_1; P_2):}$

Assume $(P_1; P_2) \xrightarrow{\beta} P'$ for some $P' \in G\Sigma$. By Definition 2.21 we have to consider two cases:

Case 1: $P_1 \in \surd$. By the assumption we must have $P_2 \xrightarrow{\beta} P'$. We get

$$red(P_2[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P'[\alpha \rightsquigarrow Q])$$

by the induction hypothesis. By Lemma 2.26 we get $red(P_1[\alpha \rightsquigarrow Q]) \in \surd$. This implies

$$(red(P_1[\alpha \rightsquigarrow Q]); red(P_2[\alpha \rightsquigarrow Q])) \xrightarrow{\beta} red(P'[\alpha \rightsquigarrow Q])$$

hence

$$red(P[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P'[\alpha \rightsquigarrow Q]).$$

Case 2: $P_1 \notin \surd$. Then $P_1 \xrightarrow{\beta} P_1'$ and $(P_1; P_2) \xrightarrow{\beta} (P_1'; P_2)$ by Definition 2.21. We get

$$red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P_1'[\alpha \rightsquigarrow Q])$$

by the induction hypothesis. By Definition 2.21 we obtain

$$(red(P_1[\alpha \rightsquigarrow Q]); \tilde{P}) \xrightarrow{\beta} (red(P_1'[\alpha \rightsquigarrow Q]); \tilde{P})$$

for any $\tilde{P} \in \Sigma$. Let $\tilde{P} = red(P_2[\alpha \rightsquigarrow Q])$ then

$$red(P[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red((P_1'; P_2)[\alpha \rightsquigarrow Q]) = red(P'[\alpha \rightsquigarrow Q]).$$

<u>$P = (P_1 \parallel_A P_2)$:</u>

Assume $(P_1 \parallel_A P_2) \xrightarrow{\beta} P'$ for some $P' \in G\Sigma$. We distinguish four different cases:

Case 1: $\alpha, \beta \notin A$. Then we have

$$P' = (P_1' \parallel_A P_2) \text{ where } P_1 \xrightarrow{\beta} P_1'$$

or

$$P' = (P_1 \parallel_A P_2') \text{ where } P_2 \xrightarrow{\beta} P_2'$$

by Definition 2.21 and $\beta \notin A$. W.l.o.g. assume the former. Then

$$red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P_1'[\alpha \rightsquigarrow Q])$$

by the induction hypothesis. Hence

$$(red(P_1[\alpha \rightsquigarrow Q]) \parallel_A red(P_2[\alpha \rightsquigarrow Q])) \xrightarrow{\beta} (red(P_1'[\alpha \rightsquigarrow Q]) \parallel_A red(P_2[\alpha \rightsquigarrow Q]))$$

by Definition 2.21 and $\beta \notin A$. We conclude

$$red((P_1 \parallel_A P_2)[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red((P_1' \parallel_A P_2)[\alpha \rightsquigarrow Q])$$

by assertion 1) of Lemma 2.20 since $\alpha \notin A$.

Case 2: $\beta \notin A$ and $\alpha \in A$. In this case we have to use the condition $\beta \notin \xi(Q)$. This condition is satisfied by the premises of assertion 1). For assertion 3) we have that $\alpha \in A$ implies $\alpha \in \chi(P)$ by definition. Since $\beta \notin A$ by the current case and P is uniquely synchronized, we have $\beta \notin \chi(P)$ by Lemma 2.10 whence we may assume $\beta \notin \xi(Q)$ as requested by the premises of assertion 3). We can now proceed in the simultaneous proof of assertion 1) and assertion 3): We have

$$P' = (P_1' \parallel_A P_2) \text{ where } P_1 \xrightarrow{\beta} P_1'$$

or

$$P' = (P_1 \parallel_A P_2') \text{ where } P_2 \xrightarrow{\beta} P_2'$$

by Definition 2.21 and since $\beta \notin A$. W.l.o.g. assume the former. Then

$$red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P_1'[\alpha \rightsquigarrow Q])$$

by the induction hypothesis. Since $\beta \notin A$ by the current case and $\beta \notin \xi(Q)$ we have $\beta \notin A \setminus \{\alpha\} \cup \xi(Q)$. Hence

$$(red(P_1[\alpha \rightsquigarrow Q]) \parallel_{A\setminus\{\alpha\}\cup\xi(Q)} red(P_2[\alpha \rightsquigarrow Q])) \xrightarrow{\beta}$$

$$(red(P_1'[\alpha \rightsquigarrow Q]) \parallel_{A\setminus\{\alpha\}\cup\xi(Q)} red(P_2[\alpha \rightsquigarrow Q]))$$

by Definition 2.21. We conclude

$$red((P_1 \parallel_A P_2)[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red((P_1' \parallel_A P_2)[\alpha \rightsquigarrow Q])$$

by assertion 2) of Lemma 2.20 since $\alpha \in A$.

Case 3: $\beta \in A$ and $\alpha \notin A$. Since $\beta \in A$ we have

$$P' = (P_1' \parallel_A P_2') \text{ and } P_1 \xrightarrow{\beta} P_1' \text{ and } P_2 \xrightarrow{\beta} P_2'$$

by Definition 2.21. This implies

$$red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P_1'[\alpha \rightsquigarrow Q])$$

and

$$red(P_2[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P_2'[\alpha \rightsquigarrow Q])$$

by the induction hypothesis. Hence

$$(red(P_1[\alpha \rightsquigarrow Q]) \parallel_A red(P_2[\alpha \rightsquigarrow Q])) \xrightarrow{\beta} (red(P_1'[\alpha \rightsquigarrow Q]) \parallel_A red(P_2'[\alpha \rightsquigarrow Q]))$$

by Definition 2.21 since $\beta \in A$. Since $\alpha \notin A$ we conclude

$$red((P_1 \parallel_A P_2)[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red((P_1' \parallel_A P_2')[\alpha \rightsquigarrow Q])$$

by assertion 1) of Lemma 2.20.

Case 4: $\beta \in A$ and $\alpha \in A$. Then

$$P' = (P_1' \parallel_A P_2') \text{ and } P_1 \xrightarrow{\beta} P_1' \text{ and } P_2 \xrightarrow{\beta} P_2'$$

by Definition 2.21. Hence

$$red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P_1'[\alpha \rightsquigarrow Q])$$

and

$$red(P_2[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P_2'[\alpha \rightsquigarrow Q])$$

by the induction hypothesis. Since $\alpha \neq \beta$ by the conditions of the lemma and $\beta \in A$ by the current case we have $\beta \in A \setminus \{\alpha\} \cup \xi(Q)$. Hence

$$(red(P_1[\alpha \rightsquigarrow Q]) \parallel_{A\setminus\{\alpha\}\cup\xi(Q)} red(P_2[\alpha \rightsquigarrow Q])) \xrightarrow{\beta}$$

$$(red(P_1'[\alpha \rightsquigarrow Q]) \parallel_{A\setminus\{\alpha\}\cup\xi(Q)} red(P_2'[\alpha \rightsquigarrow Q]))$$

by Definition 2.21. Since $\alpha \in A$ we conclude

$$red((P_1 \parallel_A P_2)[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} (red((P_1' \parallel_A P_2')[\alpha \rightsquigarrow Q])$$

by assertion 2) of Lemma 2.20.

$\underline{P = fix(x = P_1)}$.

Assume $fix(x = P_1) \xrightarrow{\beta} P'$. Then $P_1[fix(x = P_1)/x] \xrightarrow{\beta} P'$. Hence

$$\exists \hat{P} \in G\Sigma(P_1 \xrightarrow{\beta} \hat{P} \text{ and } \hat{P}[fix(x = P_1)/x] = P')$$

by Lemma 2.25. This implies

$$\exists \hat{P} \in G\Sigma(red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(\hat{P}[\alpha \rightsquigarrow Q]))$$

by the induction hypothesis whence

$$\exists \hat{P} \in G\Sigma\Big(red(P_1[\alpha \rightsquigarrow Q])[fix(x = red(P_1[\alpha \rightsquigarrow Q]))/x] \xrightarrow{\beta}$$

$$red(\hat{P}[\alpha \rightsquigarrow Q])[fix(x = red(P_1[\alpha \rightsquigarrow Q]))/x]\Big)$$

We obtain

$$\exists \hat{P} \in G\Sigma\Big(fix(x = red(P_1[\alpha \rightsquigarrow Q])) \xrightarrow{\beta}$$

$$red(\hat{P}[\alpha \rightsquigarrow Q])[fix(x = red(P_1[\alpha \rightsquigarrow Q]))/x]\Big)$$

by Definition 2.21. It follows

$$\exists \hat{P} \in G\Sigma\Big(red((fix(x = P_1))[\alpha \rightsquigarrow Q]) \xrightarrow{\beta}$$

$$red(\hat{P}[\alpha \rightsquigarrow Q])[fix(x = red(P_1[\alpha \rightsquigarrow Q]))/x]\Big)$$

by Definition 4.5 and Definition 4.13. Hence

$$\exists \hat{P} \in G\Sigma\Big(red((fix(x = P_1))[\alpha \rightsquigarrow Q]) \xrightarrow{\beta}$$

$$red((\hat{P}[fix(x = P_1)/x])[\alpha \rightsquigarrow Q]))$$

by Lemma 2.27 whence we conclude

$$red((fix(x = P_1))[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P'[\alpha \rightsquigarrow Q]).$$

We proceed to the proof of assertion 2). Again we use structural induction on $P \in G\Sigma$.

$\underline{P = * \in \{0\} \cup Idf}$: Trivial, since $red(P[\alpha \rightsquigarrow Q]) = *$.

$\underline{P = \gamma}$:

Case 1: $\gamma = \alpha$. Then $red(\alpha[\alpha \rightsquigarrow Q]) = red(Q)$. But $\beta \notin \xi(Q)$ by the condition whence the implication is trivially true.
Case 2: $\gamma \neq \alpha$. Assume $\gamma = \beta$. Then $red(\beta[\alpha \rightsquigarrow Q]) = \beta$. Hence $P' = 0$. We choose $P'' = 0$ whence $red(P''[\alpha \rightsquigarrow Q]) = 0 = P'$. Assume $\gamma \neq \beta$. Then $red(P[\alpha \rightsquigarrow Q]) = \gamma$. Since $\gamma \neq \beta$ the implication is trivially true.

$\underline{P = (P_1 + P_2)}$:

Assume $red((P_1 + P_2)[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} P'$ for some $P' \in G\Sigma$. From Definition 2.11 and Definition 2.15 we obtain

$$(red(P_1[\alpha \rightsquigarrow Q]) + red(P_2[\alpha \rightsquigarrow Q])) \xrightarrow{\beta} P'$$

hence

$$red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} P' \text{ or } red(P_2[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} P'.$$

W.l.o.g. assume the former. Then

$$\exists P''(P_1 \xrightarrow{\beta} P'' \text{ and } red(P''[\alpha \rightsquigarrow Q]) = P')$$

and

$$\exists P''((P_1 + P_2) \xrightarrow{\beta} P'' \text{ and } red(P''[\alpha \rightsquigarrow Q]) = P').$$

<u>$P = (P_1; P_2)$:</u>

Assume $red((P_1; P_2)[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} P'$ for some $P' \in G\Sigma$. We obtain

$$(red(P_1[\alpha \rightsquigarrow Q]); red(P_2[\alpha \rightsquigarrow Q])) \xrightarrow{\beta} P'.$$

Case 1: $P' = (E; red(P_2[\alpha \rightsquigarrow Q]))$ where $red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} E$.

Case 2: $P' = F$ where $red(P_2[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} F$ and $red(P_1[\alpha \rightsquigarrow Q]) \in \surd$.

For case 1 we get

$$\exists \tilde{P}(P_1 \xrightarrow{\beta} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow Q]) = E) \quad (*)$$

by the induction hypothesis. This implies

$$\exists \tilde{P}((P_1; P_2) \xrightarrow{\beta} (\tilde{P}; P_2) \text{ and } red(\tilde{P}[\alpha \rightsquigarrow Q]) = E)$$

by Definition 2.21. As

$$red((\tilde{P}; P_2)[\alpha \rightsquigarrow Q]) = (E; red(P_2[\alpha \rightsquigarrow Q])) = P'$$

by assertion $(*)$, we conclude

$$\exists P''((P_1; P_2) \xrightarrow{\beta} P'' \text{ and } red(P''[\alpha \rightsquigarrow Q]) = P')$$

by choosing $P'' = (\tilde{P}; P_2)$.

For case 2 we obtain

$$\exists P''(P_2 \xrightarrow{\beta} P'' \text{ and } red(P''[\alpha \rightsquigarrow Q]) = F)$$

by the induction hypothesis. By Lemma 2.26 we have $P_1 \in \surd$ since $red(P_1[\alpha \rightsquigarrow Q]) \in \surd$. Hence we obtain

$$\exists P''((P_1; P_2) \xrightarrow{\beta} P'' \text{ and } red(P''[\alpha \rightsquigarrow Q]) = F).$$

$\underline{P = (P_1 \|_A P_2)}$:

Assume $red((P_1 \|_A P_2)[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} P'$ for some $P' \in G\Sigma$. Again we have four cases:

Case 1: $\alpha \notin A$ and $\beta \notin A$. Then we have

$$(red(P_1[\alpha \rightsquigarrow Q]) \|_A red(P_2[\alpha \rightsquigarrow Q])) \xrightarrow{\beta} P'$$

by assertion 1) of Lemma 2.20. Since $\beta \notin A$ we have

$$P' = (E \|_A red(P_2[\alpha \rightsquigarrow Q])) \text{ where } red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} E \quad E \in G\Sigma$$

or

$$P' = (red(P_1[\alpha \rightsquigarrow Q]) \|_A F) \text{ where } red(P_2[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} F \quad F \in G\Sigma$$

by Definition 2.21. W.l.o.g. assume the former. Then

$$\exists \tilde{P}\Big(P_1 \xrightarrow{\beta} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow Q]) = E\Big)$$

by the induction hypothesis whence by Definition 2.21

$$\exists \tilde{P}\Big((P_1 \|_A P_2) \xrightarrow{\beta} (\tilde{P} \|_A P_2) \text{ and } red(\tilde{P}[\alpha \rightsquigarrow Q]) = E\Big) \quad (*)$$

since $\beta \notin A$. As $\alpha \notin A$ we have

$$red((\tilde{P} \|_A P_2)[\alpha \rightsquigarrow Q]) = (red(\tilde{P}[\alpha \rightsquigarrow Q]) \|_A red(P_2[\alpha \rightsquigarrow Q]))$$

by assertion 1) of Lemma 2.20 whence

$$red((\tilde{P} \|_A P_2)[\alpha \rightsquigarrow Q]) = (E \|_A red(P_2[\alpha \rightsquigarrow Q])) = P'$$

by $(*)$. Hence we conclude

$$\exists P''\Big((P_1 \|_A P_2) \xrightarrow{\beta} P'' \text{ and } red(P''[\alpha \rightsquigarrow Q]) = P'\Big)$$

by choosing $P'' = (\tilde{P} \|_A P_2)$.

Case 2: $\beta \notin A$ and $\alpha \in A$. Since $\alpha \in A$ we have

$$(red(P_1[\alpha \rightsquigarrow Q]) \|_{A \setminus \{\alpha\} \cup \xi(red(Q))} red(P_2[\alpha \rightsquigarrow Q])) \xrightarrow{\beta} P'$$

by assertion 2) of Lemma 2.20. Now $\beta \notin A$ and $\beta \notin \xi(Q)$ imply $\beta \notin A \setminus \{\alpha\} \cup \xi(Q)$ whence

$$P' = (E \parallel_{A\setminus\{\alpha\}\cup\xi(red(Q))} red(P_2[\alpha \rightsquigarrow Q])) \text{ where } red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} E$$

or

$$P' = (red(P_1[\alpha \rightsquigarrow Q]) \parallel_{A\setminus\{\alpha\}\cup\xi(red(Q))} F) \text{ where } red(P_2[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} F$$

by Definition 2.21. W.l.o.g. assume the former. Then

$$\exists\tilde{P}\Big(P_1 \xrightarrow{\beta} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow Q]) = E\Big)$$

by the induction hypothesis whence

$$\exists\tilde{P}\Big((P_1 \parallel_A P_2) \xrightarrow{\beta} (\tilde{P} \parallel_A P_2) \text{ and } red(\tilde{P}[\alpha \rightsquigarrow Q]) = E\Big)$$

by Definition 2.21 since $\beta \notin A$ by the current case. As $\alpha \in A$ we have

$$red((\tilde{P} \parallel_A P_2)[\alpha \rightsquigarrow Q]) = (red(\tilde{P}[\alpha \rightsquigarrow Q]) \parallel_{A\setminus\{\alpha\}\cup\xi(red(Q))} red(P_2[\alpha \rightsquigarrow Q])) = P'$$

by assertion 2) of Lemma 2.20. We conclude

$$\exists P''\Big((P_1 \parallel_A P_2) \xrightarrow{\beta} \text{ and } red(P''[\alpha \rightsquigarrow Q]) = P'\Big)$$

by choosing $P'' = (\tilde{P} \parallel_A P_2)$.

Case 3: $\beta \in A$ and $\alpha \notin A$. Then

$$(red(P_1[\alpha \rightsquigarrow Q]) \parallel_A red(P_2[\alpha \rightsquigarrow Q])) \xrightarrow{\beta} P'$$

which follows by assertion 1) of Lemma 2.20 since $\alpha \notin A$. Since $\beta \in A$ we have $P' = (E \parallel_A F)$ where

$$red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} E \text{ and } red(P_2[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} F$$

by Definition 2.21. Hence

$$\exists\tilde{P}_1\Big(P_1 \xrightarrow{\beta} \tilde{P}_1 \text{ and } red(\tilde{P}_1[\alpha \rightsquigarrow Q]) = E\Big)$$

and

$$\exists\tilde{P}_2\Big(P_1 \xrightarrow{\beta} \tilde{P}_2 \text{ and } red(\tilde{P}_2[\alpha \rightsquigarrow Q]) = F\Big)$$

by the induction hypothesis. Now

$$P' = (E \parallel_A F) = (red(\tilde{P}_1[\alpha \rightsquigarrow Q]) \parallel_A red(\tilde{P}_2[\alpha \rightsquigarrow Q]))$$

from the two assertions above whence

$$P' = red((\tilde{P}_1 \parallel_A \tilde{P}_2)[\alpha \rightsquigarrow Q])$$

by assertion 1) of Lemma 2.20 since $\alpha \notin A$. Hence

$$\exists P'' \Big((P_1 \parallel_A P_2) \xrightarrow{\beta} P'' \text{ and } red(P''[\alpha \rightsquigarrow Q]) = P' \Big)$$

by choosing $P'' = (\tilde{P}_1 \parallel_A \tilde{P}_2)$.

Case 4: $\beta \in A$ and $\alpha \in A$. Since $\alpha \in A$

$$(red(P_1[\alpha \rightsquigarrow Q]) \parallel_{A \setminus \{\alpha\} \cup \xi(red(Q))} red(P_2[\alpha \rightsquigarrow Q])) \xrightarrow{\beta} P'$$

by assertion 2) of Lemma 2.20.
First assume $\alpha \neq \beta$. Then $\beta \in A$ by the current case and $\alpha \neq \beta$ imply $\beta \in A \setminus \{\alpha\} \cup \xi(Q)$. Hence $P' = (E \parallel_{A \setminus \{\alpha\} \cup \xi(Q)} F)$ where

$$red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} E \text{ and } red(P_2[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} F$$

by Definition 2.21. This implies

$$\exists \tilde{P}_1 \Big(P_1 \xrightarrow{\beta} \tilde{P}_1 \text{ and } red(\tilde{P}_1[\alpha \rightsquigarrow Q]) = E \Big)$$

and

$$\exists \tilde{P}_2 \Big(P_2 \xrightarrow{\beta} \tilde{P}_2 \text{ and } red(\tilde{P}_2[\alpha \rightsquigarrow Q]) = F \Big)$$

by the induction hypothesis. Since $\beta \in A$ by the current case we have

$$\exists \tilde{P}_1 \exists \tilde{P}_2 ((P_1 \parallel_A P_2) \xrightarrow{\beta} (\tilde{P}_1 \parallel_A \tilde{P}_2))$$

by Definition 2.21. Since

$$P' = (E \parallel_{A \setminus \{\alpha\} \cup \xi(Q)} F) = (red(\tilde{P}_1[\alpha \rightsquigarrow Q]) \parallel_{A \setminus \{\alpha\} \cup \xi(Q)} red(\tilde{P}_2[\alpha \rightsquigarrow Q]))$$

we have

$$P' = red((\tilde{P}_1 \parallel_A \tilde{P}_2)[\alpha \rightsquigarrow Q])$$

by assertion 2) of Lemma 2.20 since $\alpha \in A$. Hence we conclude

$$\exists P''\Big((P_1 \parallel_A P_2) \xrightarrow{\beta} P'' \text{ and } red(P''[\alpha \rightsquigarrow Q]) = P'\Big)$$

by choosing $P'' = (\tilde{P}_1 \parallel_A \tilde{P}_2)$.

Now assume $\alpha = \beta$. Then we must have

$$(red(P_1[\alpha \rightsquigarrow Q]) \parallel_{A\setminus\{\alpha\}\cup\xi(Q)} red(P_2[\alpha \rightsquigarrow Q])) \overset{\beta}{\not\to} P' \quad (\ddagger)$$

which trivially validates assertion 2). To see this we note that $red(P_i[\alpha \rightsquigarrow Q]) = red(P_i[\beta \rightsquigarrow Q])$ since $\alpha = \beta$ for $i = 1, 2$. Since $\beta \notin \xi(Q)$ by assumption we have $red(P_i[\beta \rightsquigarrow Q]) \overset{\beta}{\not\to}$ for $i = 1, 2$ which by Definition 2.21 implies ($\ddagger$).

<u>$P = fix(x = P_1)$:</u>

Assume $red((fix(x = P_1))[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} P'$. Then

$$fix(x = red(P_1[\alpha \rightsquigarrow Q])) \xrightarrow{\beta} P'$$

by Definition 4.5 and Definition 4.13 whence we must have

$$red(P_1[\alpha \rightsquigarrow Q])[fix(x = red(P_1[\alpha \rightsquigarrow Q]))/x] \xrightarrow{\beta} P'$$

by Definition 2.21. This implies

$$\exists \hat{P} \in G\Sigma\Big(red(P_1[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} \hat{P} \text{ and}$$

$$\hat{P}[fix(x = red(P_1[\alpha \rightsquigarrow Q])/x] = P'\Big) \quad (*)$$

by Lemma 2.25. Further we have

$$\exists \tilde{P} \in G\Sigma(P_1 \xrightarrow{\beta} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow Q]) = \hat{P}) \quad (**)$$

by the induction hypothesis. Hence

$$\exists \tilde{P} \in G\Sigma(P_1[fix(x = P_1)/x] \xrightarrow{\beta} \tilde{P}[fix(x = P_1)/x])$$

which implies

$$\exists \tilde{P} \in G\Sigma(fix(x = P_1) \xrightarrow{\beta} \tilde{P}[fix(x = P_1)/x]) \quad (***)$$

by Definition 2.21. Further we have

$$red((\tilde{P}[fix(x = P_1)/x])[\alpha \rightsquigarrow Q])$$

$$= red(\tilde{P}[\alpha \rightsquigarrow Q])[red((fix(x = P_1))[\alpha \rightsquigarrow Q])/x]$$

by Lemma 2.27

$$= red(\tilde{P}[\alpha \rightsquigarrow Q])[fix(x = red(P_1[\alpha \rightsquigarrow Q]))/x]$$

by Definition 4.5 and Definition 4.13

$$= \hat{P}[fix(x = red(P_1[\alpha \rightsquigarrow Q]))/x]$$

by $(**)$

$$= P'$$

by $(*)$. We conclude

$$\exists P'' \in G\Sigma(fix(x = P_1) \xrightarrow{\beta} P'' \text{ and } red(P''[\alpha \rightsquigarrow Q]) = P')$$

by $(***)$ and choosing $P'' = \tilde{P}[fix(x = P_1)/x]$. ■

The condition, that the considered expressions $P \in G\Sigma$ are uniquely synchronized is crucial for the proof of assertion 3) in Lemma 2.28. Intuitively, the effect of this condition is the following: Consider a term $P = (P_1 \|_{A_1} P_2)$ where $P_1 = (Q_1 \|_{A_2} Q_2)$, $P_i, Q_i \in G\Sigma$ and $A_i \subseteq \mathcal{A}$ for $i = 1, 2$. Then a modification of the synchronization set A_2 might destroy synchronizations between processes that are induced by the terms Q_1 and P_2, i.e. terms which are resided on different syntactic levels (with respect to the nesting depth of parallel composition operators):

Example 2.29
Let $P = (P_1 \|_{A_1} P_2)$ where $P_1 = (Q_1 \|_{A_2} Q_2)$ and $Q_1 = \beta$, $Q_2 = \gamma$, $P_2 = \beta$, $A_1 = \{\beta\}$ and $A_2 = \{\alpha\}$. We have $\alpha \notin A_1$ whence the condition $(\alpha \in A_1 \wedge \beta \notin A_1) \Rightarrow \beta \notin \xi(Q)$ would be satisfied for any $Q \in R\Delta$. But we have $red(P[\alpha \rightsquigarrow \beta]) \overset{\beta}{\not\rightarrow}$. Note that P is not uniquely synchronized. In this example, the condition of unique synchronization for P would enforce either $A_1 = A_2 = \emptyset$ or $A_1 = A_2 = \{\alpha, \beta, \ldots\}$. In the former case we would have $red(P[\alpha \rightsquigarrow \beta]) \xrightarrow{\beta}$ whereas in the latter we would have $P \overset{\beta}{\not\rightarrow}$ validating assertion 3) of Lemma 2.28. □

Lemma 2.30

Let $P \in G\Sigma$ be a process expression. Then we have

1) If $\beta \notin \chi(P)$ then, for all $P' \in G\Sigma$, we have

$$P \xrightarrow{\alpha} P' \Rightarrow red(P[\alpha \leadsto \beta]) \xrightarrow{\beta} red(P'[\alpha \leadsto \beta])$$

2) If $\beta \notin alph(P)$ then, for all $P' \in G\Sigma$, we have

$$red(P[\alpha \leadsto \beta]) \xrightarrow{\beta} P' \Rightarrow \exists P''(P \xrightarrow{\alpha} P'' \text{ and } red(P''[\alpha \leadsto \beta]) = P')$$

3) If $P \in UG\Sigma$ then, for all $P' \in UG\Sigma$, we have

$$\forall \alpha \in \chi(P)\Big(P \xrightarrow{\alpha} P' \Rightarrow red(P[\alpha \leadsto \beta]) \xrightarrow{\beta} red(P'[\alpha \leadsto \beta])\Big)$$

Proof: All assertions are proved by structural induction on $P \in G\Sigma$ ($P \in UG\Sigma$ respectively). We only show the cases where the proof differs substantially from the proof of the previous lemma.

For assertion 1) we show

<u>$P = \alpha$:</u>

$\alpha \xrightarrow{\alpha} 0$, $red(P[\alpha \leadsto \beta]) = \beta$ whence $red(P[\alpha \leadsto \beta]) \xrightarrow{\beta} 0$ which completes this case since $red(0[\alpha \leadsto \beta]) = 0$.

<u>$P = (P_1 \|_A P_2)$:</u>

Assume $(P_1 \|_A P_2) \xrightarrow{\alpha} P'$ for some $P' \in G\Sigma$. We distinguish two cases:

Case 1: $\alpha \notin A$. Then

$$P' = (P_1' \|_A P_2) \text{ where } P_1 \xrightarrow{\alpha} P_1'$$

or

$$P' = (P_1 \|_A P_2') \text{ where } P_2 \xrightarrow{\alpha} P_2'$$

by Definition 2.21. W.l.o.g. assume the former. Then

$$red(P_1[\alpha \leadsto \beta]) \xrightarrow{\beta} red(P_1'[\alpha \leadsto \beta])$$

by the induction hypothesis and as $\beta \notin \chi(P_1)$. By the conditions of the lemma $\beta \notin \chi(P)$ which implies $\beta \notin A$. Hence

$$(red(P_1[\alpha \rightsquigarrow \beta])\|_A red(P_2[\alpha \rightsquigarrow \beta])) \xrightarrow{\beta} (red(P_1'[\alpha \rightsquigarrow \beta])\|_A red(P_2[\alpha \rightsquigarrow \beta]))$$

by Definition 2.21. Since $\alpha \notin A$ we conclude

$$red((P_1\|_A P_2)[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red((P_1'\|_A P_2)[\alpha \rightsquigarrow \beta])$$

by assertion 1) of Lemma 2.20.

Case 2: $\alpha \in A$. Then $P' = (P_1'\|_A P_2')$ whence

$$red(P_1[\alpha \rightsquigarrow \beta]) \xrightarrow{\beta} red(P_1'[\alpha \rightsquigarrow \beta])$$

and

$$red(P_2[\alpha \rightsquigarrow \beta]) \xrightarrow{\beta} red(P_2'[\alpha \rightsquigarrow \beta])$$

by the induction hypothesis and $\beta \notin \chi(P_i)$ $(i = 1, 2)$. Hence

$$(red(P_1[\alpha \rightsquigarrow \beta])\|_{A\setminus\{\alpha\}\cup\{\beta\}} red(P_2[\alpha \rightsquigarrow \beta])) \xrightarrow{\beta}$$

$$(red(P_1'[\alpha \rightsquigarrow \beta])\|_{A\setminus\{\alpha\}\cup\{\beta\}} red(P_2'[\alpha \rightsquigarrow \beta]))$$

by Definition 2.21. Since $\alpha \in A$ we conclude

$$red((P_1\|_A P_2)[\alpha \rightsquigarrow \beta]) \xrightarrow{\beta} red((P_1'\|_A P_2')[\alpha \rightsquigarrow \beta])$$

by assertion 2) of Lemma 2.20.

For assertion 2) we show

$\underline{P = \gamma:}$

Case 1: $\gamma = \alpha$. Then $red(\alpha[\alpha \rightsquigarrow \beta]) = \beta$. By Definition 2.21 we have $P' = 0$. We choose $P'' = 0$. Then $P \xrightarrow{\alpha} P''$ and $red(P''[\alpha \rightsquigarrow \beta]) = red(0[\alpha \rightsquigarrow \beta]) = 0 = P'$.

Case 2: $\gamma = \beta$: cannot occur due to the condition $\beta \notin \xi(P)$ of the lemma.

Case 3: $\gamma \neq \alpha$ and $\gamma \neq \beta$: The implication is trivially true.

<u>$P = (P_1 \|_A P_2)$:</u>

Assume $red((P_1 \|_A P_2)[\alpha \rightsquigarrow \beta]) \xrightarrow{\beta} P'$ for some $P' \in G\Sigma$. Again we distinguish the two cases:

Case 1: $\alpha \notin A$. Then

$$(red(P_1[\alpha \rightsquigarrow \beta]) \|_A red(P_2[\alpha \rightsquigarrow \beta])) \xrightarrow{\beta} P'$$

by assertion 1) of Lemma 2.20. By the conditions of the lemma follows $\beta \notin A$. Hence

$$P' = (E \|_A red(P_2[\alpha \rightsquigarrow \beta])) \text{ where } red(P_1[\alpha \rightsquigarrow \beta]) \xrightarrow{\beta} E$$

or

$$P' = (red(P_1[\alpha \rightsquigarrow \beta]) \|_A F) \text{ where } red(P_2[\alpha \rightsquigarrow \beta]) \xrightarrow{\beta} F$$

by Definition 2.21. W.l.o.g. assume the former. Then

$$\exists \tilde{P} \Big(P_1 \xrightarrow{\alpha} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow \beta]) = E \Big)$$

by the induction hypothesis as $\beta \notin alph(P_1)$ whence by Definition 2.21

$$\exists \tilde{P} \Big((P_1 \|_A P_2) \xrightarrow{\alpha} (\tilde{P} \|_A P_2) \text{ and } red(\tilde{P}[\alpha \rightsquigarrow \beta]) = E \Big)$$

since $\alpha \notin A$. Now $P' = (E \|_A red(P_2[\alpha \rightsquigarrow \beta]))$ and $\alpha \notin A$ whence we get

$$P' = red((\tilde{P} \|_A P_2)[\alpha \rightsquigarrow \beta])$$

by assertion 1) of Lemma 2.20. Hence we conclude

$$\exists P'' \Big((P_1 \|_A P_2) \xrightarrow{\alpha} P'' \text{ and } red(P''[\alpha \rightsquigarrow \beta]) = P' \Big)$$

by choosing $P'' = (\tilde{P} \|_A P_2)$.

Case 2: $\alpha \in A$. Then

$$(red(P_1[\alpha \rightsquigarrow \beta]) \|_{A \setminus \{\alpha\} \cup \{\beta\}} red(P_2[\alpha \rightsquigarrow \beta])) \xrightarrow{\beta} P'$$

by assertion 2) of Lemma 2.20 whence $P' = (E \|_{A \setminus \{\alpha\} \cup \{\beta\}} F)$ where

$$red(P_1[\alpha \rightsquigarrow \beta]) \xrightarrow{\beta} E \text{ and } red(P_2[\alpha \rightsquigarrow \beta]) \xrightarrow{\beta} F$$

by Definition 2.21. We obtain

$$\exists \tilde{P}_1 \Big(P_1 \xrightarrow{\alpha} \tilde{P}_1 \text{ and } red(\tilde{P}_1[\alpha \rightsquigarrow \beta]) = E \Big)$$

and

$$\exists \tilde{P}_2 \Big(P_2 \xrightarrow{\alpha} \tilde{P}_2 \text{ and } red(\tilde{P}_2[\alpha \rightsquigarrow \beta]) = F \Big)$$

by the induction hypothesis. Hence

$$\exists \tilde{P}_1 \exists \tilde{P}_2 \Big((P_1 \|_A P_2) \xrightarrow{\alpha} (\tilde{P}_1 \|_A \tilde{P}_2) \Big)$$

by Definition 2.21 since $\alpha \in A$. Now since $\alpha \in A$

$$red((\tilde{P}_1 \|_A \tilde{P}_2)[\alpha \rightsquigarrow \beta]) = (red(\tilde{P}_1[\alpha \rightsquigarrow \beta]) \|_{A \setminus \{\alpha\} \cup \{\beta\}} red(\tilde{P}_2[\alpha \rightsquigarrow \beta]))$$

by assertion 2) of Lemma 2.20 whence

$$red((\tilde{P}_1 \|_A \tilde{P}_2)[\alpha \rightsquigarrow \beta]) = P'.$$

We conclude

$$\exists P'' \Big((P_1 \|_A P_2) \xrightarrow{\alpha} P'' \text{ and } red(P''[\alpha \rightsquigarrow \beta]) = P' \Big)$$

by choosing $P'' = (\tilde{P}_1 \|_A \tilde{P}_2)$.

The proof of assertion 3) can easily be reduced to the proof of assertion 1). All induction steps are identical, except of case 1) in the induction step $P = (P_1 \|_A P_2)$ which does not occur in the proof of assertion 3): $\alpha \in \chi(P)$ implies $\alpha \in A$ (by the unique synchronisation of P). ■

Lemma 2.31
Let $P \in G\Sigma$ be a process expression and $\gamma_1, \gamma_2 \in \mathcal{A}$ such that $\gamma_1 \neq \gamma_2$. If $\gamma_1, \gamma_2 \notin alph(P)$ then, for all $P' \in G\Sigma$, we have

1) $P \xrightarrow{\alpha} P' \Rightarrow \exists P''(red(P[\alpha \rightsquigarrow (\gamma_1; \gamma_2)]) \xrightarrow{\gamma_1} P'' \xrightarrow{\gamma_2} red(P'[\alpha \rightsquigarrow (\gamma_1; \gamma_2)])$

2) $\forall P'' \in G\Sigma\Big(red(P[\alpha \leadsto (\gamma_1;\gamma_2)]) \overset{\gamma_1}{\to} P' \overset{\gamma_2}{\to} P'' \Rightarrow$
$\exists \tilde{P}(P \overset{\alpha}{\to} \tilde{P} \text{ and } red(\tilde{P}[\alpha \leadsto (\gamma_1;\gamma_2)]) = P'')\Big)$

Proof: Induction on the structure of $P \in G\Sigma$.

Proof of assertion 1).

$\underline{P \in \{0\} \cup Idf:}$ Trivial.

$\underline{P = \gamma:}$

Case 1: $\gamma \neq \alpha$. Then the implication is trivially true.

Case 2: $\gamma = \alpha$. Then $\alpha \overset{\alpha}{\to} 0$ whence $P' = 0$. Now $red(\alpha[\alpha \leadsto (\gamma_1;\gamma_2)]) = (\gamma_1;\gamma_2)$. We choose $P'' = (0;\gamma_2)$. Then

$$(red(P[\alpha \leadsto (\gamma_1;\gamma_2)]) \overset{\gamma_1}{\to} P'' \overset{\gamma_2}{\to} red(P'[\alpha \leadsto (\gamma_1;\gamma_2)])).$$

$\underline{P = (P_1 + P_2):}$

Assume $(P_1 + P_2) \overset{\alpha}{\to} P'$ for some $P' \in G\Sigma$. Then we obtain

$$P_1 \overset{\alpha}{\to} P' \text{ or } P_2 \overset{\alpha}{\to} P'$$

by Definition 2.21. W.l.o.g. assume the former. Then

$$\exists P''(red(P_1[\alpha \leadsto (\gamma_1;\gamma_2)]) \overset{\gamma_1}{\to} P'' \overset{\gamma_2}{\to} red(P'[\alpha \leadsto (\gamma_1;\gamma_2)])).$$

This implies

$$\exists P''(red((P_1 + P_2)[\alpha \leadsto (\gamma_1;\gamma_2)]) \overset{\gamma_1}{\to} P'' \overset{\gamma_2}{\to} red(P'[\alpha \leadsto (\gamma_1;\gamma_2)])).$$

$\underline{P = (P_1;P_2):}$

Assume $(P_1;P_2) \overset{\alpha}{\to} P'$ for some $P' \in G\Sigma$.

Case 1: $P' = (P_1'; P_2)$ where $P_1 \xrightarrow{\alpha} P_1'$

Case 2: $P' = P_2'$ where $P_2 \xrightarrow{\alpha} P_2'$ and $P_1 \in \surd$

For case 1 we obtain

$$\exists P''(red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P'' \xrightarrow{\gamma_2} red(P_1'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))$$

by the induction hypothesis. This implies

$$\exists P''\Big((red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]); red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \xrightarrow{\gamma_1} (P''; red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))$$

$$\text{and}$$

$$(P''; red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_2} (red(P_1'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]); red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\Big).$$

We obtain

$$\exists P''\Big(red((P_1; P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} (P''; red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \text{ and}$$

$$(P''; red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \xrightarrow{\gamma_2} red((P_1'; P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\Big)$$

by Definition 2.11 and Definition 2.15. Hence we conclude the desired result

$$\exists \tilde{P}(red((P_1; P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} \tilde{P} \xrightarrow{\gamma_2} red((P_1'; P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))$$

by choosing $\tilde{P} = (P''; red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$.

For case 2 we have

$$\exists P''(red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P'' \xrightarrow{\gamma_2} red(P_2'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))$$

by the induction hypothesis. Since $P_1 \in \surd$ we have $red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \in \surd$ by Lemma 2.26. Hence

$$\exists P''((red((P_1; P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P'' \xrightarrow{\gamma_2} red(P_2'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))).$$

$\underline{P = (P_1 \|_A P_2):}$

Assume $(P_1\|_A P_2) \xrightarrow{\alpha} P'$ for some $P' \in G\Sigma$.

Case 1: $\alpha \notin A$. Then

$$P' = (P_1'\|_A P_2) \text{ where } P_1 \xrightarrow{\alpha} P_1'$$

or

$$P' = (P_1\|_A P_2') \text{ where } P_2 \xrightarrow{\alpha} P_2'$$

by Definition 2.21. W.l.o.g. assume the former. Then

$$\exists P''(red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P'' \xrightarrow{\gamma_2} red(P_1'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))$$

by the induction hypothesis. This implies

$$\exists P''\Big((red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \xrightarrow{\gamma_1} (P''\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))$$

$$\text{and}$$

$$(P''\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \xrightarrow{\gamma_2} (red(P_1'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))\Big).$$

since $\gamma_1, \gamma_2 \notin A$ which follows by the condition of the lemma. Since $\alpha \notin A$ we obtain

$$\exists P''\Big(red((P_1\|_A P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} (P''\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))$$

$$\text{and}$$

$$(P''\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \xrightarrow{\gamma_2} red((P_1'\|_A P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\Big).$$

by assertion 1) of Lemma 2.20. We conclude

$$\exists \tilde{P}(red((P_1\|_A P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} \tilde{P} \xrightarrow{\gamma_2} red((P_1'\|_A P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))$$

by choosing $\tilde{P} = (P''\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))$.

Case 2: $\alpha \in A$. Then $P' = (P_1'\|_A P_2')$ where

$$P_1 \xrightarrow{\alpha} P_1' \text{ and } P_2 \xrightarrow{\alpha} P_2'$$

by Definition 2.21. Hence

$$\exists \tilde{P}_1\Big(red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} \tilde{P}_1 \xrightarrow{\gamma_2} red(P_1'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\Big)$$

and

$$\exists \tilde{P}_2 \Big(red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} \tilde{P}_2 \xrightarrow{\gamma_2} red(P_2'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \Big)$$

by the induction hypothesis whence

$$\exists \tilde{P}_1 \exists \tilde{P}_2 \Big((red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \|_{A\setminus\{\alpha\}\cup\{\gamma_1,\gamma_2\}} red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \xrightarrow{\gamma_1}$$

$$(\tilde{P}_1 \|_{A\setminus\{\alpha\}\cup\{\gamma_1,\gamma_2\}} \tilde{P}_2) \xrightarrow{\gamma_2}$$

$$(red(P_1'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \|_{A\setminus\{\alpha\}\cup\{\gamma_1,\gamma_2\}} red(P_2'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \Big)$$

by Definition 2.21. Since $\alpha \in A$

$$\exists \tilde{P}_1 \exists \tilde{P}_2 \Big(red((P_1 \|_A P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} (\tilde{P}_1 \|_{A\setminus\{\alpha\}\cup\{\gamma_1,\gamma_2\}} \tilde{P}_2) \text{ and}$$

$$(\tilde{P}_1 \|_{A\setminus\{\alpha\}\cup\{\gamma_1,\gamma_2\}} \tilde{P}_2) \xrightarrow{\gamma_2} red((P_1' \|_A P_2')[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \Big)$$

by assertion 2) of Lemma 2.20. We conclude

$$\exists P'' \Big(red((P_1 \|_A P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P'' \text{ and}$$

$$P'' \xrightarrow{\gamma_2} red((P_1' \|_A P_2')[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \Big)$$

by choosing $P'' = (\tilde{P}_1 \|_{A\setminus\{\alpha\}\cup\{\gamma_1,\gamma_2\}} \tilde{P}_2)$.

$\underline{P = fix(x = P_1)}$:

Assume $P = fix(x = P_1) \xrightarrow{\alpha} P'$. Then $P_1[fix(x = P_1)/x] \xrightarrow{\alpha} P'$. Hence

$$\exists \hat{P} (P_1 \xrightarrow{\alpha} \hat{P} \text{ and } \hat{P}[fix(x = P_1)/x] = P')$$

by Lemma 2.25. This implies

$$\exists \tilde{P} \Big(red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} \tilde{P} \xrightarrow{\gamma_2} red(\hat{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \Big)$$

by the induction hypothesis. Hence

$$\exists \tilde{P} \Big(red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))/x] \xrightarrow{\gamma_1}$$

$$\tilde{P}[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))/x] \xrightarrow{\gamma_2}$$

$$red(\hat{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))/x]\Big)$$

which implies

$$\exists \tilde{P}\Big(fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \xrightarrow{\gamma_1}$$
$$\tilde{P}[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))/x] \xrightarrow{\gamma_2}$$
$$red(\hat{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))/x]\Big)$$

by Definition 2.21. It follows

$$\exists \tilde{P}\Big(red((fix(x = P_1))[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1}$$
$$\tilde{P}[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))/x] \xrightarrow{\gamma_2}$$
$$red(\hat{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])[red((fix(x = P_1))[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])/x]\Big)$$

by Definition 4.5 and Definition 4.13. We get

$$\exists \tilde{P}\Big(red(fix(x = P_1)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1}$$
$$\tilde{P}[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))/x] \xrightarrow{\gamma_2}$$
$$red((\hat{P}[fix(x = P_1)/x])[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\Big)$$

by Lemma 2.27. We conclude

$$\exists P''\Big(red(fix(x = P_1)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1}$$
$$P'' \xrightarrow{\gamma_2}$$
$$red(P'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\Big)$$

by choosing $P'' = \tilde{P}[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))/x]$.

Proof of assertion 2).

$\underline{P \in \{0\} \cup Idf}$: Trivial.

$\underline{P = \gamma}$:

Case 1: $\gamma \neq \alpha$. The $red(\gamma[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = \gamma$. Hence the implication is trivially true since $\gamma_1, \gamma_2 \notin \xi(P)$ by the condition of assertion 2).

Case 2: $\gamma = \alpha$. Then $red(\alpha[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = (\gamma_1;\gamma_2)$. Hence we have $P' = (0;\gamma_2)$ and $P'' = 0$. On the other hand $\alpha \xrightarrow{\alpha} 0$. We choose $\tilde{P} = 0$. Hence

$$P \xrightarrow{\alpha} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = P''.$$

<u>$P = (P_1 + P_2)$:</u>

Assume $red((P_1 + P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P' \xrightarrow{\gamma_2} P''$ for $P', P'' \in G\Sigma$. We obtain

$$(red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) + red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \xrightarrow{\gamma_1} P' \xrightarrow{\gamma_2} P''.$$

Hence we obtain

$$red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P' \xrightarrow{\gamma_2} P''$$

or

$$red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P' \xrightarrow{\gamma_2} P''.$$

W.l.o.g. assume the former. Then

$$\exists \tilde{P}(P_1 \xrightarrow{\alpha} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = P'')$$

by the induction hypothesis. This implies

$$\exists \tilde{P}((P_1 + P_2) \xrightarrow{\alpha} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = P'').$$

<u>$P = (P_1;P_2)$:</u>

Assume $red((P_1;P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P' \xrightarrow{\gamma_2} P''$ for $P', P'' \in G\Sigma$. We obtain

$$(red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]); red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \xrightarrow{\gamma_1} P' \xrightarrow{\gamma_2} P''.$$

Case 1)

$$P' = (E; red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \text{ where } red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} E$$

and

$$P'' = (E'; red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \text{ where } E \xrightarrow{\gamma_2} E'$$

(Note that $red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1}$ implies $E \notin \surd$ since $\gamma_1 \notin \xi(P)$)

Case 2)

$$P' = F \text{ where } red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} F \text{ and } red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \in \surd$$

and

$$P'' = F' \text{ where } F \xrightarrow{\gamma_2} F'$$

For case 1 we have

$$red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} E \xrightarrow{\gamma_2} E'$$

whence we get

$$\exists \tilde{P}(P_1 \xrightarrow{\alpha} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = E')$$

by the induction hypothesis. This implies

$$\exists \tilde{P}((P_1;P_2) \xrightarrow{\alpha} (\tilde{P};P_2)) \quad (*)$$

and

$$(red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]); red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) = (E'; red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))$$

which gives

$$red((\tilde{P};P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = (E'; red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) = P'' \quad (**)$$

Taking $(*)$ and $(**)$ together we obtain

$$\exists \tilde{P}((P_1;P_2) \xrightarrow{\alpha} (\tilde{P};P_2) \text{ and } red((\tilde{P};P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = P'')$$

whence we obtain

$$\exists \tilde{P}((P_1;P_2) \xrightarrow{\alpha} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = P'').$$

For case 2 we obtain

$$\exists \tilde{P}(P_2 \xrightarrow{\alpha} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = F')$$

by the induction hypothesis. From $red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \in \surd$ we have $P_1 \in \surd$ by Lemma 2.26. Hence we obtain

$$\exists \tilde{P}((P_1;P_2) \xrightarrow{\alpha} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = F').$$

<u>$P = (P_1 \|_A P_2)$:</u>

Assume $red((P_1\|_A P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P' \xrightarrow{\gamma_2} P''$ for some $P', P'' \in G\Sigma$.

Case 1) $\alpha \notin A$. Then

$$(red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \xrightarrow{\gamma_1} P' \xrightarrow{\gamma_2} P''$$

by assertion 2) of Lemma 2.20. By the condition of the lemma follows $\gamma_1, \gamma_2 \notin A$. We distinguish two sub-cases:

Case (I)

$$P' = (E\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \text{ where } red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} E$$

and

$$P'' = (E'\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \text{ where } E \xrightarrow{\gamma_2} E'$$

(Note that $red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \overset{\gamma_2}{\not\rightarrow}$ since $\gamma_2 \notin \xi(P)$)

Case (II)

$$P' = (red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\|_A F) \text{ where } red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} F$$

and

$$P'' = (red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\|_A F') \text{ where } F \xrightarrow{\gamma_2} F'$$

(Note that $red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \overset{\gamma_2}{\not\rightarrow}$ since $\gamma_2 \notin \xi(P)$)

W.l.o.g we only consider case (I), i.e. we have

$$red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} E \xrightarrow{\gamma_2} E'$$

whence we get

$$\exists \tilde{P}(P_1 \overset{\alpha}{\to} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = E').$$

This implies

$$\exists \tilde{P}((P_1\|_A P_2) \overset{\alpha}{\to} (\tilde{P}\|_A P_2)) \quad (*)$$

since $\alpha \notin A$ and

$$(red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) = (E'\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))$$

which gives

$$red((\tilde{P}\|_A P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = (E'\|_A red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) = P'' \quad (**)$$

by assertion 1) of Lemma 2.20. Taking $(*)$ and $(**)$ together we obtain

$$\exists \tilde{P}((P_1\|_A P_2) \overset{\alpha}{\to} (\tilde{P}\|_A P_2) \text{ and } red((\tilde{P}\|_A P_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = P'')$$

whence

$$\exists \tilde{P}((P_1\|_A P_2) \overset{\alpha}{\to} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = P'').$$

Case 2) $\alpha \in A$. Then

$$(red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\|_{A\setminus\{\alpha\}\cup\{\gamma_1,\gamma_2\}} red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \overset{\gamma_1}{\to} P' \overset{\gamma_2}{\to} P''$$

by assertion 2) of Lemma 2.20. Since $\gamma_1 \in A \setminus \{\alpha\} \cup \{\gamma_1,\gamma_2\}$ we have $P' = (E\|_{A\setminus\{\alpha\}\cup\{\gamma_1,\gamma_2\}} F)$ where

$$red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \overset{\gamma_1}{\to} E \text{ and } red(P_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \overset{\gamma_1}{\to} F.$$

Since $P' \overset{\gamma_2}{\to} P''$ and $\gamma_2 \in A \setminus \{\alpha\} \cup \{\gamma_1,\gamma_2\}$ we have $P'' = (E'\|_{A\setminus\{\alpha\}\cup\{\gamma_1,\gamma_2\}} F')$ where

$$E \overset{\gamma_2}{\to} E' \text{ and } F \overset{\gamma_2}{\to} F'.$$

Hence

$$\exists \tilde{P}_1\Big(P_1 \overset{\alpha}{\to} \tilde{P}_1 \text{ and } red(\tilde{P}_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = E'\Big)$$

and

$$\exists \tilde{P}_2\Big(P_2 \overset{\alpha}{\to} \tilde{P}_2 \text{ and } red(\tilde{P}_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = F'\Big)$$

by the induction hypothesis. Since $\alpha \in A$

$$\exists \tilde{P}_1 \exists \tilde{P}_2 \Big((P_1 \|_A P_2) \xrightarrow{\alpha} (\tilde{P}_1 \|_A \tilde{P}_2) \Big)$$

by Definition 2.21. From assertion 2) of Lemma 2.20 follows

$$red((\tilde{P}_1 \|_A \tilde{P}_2)[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) =$$

$$(red(\tilde{P}_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \|_{A \setminus \{\alpha\} \cup \{\gamma_1,\gamma_2\}} red(\tilde{P}_2[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) =$$

$$(E' \|_{A \setminus \{\alpha\} \cup \{\gamma_1,\gamma_2\}} F') = P''$$

whence we conclude

$$\exists \tilde{P} \Big((P_1 \|_A P_2) \xrightarrow{\alpha} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = P'' \Big)$$

by choosing $\tilde{P} = (\tilde{P}_1 \|_A \tilde{P}_2)$.

$\underline{P = fix(x = P_1)}$:

Assume $red((fix(x = P_1))[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P' \xrightarrow{\gamma_2} P''$. Then

$$fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \xrightarrow{\gamma_1} P' \xrightarrow{\gamma_2} P''$$

Hence we must have

$$red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))/x] \xrightarrow{\gamma_1} P' \xrightarrow{\gamma_2} P''$$

by Definition 2.21. By Lemma 2.25

$$\exists \hat{P} \Big(red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} \hat{P} \text{ and}$$

$$\hat{P}[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))/x] = P' \Big)$$

Since $P' \xrightarrow{\gamma_2} P''$ we have $\hat{P}[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))/x] \xrightarrow{\gamma_2} P''$ whence

$$\exists \tilde{P} \Big(\hat{P} \xrightarrow{\gamma_2} \tilde{P} \text{ and } \tilde{P}[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]))/x] = P'' \Big)$$

by Lemma 2.25. Hence

$$\exists \bar{P}(P_1 \xrightarrow{\alpha} \bar{P} \text{ and } red(\bar{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = \tilde{P})$$

by the induction hypothesis. Hence

$$\exists \bar{P}(P_1[fix(x = P_1)/x] \xrightarrow{\alpha} \bar{P}[fix(x = P_1)/x])$$

which implies

$$\exists \bar{P}(fix(x = P_1) \xrightarrow{\alpha} \bar{P}[fix(x = P_1)/x])$$

by Definition 2.21. Furthermore

$$red((\bar{P}[fix(x = P_1)/x])[\alpha \rightsquigarrow (\gamma_1; \gamma_2)]) =$$

$$red(\bar{P}[\alpha \rightsquigarrow (\gamma_1; \gamma_2)])[red((fix(x = P_1))[\alpha \rightsquigarrow (\gamma_1; \gamma_2)])/x] =$$

$$\tilde{P}[red((fix(x = P_1))[\alpha \rightsquigarrow (\gamma_1; \gamma_2)])/x] =$$

$$\tilde{P}[fix(x = red(P_1[\alpha \rightsquigarrow (\gamma_1; \gamma_2)])/x] =$$

$$P''$$

which gives

$$\exists \tilde{P}\Big(fix(x = P_1) \xrightarrow{\alpha} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1; \gamma_2)]) = P''\Big)$$

by choosing $P'' = \bar{P}[fix(x = P_1)/x]$. ■

Lemma 2.32
Let $P \in G\Sigma$ and $\gamma_1, \gamma_2 \in \mathcal{A}$ such that $\gamma_1 \neq \gamma_2$. If $\gamma_1, \gamma_2 \notin alph(P)$ then, for all $P' \in G\Sigma$, we have

1) $P \xrightarrow{\alpha} P' \Rightarrow \forall i \in \{1,2\}\Big(red(P[\alpha \rightsquigarrow (\gamma_1 + \gamma_2)]) \xrightarrow{\gamma_i} red(P'[\alpha \rightsquigarrow (\gamma_1 + \gamma_2)])\Big)$

2) $\exists i \in \{1,2\}\Big(red(P[\alpha \rightsquigarrow (\gamma_1 + \gamma_2)]) \xrightarrow{\gamma_i} P' \Rightarrow$
$\exists \tilde{P}(P \xrightarrow{\alpha} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1 + \gamma_2)])) = P')\Big)$

Proof: The proof is by an induction on the structure of $P \in G\Sigma$ similarly to the proof of the previous lemma. ■

2.3.2 Discussion

In this thesis, a refinement approach of type "refinement operator", "interleaving", "syntactic action refinement" and "non-atomic" has been chosen in order to support the hierarchical design of *RCSs*. This decision has been taken for the following reasons:

- Refinement through a hierarchy of designs has the disadvantage that no clear separation between different abstraction levels is achieved. This gives rise to confusion in the design process rather than it adds transparency to it. A clear separation of different abstraction levels is considered to be a valuable feature in any design procedure [88, 90]. This can be achieved in refinement operator based approaches.

- An important feature of syntactic action refinement is its effective algorithmic feasibility. For example, syntactic action refinement for standard *PAs* can always implemented effectively since process expressions are strings of finite length even if they denote semantic objects of infinite size like, for example, infinite state transition systems. Clearly, semantic action refinement cannot be implemented effectively for infinite state transition systems. In this thesis, we are concerned with infinite state *LTSs* which excludes the usage of semantic action refinement. Apart from this, syntactic action refinement is easier to understand than semantic action refinement due to its definitional clarity (the reader is invited to compare the two notions in [88] with each other).

The approach of atomic action refinement is much more restrictive than the notion of non-atomic action refinement. Finally, we have argued that *PAs* and *LTSs* together with bisimulation equivalence are a very suitable setting to model *RCSs*. Being an interleaving based setting, our approach to refinement will necessarily be of type interleaving. It is well known, that (non-atomic) syntactic action refinement in interleaving semantics has some particular disadvantages: Many interleaving based behavioural equivalences (like, for example, strong bisimulation equivalence or trace equivalence) that are used for verification [2] are not preserved under the application of syntactic action refinement operators [36]. More precisely, these equivalences are no congruence for the syntactic action refinement operator. This problem is mainly caused by the mutual interference of syntactic action refinement with synchronization issues, in particular, syntactic action refinement operators do not distribute over operators for parallel composition [88]. As will be become clear in Section 4.2, the results of this thesis can be used to alleviate this problem (see also [138]).

3 Verification of Reactive System Designs

The difficulties inherent in designing and realizing $RCSs$ steam from the ability of $RCSs$ to execute subcomponents concurrently which results in the need to ensure an appropriate synchronization of these subcomponents. As $RCSs$ are no monolithic systems but maintain an ongoing interaction with their environment, the environment itself can be viewed as an $RCSs$ that executes concurrently with the $RCSs$ under consideration [103, page 65]. The main problem in designing and analyzing $RCSs$ is thus to intellectually manage the logical complexity of the synchronization and the interaction of many constituent $RCSs$ that execute in parallel. Various methods have been proposed which help to gain an increased confidence in physical $RCSs$ and their designs (see also [163] for a more detailed presentation of the two methods reviewed below some other methods and various references):

- *Dynamic Testing:* Dynamic Testing amounts to compare the behaviour that is observed by executing physical $RCSs$ with a specification of the intended behaviour. This approach has two principal drawbacks: First, the correctness of a range of executions does not admit to infer that all executions of the $RCSs$ are correct. Faults might lurk in the very next execution of the $RCSs$. Second, testing cannot be applied in the design phase as it can only be applied to physical $RCSs$. Consequently, testing does not allow to detect conceptual errors already in the design phase but foremost when $RCSs$ have actually be realized (and with them the error as well). Tracing down the error will be very difficult due to the overwhelming complexity of physical $RCSs$.

- *Symbolic Execution:* In Symbolic execution (also called *simulation*), the design of physical $RCSs$ is executed symbolically and the observed behaviour is compared with a specification of the intended behaviour. The difference to dynamic testing is that models (the designs) of $RCSs$ are executed instead physical $RCSs$. This makes symbolic execution applicable in the design phase and thus evades much of the complexity dynamic testing has to deal with. However, as dynamic testing it is not a method that allows to infer complete correctness of the whole design on the basis of a range of correct simulations.

As has already been discussed in the introduction of this thesis, verification (and also simulation) suffers from the fact that only models of $RCSs$ are verified and that verification of physical $RCSs$ is impossible. At first glimpse, one might assume that dynamic testing is superior in this aspect. However, executions of physical $RCSs$ are always subject to physical influences exerted by the environment and can thus vary with the location (think of testing a physical $RCSs$ in a desert or in the arctic) and also with time (think of testing a physical $RCSs$ in the desert, once at daytime and once at night). Hence, it seems that dynamic testing suffers from similar drawbacks than formal verification.

Neither one of the above mentioned methods nor formal verification allow to ensure the correctness of physical $RCSs$ but all of them can be used to gain an increased confidence in such systems. Of course, the highest benefit comes with the joint application of different methods during the design phase and after the realization of $RCSs$. There is a general consensus that different approaches support each other and that research should be pursued in the particular approaches but also in possible combinations of different methods [24, 48, 53, 163, 216].

This thesis aims to contribute to the research in formal verification of $RCSs$-designs. The remainder of this chapter will thus be concerned with a review of some formalisms that play an important role in verification and a review of the most prominent approaches to $RCSs$-design verification.

3.1 Expressing Properties of Reactive Systems: Modal and Temporal Logics

As has already been explained, formal verification means to show that $RCSs$-designs satisfy some desirable properties. Particularly important formalisms used to specify properties of $RCSs$ are *temporal logics* (TLs for short). General surveys on different TLs and their role in computer science are [15, 213, 40, 68, 120, 169, 190, 83, 62, 192]. TLs are formal languages that extend classical propositional logic or first order predicate logic by a set of *temporal operators*. These operators provide a means to formalize time dependent properties of $RCSs$. For example, TLs allow to formalize properties like "the action α will eventually be executed". Temporal logics can also be seen as a special case of *modal logics* (see for example [57, 37]) which augment classical logics by *modal operators* in order to formalize different "modes of truth" like, for example, that an assertion is "necessarily true" or "possibly true". Interpreting modal operators in time dependent contexts (that is as temporal operators) specializes modal logics to temporal logics (see also [144, pp. 76], [120, pp. 793] and [68, pp. 1048] where the historical evolution of modal and temporal logics is addressed).

TLs can be roughly classified according to the following parameters[14]: "propositional" versus "first order predicate" TLs, "endogenous" versus "exogenous" TLs, "branching time" versus "linear time" TLs, and "past time" versus "future time" TLs.

- *Propositional* TLs versus *first order predicate* TLs. Whereas propositional logic constitutes the non-temporal (or non-modal) part of propositional TLs, the usage of variables, constants, functions, predicates and quantifiers is allowed in first order predicate TLs. In contrary to propositional TLs, first order predicate TLs tend to be highly undecidable (see [68, page 998]).

- *endogenous* TLs versus *exogenous* TLs. Whereas formulas of endogenous TLs (also called "global TLs") are interpreted over one system, exogenous TLs (also called "compositional" TLs) allow to express properties of several different systems within one formula. Most TLs used for the verification of $RCSs$-designs are endogenous. However, exogenous TLs facilitate compositional reasoning: The proof that a $RCSs$-design satisfies a formula can be divided into correctness proofs of constituent parts of the $RCSs$-design.

- *Branching time* TLs versus *linear time* TLs. The way in which the nature of time is conceived gives rise to different types of TLs. Amongst them, the most prominent types of TLs are linear time TLs and branching time TLs. In linear time TLs, the course of time is assumed to be linear, that is, at each moment in

[14]This classification and a detailed discussion on the different types of TLs can be found in [68, pp. 998].

time there exists only one possible future moment. In contrary, time is assumed to have a tree-like nature in branching time TLs, that is, at each moment in time there exist several possible future moments. Prominent linear time TLs are, for example, the "Linear Time Temporal Logic" LTL (see for example [173, 141, 143, 144]) and the "Temporal Logic of Actions" TLA (see for example [126, 76]). Important branching time TLs are, for example, the "computational tree logics" CTL and CTL^* (see for example [41, 73, 69, 75]), the "Hennessy-Milner-Logic" HML [153, 154] and the "Modal Mu-Calculus" $\mu\mathcal{L}$ [119] (see also [192]). Various articles discuss the question whether linear time TLs or branching time TLs are superior in order to reason about $RCSs$-designs [71, 40, 75, 206]. However, using a particular type of logic finally depends on pragmatic rather then theoretical arguments (see also [206]). References to many linear time and branching time TLs can be found in [124, page 27].

Yet another way to conceive the nature of time is to represent time as a set of time instances (moments) equipped with a partial order. This gives rise to so called *partial order* TLs, examples of which are the "Interleaving Set Temporal Logic" $ISTL$ [112] or the "Event Structure Temporal Logic" ESL [168]. Surveys on partial order TLs are, for example, [169, 62].

- *Past time* TLs versus *future time* TLs. Most TLs used in computer science contain only future time temporal operators. Future time temporal operators allow to formalize properties of future moments whereas past time temporal operators allow to express properties of moments in time that have already been passed through by a system. Though the inclusion of past time temporal operators in TLs does in general add no expressive power, they can be useful for compositional verification [98, 133]. Some other articles on the usage of past time temporal operators in TLs are, for example, [188, 127, 128, 108].

The concept of ω-*automata* (see [199] for a survey) is closely related to the concept of TLs. As opposed to ordinary finite state automata FSA (see [170] for a survey), ω-automata are finite state automata that accept "infinite objects" like, for example, words of infinite length or trees of infinite size. The set of infinite words (or infinite trees) accepted by an ω-automaton are conceived to be (abstract) representations of the computation paths (or trees) of that physical $RCSs$ which is modelled by the

ω-automata. Historically, ω-automata have been used in order to develop decision procedures for logics. The fundamental result was given by M. O. Rabin who showed that the *monadic second order logic* $MSOL$ is decidable [176] and much of the work of devising decision procedures for other logics (see for example [195, 160, 161, 196]) has been based on Rabin's results.

3.2 Classical Verification Techniques

In this section, we review some techniques which can be used for the verification of $RCSs$-designs.

- In [141], Z. Manna and A. Pnueli gave a sound and complete axiom system of a linear time temporal logic (LTL) to reason "by hand" about $RCSs$-designs (see also [173] and [68, pp.1054] for further information). However the probability that proofs are incorrect or too complex to be done by hand is huge regarding the enormous complexity of computer systems used nowadays. For this reason deduction systems have been invented to check established hand proofs (called proof checkers) or to prove properties (theorems) of designs automatically by theorem provers (for general information on these topics see [68, pp.1054] and [124, 48, 1, 181, 32]). When applied in the context of program verification a theorem prover investigates the question '$\phi \Rightarrow \varphi$?' where ϕ denotes the $RCSs$-design and φ the specification which is to be checked. Most often however, the axiom systems on which theorem provers are based are incomplete, infinite or the underlying logic is too expressive whence the above question tends to be undecidable. To circumvent this problem interactive theorem provers can be used where human ingenuity is involved in order to guide the search for a correctness proof (see [181, 48] for general information and various references).

- Related to the above approach to verification is a method based on the theory of ω-automaton on infinite words or infinite trees. There the $RCSs$-design and the specification are given as *ω-automata* $\mathcal{A}^M$ and $\mathcal{A}^S$ and one shows that the language $L(\mathcal{A}^M)$ accepted by $\mathcal{A}^M$ is a subset of the language $L(\mathcal{A}^S)$ accepted by $\mathcal{A}^S$, that is, $L(\mathcal{A}^M) \subseteq L(\mathcal{A}^S)$ (see, for example, [207, 205, 122]). Optimization techniques to alleviate the state space explosion problem like, for example, *localization reduction* [124] (where only the parts of the system which are relevant for the verification of the property under consideration are verified) have been investigated. For an overview and various references on this approach see [124].

- In [41, 175], E. M. Clarke and J. Sifakis independently stipulated the investigation of an other method for $RCSs$-design verification, called *model checking.* A model checker is an algorithm which, on input a (operational) $RCSs$-design P and a formula φ of a suitable temporal logic, decides whether P is a model of φ , that is, whether P possesses (or satisfies) the property φ (denoted $P \models \varphi$). Though very appealing being a fully algorithmic (computer-aided) approach, verification based on model checking has two major drawbacks: Firstly, the designs under consideration are usually restricted to have a finite state space. Some sophisticated methods like "local" or "on-the-fly" model checking algorithms, where only the important parts of a system[15] are represented in a demand-driven fashion have been investigated to model check infinite state designs (see, for example, [194, 28, 106, 191]). In general however model checking infinite state designs is undecidable for sufficiently expressive languages[16]. Secondly, the representation of the (finite) state spaces of many designs exceeds all conceivable computational resources due to their gigantic size. Since (classical) model checker usually relay on the exhaustive investigation of state spaces, many $RCSs$-designs cannot be verified using the naive model checking approach. Dramatic improvements have been achieved by optimization techniques: *Ordered binary decision diagrams* ($OBDDs$) [31] can be used to (symbolically) represent state spaces in a concise form. *Symbolic model checking* [150, 33] (surveyed in [47, 99, 45]) made it possible to verify some very large designs not manageable by conventional model checkers [33]. Though sym-

[15]The parts of the system which are relevant with respect to the property to be checked.

[16]The used specification languages highly influence the merits of model checking techniques: For example, though model-checking formulas of the Modal Mu-Calculus [119] is decidable for infinite sequential processes [34] it already becomes undecidable for the (parallel) process algebra $VBPP$ [77]. Generally, as soon as the language for formalizing $RCSs$-designs has Turing strength and the specification logic is sufficiently expressive, almost no interesting properties are decidable.

bolic model checking is meanwhile successfully applied in industry [67, 14] (see also [46]), it is known that many *RCSs*-designs can not be expressed concisely using *OBDDs* [210] whence they usually remain out of the scope of symbolic model checking techniques. In [148], partial order semantics are described on which a technique called *partial order reduction* is based. This technique can be used by model checkers to partition the state space of designs into "equivalent" subsets of states. Consequently, only the representative subsets of the complete state space have to be verified by the model checker [86, 214, 211, 7, 84] (a survey of various partial order methods can be found in [167]). *Abstract interpretations*, a technique invented by R. Cousot and P. Cousot (see for example [55, 54, 56]), are used to collapse subsets of state spaces of designs P into one (or more) abstract states. Depending on the property φ under consideration, it can be sufficient to check that the design P^A which is based on the abstracted state space satisfies φ to conclude that the (concrete) design P satisfies φ. This technique has been exploited to reduce the costs of model checking algorithms[17], for example, in [44, 115, 59]. Some of the above mentioned optimization methods can be applied in combination: For example, symbolic model checking can be enhanced by using abstraction techniques [113] or partial order reduction [7]. Some other optimization techniques, various model checking tools and successful applications of model checkers in practice can be found in [48]. Still, the verification of many *RCSs*-designs remained to be intractable by the model checking approach due to the well known problem of state space explosion: Depending on the semantics used, n actions executed concurrently might require a state space of cardinality $n!$ to semantically represent the according design. In other words, the description of a *RCSs*-design (for example a process algebraic expressions) might be exponentially more concise than the state space of its semantical representation (for example, the transition system induced by a process algebraic expression).

- The situation underlying the above verification methods is that the *RCSs*-design and the specification are already given whence the design verification

[17]Infinite state designs can sometimes be model checked via abstractions by first collapsing the infinite set of states into an (abstract) finite set of states see, for example, [17].

can be carried out. It is implicitly assumed that the specification is reasonable in the sense, that "in principle" it is possible to find a design that satisfies it. This however might not be the case due to inconsistent parts of the specification leading to the question of *satisfiability* classically investigated in the context of logics: Given a formula φ in the logic under consideration, does an object which satisfies φ exist or not? An algorithm $\mathcal{D}_L$ which decides this question for a particular logic L is called a *decision procedure* (with respect to satisfiability) for L (see also [68, pp.1030]). Decision procedures are exploited in another approach to the verification of $RCSs$-designs, called *automated program synthesis.* There, a decision procedure is used to check satisfiability of a formula (the specification) φ of the intended $RCSs$-design. Provided φ is satisfiable a (usually finite state) design that satisfies φ is constructed. Unsatisfiability of φ tantamount to inconsistency of φ whence φ has to be modified accordingly. This approach was applied successfully to generate the *synchronization skeleton* of $RCSs$-designs, that is, the program part where details irrelevant to synchronization are ignored [42, 145, 174, 11] (see also [68, pp.1058] for further discussion).

- Finally, verification can be based on the direct comparison of operational $RCSs$-designs P, Q. Usually, a binary relation $\mathcal{R}$ between designs is defined formalizing those aspects under which P and Q are considered to be semantically equal thereby abstracting away from irrelevant implementation details. Prominent instances of $\mathcal{R}$ are *(strong) bisimulation equivalence*, *(strong) trace equivalence* or *failure trace equivalence* (see Section 2.1.1). If we consider P to be a concrete design (that is an implementation-near design) and Q an abstract design (which exhibits less implementation details), $\mathcal{R}(P, Q)$ formalizes that P and Q share particular semantical properties (subject to the definition of $\mathcal{R}$) though being expressed at different levels of abstraction. For this reason instances of $\mathcal{R}$ are called *implementation relations* in the context of $RCSs$-design verification (for an overview on this topic see, for example, [2]). The predicate $\mathcal{R}(P, Q)$ can also be conceived as one step in a refinement sequence leading towards a precise implementation of an algorithm which might then be translatable into an executable system. This technique is thus based on hierarchies of designs which we called semantics preserving refinements in Section 2.3.

Combinations of some of the above reviewed verification techniques have been investigated, for example, in [164] (proof checking with model checking) or [65, 182] (theorem proving with model checking).

3.3 The Problem with Classical Verification Techniques

To exemplify the main disadvantage implicit in the approaches to $RCSs$-design verification discussed in the previous chapter, we recast in a uniform way:

- Mechanical Verification (see, e.g., [141]). Task: Given formulas φ (denoting the desired properties of the system) and ϕ (denoting the design) in an appropriate logic, use an appropriate deductive system D to derive a proof that φ follows from ϕ, i.e., $\phi \overset{D}{\Rightarrow} \varphi$. We recast this into $\varphi \in 2^{TH_D(\phi)}$, where $TH_D(\phi)$ is the least set that contains ϕ and is closed under $\overset{D}{\Rightarrow}$.

- ω-automata (see, e.g., [124]). Task: Given the description of a system and a specification as ω-automata $\mathcal{A}^M$ and $\mathcal{A}^S$, check $L(\mathcal{A}^M) \subseteq L(\mathcal{A}^S)$. We recast this into $\mathcal{A}^M \in \{\mathcal{A} \mid L(\mathcal{A}) \in 2^{L(\mathcal{A}^S)}\}$.

- Model Checking (see, e.g., [41, 175]). Task: Given a (transition) system P and a specification φ in a logic L, check $P \models^L \varphi$. We recast this into $P \in \{Q \mid Q \models^L \varphi\}$.

- Synthesis of Reactive Systems (see, e.g., [41]). Task: Given the specification φ of the properties to be satisfied by the intended design in a suitable logic L, create by means of a transformation T a (transition) system $T(\varphi)$ such that $T(\varphi) \models^L \varphi$, i.e. $T(\varphi) \in \{Q \mid Q \models^L \varphi\}$.

- Semantics-Preserving Refinement (see, e.g., [2]). Task: Given the (transition) systems P and Q, check that $\mathcal{R}(P, Q)$ for a desired implementation relation $\mathcal{R}$. We recast this into $P \in Q^{\mathcal{R}}$ where $Q^{\mathcal{R}}$ is the least set which contains Q and is closed under the relation $\mathcal{R}$, i.e. $P \in S^{\mathcal{R}}$ and $\mathcal{R}(P, Q)$ implies $Q \in S^{\mathcal{R}}$.

We observe that it is possible to define the task to be solved in all the verification methods mentioned above by

$$D \in \mathcal{Q}(S).$$

Verification in the Hierarchical Development of Reactive Systems.

In the case of mechanical verification, D denotes a specification whereas D denotes a $RCSs$-design in all other cases mentioned above. $\mathcal{Q}(S)$ represents the theory of a system in the case of mechanical verification whereas $\mathcal{Q}$ is a language parameterized by a specification S in the other approaches. What is missing in all the above discussed verification methods are transformations ref and Ref which satisfy the property induced by the diagram shown in Figure 3.3.

$$\begin{tikzcd}
D \arrow[d, "ref"'] & \in & \mathcal{Q}(S) \arrow[d, "Ref"] \\
ref(D) & \in & \mathcal{Q}(Ref(S))
\end{tikzcd}$$

Figure 4: Transformations of Designs and Specifications

The relevance of the existence of such transformations comes form the following observation: The (coarse) live cycle of a $RCSs$-design usually consists of its development and its maintenance. Eventually the design might be scraped. However, meanwhile its (hopefully long) lifetime, the design will most likely be subject of repeated alignment to new requirements and resources. This involves an adaptation of the old specification to comprise the new requirements imposed on the design and of course the adaption of the design itself. Any change of the old design or the old specification however requires a reapplication of the whole verification procedure to the new design and the new specification if one of the above verification methods is used. This means that the information $D \in \mathcal{Q}(S)$ cannot be reused whence every change of the design or the specification requires to apply the chosen verification method from the very scratch again. Hence, the above presented verification methods do not support the procedure of hierarchical design system development under which we understood the stepwise development of a design (where a design is developed by successive enrichment with details) and design maintenance (where parts of the design are changed or extended) when used in isolation. The remainder of this thesis is devoted to the introduction of such transformations.

4 Verification in the Hierarchical Development of Reactive Systems

As we have seen in Section 2.3, the method of syntactic action refinement (in process algebras) supports and facilitates the hierarchical development of (potentially infinite state) transition systems: Actions α that occur in process expressions P are refined by more complex expressions Q thereby yielding more detailed process descriptions $P[\alpha \rightsquigarrow Q]$. Considering a verification setting based on process algebras and logics (like, for example, the model checking approach), the following problems arise: Knowing that the system induced by a process term P satisfies a particular formula does neither tell us which formulas are satisfied by the system induced by the refined term $P[\alpha \rightsquigarrow Q]$, nor which system satisfies a refined specification.

In Section 4.1, we define syntactic action refinement for Modal Mu-Calculus formulas (denoted by $\varphi[\alpha \rightsquigarrow Q]$). In Section 4.2 we show that the assertion

$$P \models \varphi \Leftrightarrow P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q] \quad (*)$$

holds under certain conditions (see Section 4.2, Theorem 4.42). In the above assertion, P and Q are $R\Sigma$- and $R\Delta$-process terms respectively, and φ is a Modal Mu-Calculus formula which in addition might contain refinement operators. Intuitively, assertion $(*)$ says, that the transition system induced by a term P satisfies a specification φ (denoted by $P \models \varphi$) if and only if the transition system induced by the refined term $P[\alpha \rightsquigarrow Q]$ satisfies the refined specification $\varphi[\alpha \rightsquigarrow Q]$. Assertion $(*)$ embodies what we understand by *simultaneous syntactic action refinement*: The satisfaction relation "$\models$" is preserved ($\Rightarrow$-direction) and reflected ($\Leftarrow$-direction) under the simultaneous application of refinement operators to process expressions and formulas.

To motivate the investigations carried out in Section 4.2 we consider the following example. Assume, that a given process term P induces the system shown in Figure 5 (a) and that the terms $Q_\alpha, \ldots, Q_\delta$ (which induce the subsystems in the according rectangles) are subexpressions of the term P. Let us only consider the actions occurring in the term Q_δ. We observe, that the system in Figure 5 (a) has the temporal property "eventually, the action δ_1 or the action δ_2 is executed". This property can be denoted by a formula φ of the Modal Mu-Calculus. Further assume, that the

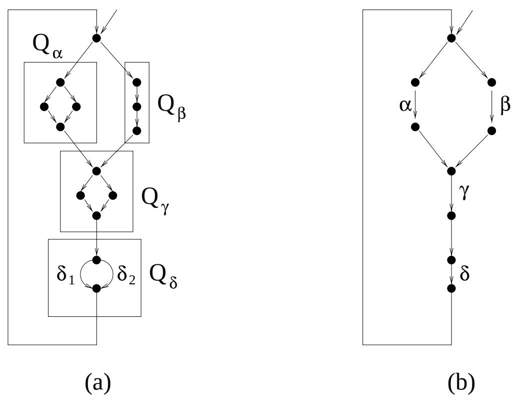

Figure 5: System Changes Induced by Syntactic Action Refinement

process term P_s induces the system shown in Figure 5 (b). This system has the property "eventually, the action δ is executed" which can be expressed by a Modal Mu-Calculus formula φ_s.

The process term P arises from the term P_s by four syntactic action refinements, namely

$$P = P_s[\alpha \rightsquigarrow Q_\alpha][\beta \rightsquigarrow Q_\beta][\gamma \rightsquigarrow Q_\gamma][\delta \rightsquigarrow Q_\delta].$$

The definition of SAR for the Modal Mu-Calculus proposed in this paper will be such that

$$\varphi = \varphi_s[\alpha \rightsquigarrow Q_\alpha][\beta \rightsquigarrow Q_\beta][\gamma \rightsquigarrow Q_\gamma][\delta \rightsquigarrow Q_\delta].$$

By assertion $(*)$ we thus immediately know that

$$P_s \models \varphi_s \text{ if and only if } P \models \varphi.$$

As the above example indicates (and as has already been discussed in Section 1.3), assertion $(*)$ can be used for the following:

- It offers the possibility of simplifying the verification task by repeatedly applying $(*)$ as was shown in the example. Instead of checking that
$$P = P_s[\alpha \rightsquigarrow Q_\alpha]\ldots[\delta \rightsquigarrow Q_\delta] \models \varphi_s[\alpha \rightsquigarrow Q_\alpha]\ldots[\delta \rightsquigarrow Q_\delta] = \varphi$$
we only have to check $P_s \models \varphi_s$. We will see, how this application of assertion $(*)$ can enhance model checking techniques.

- Let us assume that the specification of a system is developed incrementally and that the initial specification φ_s is now given more details by refining it to $\varphi = \varphi_s[\alpha \rightsquigarrow Q_\alpha]\ldots[\delta \rightsquigarrow Q_\delta]$. When we look for an implementation of φ, we only have to supply an implementation P_s for φ_s thereby automatically obtaining a process $P = P_s[\alpha \rightsquigarrow Q_\alpha]\ldots[\delta \rightsquigarrow Q_\delta]$ such that $P \models \varphi$. We will see, that this application of assertion $(*)$ can be used as a method of *a priori verification*, that is, assertion $(*)$ allows to incorporate verification into the procedure of hierarchical system development.

Existing verification methods can be used to verify $P \models \varphi$. Subsequently, assertion $(*)$ can be used in the procedure of (a priori correct) hierarchical system engineering: Via assertion $(*)$, the refinement of the original property φ into the new property $\varphi[a \rightsquigarrow Q]$ automatically supplies the system $P[a \rightsquigarrow Q]$ such that $P[a \rightsquigarrow Q] \models \varphi[a \rightsquigarrow Q]$.

Section 4.3 is devoted to investigations of how the conditions under which assertion $(*)$ holds can be alleviated. In Section 4.4 we are concerned with complexity issues of the method of simultaneous syntactic action refinement. An extension of the Modal Mu-Calculus (called the generalized Modal Mu-Calculus, $\mu\mathcal{L}_g$) is introduced and syntactic action refinement is defined for it. Finally, we show that an assertion as assertion $(*)$ can be proved for $\mu\mathcal{L}_g$. We will demonstrate, that employing $\mu\mathcal{L}_g$ instead of the standard Modal Mu-Calculus makes simultaneous syntactic action refinement much more efficient.

4.1 Specifying Reactive Systems: The Modal Mu-Calculus and Syntactic Action Refinement for the Modal Mu-Calculus

The *Modal Mu-Calculus*, $\mu\mathcal{L}$ as developed by [119] is a particularly expressive branching time temporal logic as most of the logics commonly used to reason about reactive systems can be translated into it [72, 58]. In fact, every logic over transition systems which does not distinguish bisimular systems and is translatable into the monadic second order logic $MSOL$ can be translated into $\mu\mathcal{L}$ [107]. The Modal Mu-Calculus $\mu\mathcal{L}$ is thus often considered as a generic "assembly" logic [28, 20, 53, 59, 34]. In [160, 161, 70] it was shown, that when interpreted on infinite trees $\mu\mathcal{L}$ is equally expressive as nondeterministic ω-automata on infinite trees and hence as powerful as $MSOL$ on those structures. With regard to model checking, $\mu\mathcal{L}$ is one of the primarily considered logics (extensive literature on $\mu\mathcal{L}$-model checking can be found in [1]. See also [194, 28, 44, 106, 191, 18, 59, 45, 217]).

From a practical point of view we note that many model checking tools like the NCSU Concurrency Workbench [52], the (Edinburgh) Concurrency Workbench [51] or the Concurrency Factory [50] are based on $\mu\mathcal{L}$. Complete axiomatizations of $\mu\mathcal{L}$ have been given in [208, 209] opening this logic to the field of theorem proving. Proof systems based on $\mu\mathcal{L}$ have been developed, for example, with respect to transition systems [9] and state-chart processes [132]. PVS (Prototype Verification System) is an interactive theorem prover based on higher order logic which in addition uses a $\mu\mathcal{L}$-decision procedure for a well defined fragment of PSV. The DFA&OPT-MetaFrame tool kit can be used to synthesize efficient data flow analysis algorithms from specifications given in $\mu\mathcal{L}$ with additional backward modalities [118, 188]. In addition, $\mu\mathcal{L}$ was applied in the field of automated program synthesis: $\mu\mathcal{L}$ is used in [116, 123] to synthesize reactive systems. Further, $\mu\mathcal{L}$ has found application in the field of artificial intelligence [85].

Definition 4.1 (Modal Mu-Calculi)
The (negation free form of the) Modal Mu-Calculus $\mu\mathcal{L}$ (see [119]) is generated by the grammar

$$\varphi ::= \top \mid \bot \mid Z \mid (\varphi_1 \vee \varphi_2) \mid (\varphi_1 \wedge \varphi_2) \mid [\alpha]\varphi \mid \langle\alpha\rangle\varphi \mid \nu Z.\varphi \mid \mu Z.\varphi$$

where α ranges over the set $\mathcal{A}$ and Z ranges over a fixed set Var of variables.

Let $\mathcal{O}n$ be the class of ordinals ranged over by η, κ, λ, I. The approximative Modal Mu-Calculus App (see [119]) is the language generated by the grammar

$$\varphi ::= \top \mid \bot \mid Z \mid \bigvee_{\kappa \in I} \varphi_\kappa \mid \bigwedge_{\kappa \in I} \varphi_\kappa \mid [\alpha]\varphi \mid \langle\alpha\rangle\varphi \mid \nu^\lambda Z.\varphi \mid \mu^\lambda Z.\varphi$$

where $\kappa, \lambda, I \in \mathcal{O}n$. For $I = \{1,2\}$ we let $\bigvee_{\kappa \in I} \varphi_\kappa := (\varphi_1 \vee \varphi_2)$ and $\bigwedge_{\kappa \in I} \varphi_\kappa := (\varphi_1 \wedge \varphi_2)$. Let $\mu\mathcal{L}^{App}$ be the language generated by the grammar which consists of all clauses that are used in the grammars of the languages $\mu\mathcal{L}$ and App. □

As an aside, please note that for $\lambda, I \in \mathcal{O}n$ we have that $\lambda \in I \Leftrightarrow \lambda < I$.

The modal operators of $\mu\mathcal{L}^{App}$ can be conceived as follows: A $\mu\mathcal{L}^{App}$-formula of the form $[\alpha]\varphi$ is satisfied by a process P which, by committing any α-transition, must evolve to a process P' which satisfies φ. Dually, A $\mu\mathcal{L}^{App}$-formula of the form $\langle\alpha\rangle\varphi$

is satisfied by a process P that is able to commit an α-transition thereby evolving to a process P' which satisfies φ. The (very rough) intuition behind a maximum fixed point operator formula $\varphi = \nu Z.\varphi'$ is, that a process satisfies φ if it always satisfies φ', regardless of the transitions P might execute. Dually, a process P satisfies a minimum fixed point operator formula $\mu Z.\varphi'$ if P eventually reaches a state where φ' holds. The properties denoted by formulas of the form $\nu^\lambda Z.\varphi'$ (dually $\mu^\lambda Z.\varphi'$) can be thougth of as "approximations" of the property denoted by the formula $\nu Z.\varphi'$ ($\mu Z.\varphi'$ respectively).

We now introduce an operator for syntactic action refinement to the logic $\mu\mathcal{L}^{App}$.

Definition 4.2 (Action refinement for Modal Mu-Calculi)
Let $R\mu\mathcal{L}$ ($R\mu\mathcal{L}^{App}$) be the language generated by the grammar for $\mu\mathcal{L}$ ($\mu\mathcal{L}^{App}$) augmented with the rule $\varphi ::= \varphi[\alpha \rightsquigarrow Q]$ where $Q \in R\Delta$. $\square$

We let σ range over the set $\{\mu, \nu\}$. A *(approximation) fixed point formula* has the form $\sigma Z.\varphi$ ($\sigma^\lambda Z.\varphi$ respectively.) in which σZ *binds* free occurrences of Z in φ. A variable Z is called *free* iff it is not bound. A $R\mu\mathcal{L}^{App}$-formula φ is called *closed* iff every variable Z which occurs in φ is bound. A $R\mu\mathcal{L}^{App}$-formula φ is called *guarded* iff every occurrence of a variable Z in φ lies in the scope of a modality $[\alpha]$ or $\langle\alpha\rangle$. For $L \subseteq \mu\mathcal{L}^{App}$, we let $CGL := \{\varphi \in \mu\mathcal{L}^{App} \mid \varphi \text{ is closed and guarded}\}$. Below, we define a function which yields the set of performances that occur in a $R\mu\mathcal{L}^{App}$-formula.

Definition 4.3 (Performance-sets of formulas)
The function $\xi : R\mu\mathcal{L}^{App} \to 2^{\mathcal{A}}$ is defined as follows:

$$\xi(*) := \emptyset \textit{ if } * \in \{\top, \bot\} \cup Var\ , \qquad \xi(\bigotimes \varphi_\kappa) := \bigcup_{\kappa\in I} \xi(\varphi_\kappa) \textit{ if } \bigotimes \in \{\bigvee_{\kappa\in I}, \bigwedge_{\kappa\in I}\}\ ,$$

$$\xi([\alpha]\varphi) := \{\alpha\} \cup \xi(\varphi) \qquad \xi(\langle\alpha\rangle\varphi) := \{\alpha\} \cup \xi(\varphi)\ ,$$

$$\xi(\sigma Z.\varphi) := \xi(\varphi) \qquad \xi(\sigma^\lambda Z.\varphi) := \xi(\varphi)\ ,$$

$$\xi(\varphi[\alpha \rightsquigarrow Q]) := \begin{cases} \xi(\varphi) \setminus \{\alpha\} \cup \xi(Q) & \textit{if } \alpha \in \xi(\varphi) \\ \xi(\varphi) & \textit{else} \end{cases}$$

$\square$

Definition 4.4 (alphabet-disjointness of formulas from process terms)
A formula $\varphi \in R\mu\mathcal{L}^{App}$ is called alphabet-disjoint from a process expression $P \in R\Sigma$ iff $\xi(\varphi) \cap alph(P) = \emptyset$. □

We can now introduce the concept of logical substitution on which the reduction function for $R\mu\mathcal{L}^{App}$-formulas (Definition 4.13) will be based.

Definition 4.5 (Logical substitution for $\mu\mathcal{L}^{App}$)
Let $Q, Q_1, Q_2 \in \Delta$ and $\phi, \varphi, \varphi_\lambda \in \mu\mathcal{L}^{App}$ ($\lambda \in \mathcal{O}n$). The operation of logical substitution, *$(\phi)\{\alpha \rightsquigarrow Q\}$ is defined as follows:*

$$(*)\{\alpha \rightsquigarrow Q\} := * \quad \textit{if } * \in \{\top, \bot\} \cup Var$$

$$(\bigotimes \varphi_\kappa)\{\alpha \rightsquigarrow Q\} := \bigotimes (\varphi_\kappa)\{\alpha \rightsquigarrow Q\} \quad \textit{if } \bigotimes \in \{\bigwedge_{\kappa \in I}, \bigvee_{\kappa \in I}\}$$

$$(\triangle_\beta \varphi)\{\alpha \rightsquigarrow Q\} := \triangle_\beta (\varphi)\{\alpha \rightsquigarrow Q\} \quad \textit{if } \alpha \neq \beta$$

$$(\triangle_\alpha \varphi)\{\alpha \rightsquigarrow Q\} :=$$

$$\begin{cases} \triangle_\beta(\varphi)\{\alpha \rightsquigarrow Q\} & \textit{if } Q = \beta \\ ((\triangle_\gamma(\varphi)\{\alpha \rightsquigarrow Q\})\{\gamma \rightsquigarrow Q_1\} \wedge (\triangle_\delta(\varphi)\{\alpha \rightsquigarrow Q\})\{\delta \rightsquigarrow Q_2\}) & \textit{if } Q = (Q_1 + Q_2) \\ (\triangle_\gamma(\triangle_\delta(\varphi)\{\alpha \rightsquigarrow Q\})\{\delta \rightsquigarrow Q_2\})\{\gamma \rightsquigarrow Q_1\} & \textit{if } Q = (Q_1 ; Q_2) \end{cases}$$

$$(\sigma^\lambda Z.\varphi)\{\alpha \rightsquigarrow Q\} := \sigma^\lambda Z.(\varphi)\{\alpha \rightsquigarrow Q\}$$

$$(\sigma Z.\varphi)\{\alpha \rightsquigarrow Q\} := \sigma Z.(\varphi)\{\alpha \rightsquigarrow Q\}$$

where in each clause $\triangle_\epsilon$ means throughout either $\langle\epsilon\rangle$ or $[\epsilon]$ for all $\epsilon \in \mathcal{A}$. We require that $\gamma, \delta \in Var_{Act}$ are fresh action variables, that is, $\gamma \neq \delta$ and $\gamma, \delta \notin \xi((\varphi)\{\alpha \rightsquigarrow Q\})$ and $\gamma \notin \xi((\triangle_\delta(\varphi)\{\alpha \rightsquigarrow Q\})\{\delta \rightsquigarrow Q_2\})$. □

When applied to a formula φ, the operation of logical substitution $\cdot\{\alpha \rightsquigarrow Q\}$ replaces each occurrence of the action α in φ by the logical structure exhibited by the term Q: As a term $Q = (Q_1 + Q_2)$ can execute actions from both components Q_1 and Q_2, the binary choice operator $+$ is modelled by conjunction. A term $Q = (Q_1; Q_2)$ is logically modelled by appropriate nested sequences of modalities that also reflect the branching information contained in the term Q.

Example 4.6

Let $Q_1 = (\beta; (\gamma + \delta))$ and $Q_2 = ((\beta; \gamma) + (\beta; \delta))$ be process terms and $\phi = \langle\alpha\rangle\top$ be a formula. Then $(\phi)\{\alpha \rightsquigarrow Q_1\} = \langle\beta\rangle(\langle\gamma\rangle\top \wedge \langle\delta\rangle\top)$ whereas $(\phi)\{\alpha \rightsquigarrow Q_2\} = (\langle\beta\rangle\langle\gamma\rangle\top \wedge \langle\beta\rangle\langle\delta\rangle\top)$ □

Example 4.7

Let $\varphi = \mu Z.[\alpha](\langle\beta\rangle Z \vee [\alpha]\bot)$ and $Q = (\delta + \xi)$. Then

$$(\varphi)\{\alpha \rightsquigarrow Q\} = \mu Z.([\delta](\langle\beta\rangle Z \vee ([\delta]\bot \wedge [\xi]\bot)) \wedge [\xi](\langle\beta\rangle Z \vee ([\delta]\bot \wedge [\xi]\bot))). \qquad \square$$

Remark 4.8

To avoid excessive use of brackets we sometimes use the notation $\varphi\{\alpha \rightsquigarrow Q\}$ instead of $(\varphi)\{\alpha \rightsquigarrow Q\}$ if the context avoids ambiguity. □

In the case of $([\alpha]\varphi)\{\alpha \rightsquigarrow Q\}$ and $(\langle\alpha\rangle\varphi)\{\alpha \rightsquigarrow Q\}$ the right-hand side of Definition 4.5 involves $\{\alpha \rightsquigarrow Q\}$ as well as terms of the type $[\alpha]\varphi$ respectively. $\langle\alpha\rangle\varphi$. Hence we must prove that the operation of logical substitution is always defined. For this purpose we introduce the following notation. The *length* $|\cdot| : \mu\mathcal{L}^{App} \to On$ of formulas is given by $|*| := 1$ where $* \in \{\bot, \top\} \cup Var$, $|\bigotimes \varphi_\kappa| := \sup_{\kappa\in I}(|\varphi_\kappa|) + 1$ where $\bigotimes \in \{\bigvee_{\kappa\in I}, \bigwedge_{\kappa\in I}\}$, $|\triangle_\alpha\varphi| := |\varphi| + 1$ where $\triangle_\alpha \in \{[\alpha], \langle\alpha\rangle\}$, $|\sigma Z.\varphi| := |\varphi| + 1$ and $|\sigma^\lambda Z.\varphi| := |\varphi| + 1$.

Example 4.9

Let $\varphi = \sigma^\lambda Z.[\alpha]Z$. Then $|\varphi| = 3$ for all $\lambda \in \mathcal{O}n$. □

The length $|\cdot| : \Delta \to \mathbb{N}$ of process expressions is given by $|\alpha| := 1$ and $|(Q_1 \mathit{op}\, Q_2)| := 1 + |Q_1| + |Q_2|$ where $\mathit{op} \in \{;, +\}$. The relation $\prec \subseteq (\Delta \times \mu\mathcal{L}^{App})^2$ is defined by $(Q_1, \psi) \prec (Q_2, \varphi)$ iff $|Q_1| < |Q_2|$ or ($|Q_1| = |Q_2|$ and $|\psi| < |\varphi|$). By using the relation $\prec$ on the set $(\Delta \times \mu\mathcal{L}^{App})^2$ the effect of decreasing the complexity (length) of Q by the application of the substitution operator $(\varphi)\{\alpha \rightsquigarrow Q\}$ is stronger than the effect of reducing the complexity of φ. The following result shows that the operation of logical substitution is always defined.

Lemma 4.10

Let $\varphi \in \mu\mathcal{L}^{App}$ be a formula and $Q \in \Delta$ be a process expression. Then we have that $(\varphi)\{\alpha \rightsquigarrow Q\} \in \mu\mathcal{L}^{App}$ for any $\alpha \in \mathcal{A}$.

Proof: By well-founded induction on the relation $\prec$. We only show the case where $\varphi = [\alpha]\varphi'$ and $Q = (Q_1 + Q_2)$. Let $\gamma, \delta \in Var_{Act}$ be as required in Definition 4.5.

By Definition 4.5 we have $(\varphi)\{\alpha \rightsquigarrow Q\} =$

$$\Big(([\gamma](\varphi')\{\alpha \rightsquigarrow Q\})\{\gamma \rightsquigarrow Q_1\} \wedge ([\delta](\varphi')\{\alpha \rightsquigarrow Q\})\{\delta \rightsquigarrow Q_2\}\Big) \; (1)$$

Clearly $|\varphi'| < |[\alpha]\varphi'|$ whence $(Q, \varphi') \prec (Q, [\alpha]\varphi')$, hence $(\varphi')\{\alpha \rightsquigarrow Q\}) = \hat{\varphi} \in \mu\mathcal{L}^{App}$. Hence $[\gamma]\hat{\varphi} \in \mu\mathcal{L}^{App}$. As $(Q_1, [\gamma]\hat{\varphi}) \prec (Q, [\alpha]\varphi')$ since $|Q_1| < |Q|$ we obtain $([\gamma]\hat{\varphi})\{\gamma \rightsquigarrow Q_1\} = \hat{\varphi}_1 \in \mu\mathcal{L}^{App}$. A similar argument provides $([\delta]\hat{\varphi})\{\delta \rightsquigarrow Q_2\} = \hat{\varphi}_2 \in \mu\mathcal{L}^{App}$ for the second conjunct of (1). Hence $(\hat{\varphi}_1 \wedge \hat{\varphi}_2) \in \mu\mathcal{L}^{App}$. ■

Lemma 4.11

Let $\varphi \in \mu\mathcal{L}^{App}$, $Q \in \Delta$ and $\alpha \in \mathcal{A}$. Then we have

$$\xi((\varphi)\{\alpha \rightsquigarrow Q\}) := \begin{cases} \xi(\varphi) \setminus \{\alpha\} \cup \xi(Q) & \textit{if } \alpha \in \xi(\varphi) \\ \xi(\varphi) & \textit{else} \end{cases}$$

Proof: By well-founded induction on the relation $\prec$. ■

The following lemma can be seen as the counterpart of Remark 2.13 for the logical framework.

Lemma 4.12

Let $Q_1, Q_2 \in \Delta$ be process expressions and $\varphi \in \mu\mathcal{L}^{App}$ be a formula. Let $\gamma_1, \gamma_2 \in \mathcal{A}$ such that $\gamma_1 \neq \gamma_2$ and $* \in \{+, ;\}$. If $\gamma_1, \gamma_2 \notin \xi((Q_1 * Q_2)) \cup \xi(\varphi)$ then we have that $((\varphi)\{\alpha \rightsquigarrow (\gamma_1 * \gamma_2)\})\{\gamma_2 \rightsquigarrow Q_2\})\{\gamma_1 \rightsquigarrow Q_1\} = (\varphi)\{\alpha \rightsquigarrow (Q_1 * Q_2)\}$.

Proof: By induction on the structure of $\varphi \in \mu\mathcal{L}^{App}$.

We now come to the definition of the logical reduction function.

Definition 4.13 (Logical reduction function for $R\mu\mathcal{L}^{App}$)

Let $Q \in R\Delta$ be a process expression and $\varphi \in R\mu\mathcal{L}^{App}$ be a formula. We define the logical reduction function $\mathcal{R}ed : R\mu\mathcal{L}^{App} \to \mu\mathcal{L}^{App}$ *as follows:*

$$\mathcal{R}ed(*) := * \quad \textit{if } * \in \{\top, \bot\} \cup Var \qquad \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q]) := (\mathcal{R}ed(\varphi))\{\alpha \rightsquigarrow red(Q)\}$$

$$\mathcal{R}ed(\bigotimes \varphi_\kappa) := \bigotimes \mathcal{R}ed(\varphi_\kappa) \quad \textit{if} \bigotimes \in \{\bigwedge_{\kappa\in I}, \bigvee_{\kappa \in I}\}$$

$$\mathcal{R}ed([\beta]\varphi) := [\beta]\mathcal{R}ed(\varphi)\ , \quad \mathcal{R}ed(\langle\beta\rangle\varphi) := \langle\beta\rangle\mathcal{R}ed(\varphi)$$

$$\mathcal{R}ed(\sigma^\lambda Z.\varphi) := \sigma^\lambda Z.\mathcal{R}ed(\varphi) \qquad \mathcal{R}ed(\sigma Z.\varphi) := \sigma Z.\mathcal{R}ed(\varphi) \qquad \Box$$

Some elementary properties of the logical reduction function $\mathcal{R}ed$ are given below. Remark 4.14 states that one application of the reduction function is enough to remove all refinement operators occurring in a formula. It can be conceived as the counterpart of Remark 2.17 for the logical framework.

Remark 4.14
Let $Q \in R\Delta$ be a process expression and $\varphi \in R\mu\mathcal{L}^{App}$ be a formula. Then we have that $\mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q]) = \mathcal{R}ed(\mathcal{R}ed(\varphi)[\alpha \rightsquigarrow Q])$.

The proof follows immediately from Definition 4.5 and Definition 4.13. ■

Lemma 4.15 states that the result of the reduction of formulas with nested refinements is equal to the result of the refinement on certain formulas without nested refinements. It is the counterpart of Lemma 2.19.

Lemma 4.15
Let $Q_1, Q_2 \in R\Delta$ be process expressions, $\varphi \in R\mu\mathcal{L}^{App}$ be a formula and $* \in \{;, +\}$. If $\gamma_1, \gamma_2 \in \mathcal{A}$ such that $\gamma_1 \neq \gamma_2$ and $\gamma_1, \gamma_2 \notin \xi(\varphi) \cup \xi(red(Q_1 * Q_2))$ then we have that $\mathcal{R}ed(((\varphi[\alpha \rightsquigarrow (\gamma_1 * \gamma_2)])[\gamma_2 \rightsquigarrow Q_2])[\gamma_1 \rightsquigarrow Q_1]) = \mathcal{R}ed(\varphi[\alpha \rightsquigarrow (Q_1 * Q_2)])$.

Proof: Follows by Lemma 4.12 and Remark 4.14. ■

The following lemma states that the logical reduction function $\mathcal{R}ed$ is always defined.

Lemma 4.16
Let $\varphi \in R\mu\mathcal{L}^{App}$ be a formula. Then $\mathcal{R}ed(\varphi) \in \mu\mathcal{L}^{App}$.

Proof: The proof is by induction on the structure of φ using Lemma 4.10. ■

The next lemma shows that the set of performances of a formula $\varphi \in R\mu\mathcal{L}^{App}$ remains unchanged under the application of the reduction function.

Lemma 4.17

Let $\varphi \in R\mu\mathcal{L}^{App}$ be a formula. Then we have $\xi(\varphi) = \xi(\mathcal{R}ed(\varphi))$.

Proof: The proof is by induction on the structure of $\varphi \in R\mu\mathcal{L}^{App}$ using Lemma 2.18 and Lemma 4.11. ■

Definition 4.18

Let $\varphi, \psi_1, \ldots, \psi_n \in \mu\mathcal{L}^{App}$ and $Z_1, \ldots, Z_n \in Var$ be pairwise distinct variables. The $\mu\mathcal{L}^{App}$-formula $\varphi[\psi_1/Z_1]\ldots[\psi_n/Z_n]$ arises from φ by substituting each free occurrence of the variables $Z_1, \ldots, Z_n$ in φ simultaneously by the formulas $\psi_1, \ldots, \psi_n$. □

The next lemma shows that the application of substitution and reduction can be permuted in an appropriate way.

Lemma 4.19

Let $\varphi, \psi \in \mu\mathcal{L}^{App}$, $Q \in R\Delta$ and $Z \in Var$. Then

$$\mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])[\mathcal{R}ed(\psi[\alpha \rightsquigarrow Q])/Z] = \mathcal{R}ed((\varphi[\psi/Z])[\alpha \rightsquigarrow Q]).$$

Proof: Immediate. ■

We now define an "interpretation function" which maps $\mu\mathcal{L}$-formulas to *App*-formulas. This function will later be use to "transfer" results concerned with the logic *App* to the logic $\mu\mathcal{L}$.

Definition 4.20

Let $\omega_1 \in \mathcal{O}n$ be the first uncountable ordinal (that is, the least ordinal with cardinality $\aleph_1$). The interpretation function $\mathcal{I} : \mu\mathcal{L} \to App$ *is defined by*

$\mathcal{I}(*) := *$ *for* $* \in \{\bot, \top\} \cup Var$,

$\mathcal{I}((\varphi_1 \odot \varphi_2)) := (\mathcal{I}(\varphi_1) \odot \mathcal{I}(\varphi_2))$ *for* $\odot \in \{\wedge, \vee\}$,

$\mathcal{I}([\alpha]\varphi) := [\alpha]\mathcal{I}(\varphi)$, $\mathcal{I}(\langle\alpha\rangle\varphi) := \langle\alpha\rangle\mathcal{I}(\varphi)$, $\mathcal{I}(\sigma Z.\varphi) := \sigma^{\omega_1} Z.\mathcal{I}(\varphi)$ □

According to the last rule of the definition of $\mathcal{I}$, a fixed point formula $\varphi = \sigma Z.\varphi'$ is interpreted in *App* by its ω_1-fold approximation. As we will see later, this approximation constitutes a formula that is equivalent to φ.

The lemma below shows that the interpretation and the reduction of a $\mu\mathcal{L}$-formula can be permuted.

Lemma 4.21

Let $\varphi \in \mu\mathcal{L}$ be a formula and $Q \in \Delta$ be a process expression. Then we have that $\mathcal{R}ed(\mathcal{I}(\varphi)[\alpha \rightsquigarrow Q]) = \mathcal{I}(\mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q]))$.

Proof: First note that $\mathcal{I}(\mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])) = \mathcal{I}((\varphi)\{\alpha \rightsquigarrow Q\})$ since $\varphi \in \mu\mathcal{L}$ and $Q \in \Delta$. Similarly, $\mathcal{R}ed(\mathcal{I}(\varphi)[\alpha \rightsquigarrow Q]) = (\mathcal{I}(\varphi))\{\alpha \rightsquigarrow Q\}$ since $\mathcal{I}(\varphi) \in App$ and $Q \in \Delta$. It thus suffices to show $(\mathcal{I}(\varphi))\{\alpha \rightsquigarrow Q\} = \mathcal{I}((\varphi)\{\alpha \rightsquigarrow Q\})$. This is done by well-founded induction on the relation $\prec$ involving a case discrimination on the structure of $\varphi \in \mu\mathcal{L}$ and a subsidiary case discrimination on the structure of $Q \in \Delta$. We only show the case where $\varphi = [\alpha]\varphi'$ and $Q = (Q_1; Q_2)$: We have

$$
\begin{aligned}
&(\mathcal{I}([\alpha]\varphi'))\{\alpha \rightsquigarrow Q\}\\
&= ([\alpha]\mathcal{I}(\varphi'))\{\alpha \rightsquigarrow Q\}\\
&= \Big([\gamma]\Big([\delta](\mathcal{I}(\varphi'))\{\alpha \rightsquigarrow Q\}\Big)\{\delta \rightsquigarrow Q_2\}\Big)\{\gamma \rightsquigarrow Q_1\} \quad \text{(By Definition 4.5)}\\
&= \Big([\gamma]\Big([\delta]\mathcal{I}((\varphi')\{\alpha \rightsquigarrow Q\})\Big)\{\delta \rightsquigarrow Q_2\}\Big)\{\gamma \rightsquigarrow Q_1\} \quad \text{(Induction, } |\varphi'| < |\varphi|)\\
&= \Big([\gamma]\Big(\mathcal{I}([\delta](\varphi')\{\alpha \rightsquigarrow Q\})\Big)\{\delta \rightsquigarrow Q_2\}\Big)\{\gamma \rightsquigarrow Q_1\} \quad \text{(By Definition 4.20)}\\
&= \Big([\gamma]\mathcal{I}\Big(([\delta](\varphi')\{\alpha \rightsquigarrow Q\})\{\delta \rightsquigarrow Q_2\}\Big)\Big)\{\gamma \rightsquigarrow Q_1\} \quad \text{(Induction, } |Q_2| < |Q|)\\
&= \Big(\mathcal{I}\Big([\gamma]([\delta](\varphi')\{\alpha \rightsquigarrow Q\})\{\delta \rightsquigarrow Q_2\}\Big)\Big)\{\gamma \rightsquigarrow Q_1\} \quad \text{(By Definition 4.20)}\\
&= \mathcal{I}\Big(\Big([\gamma]([\delta](\varphi')\{\alpha \rightsquigarrow Q\})\{\delta \rightsquigarrow Q_2\}\Big)\{\gamma \rightsquigarrow Q_1\}\Big) \quad \text{(Induction, } |Q_1| < |Q|)\\
&= \mathcal{I}((\varphi)\{\alpha \rightsquigarrow Q\}) \quad \text{(By Definition 4.5)}
\end{aligned}
$$

■

We now extend the satisfaction relation of the Modal Mu-Calculus (see, for example, [119, 192]) in order to handle (logical) action refinement operators.

Definition 4.22 (Satisfaction of $R\mu\mathcal{L}^{App}$-formulas)

Let $P \in R\Sigma$, $Q \in R\Delta$, $\varphi, \psi \in R\mu\mathcal{L}^{App}$ and $Z \in Var$. Let $\vartheta : Var \to 2^{R\Sigma}$ be a valuation function[18]*.*

[18]The customary updating notation is used: $\vartheta[\mathcal{E}/Z]$ is the valuation ϑ' which agrees with ϑ on all variables $Z \in Var$ except Z, and $\vartheta'(Z) = \mathcal{E}$.

$P \models_\vartheta \top \quad , \qquad P \not\models_\vartheta \bot \quad , \qquad P \models_\vartheta Z$ *iff* $P \in \vartheta(Z)$

$P \models_\vartheta \bigwedge_{\kappa \in I} \varphi_\kappa$ *iff* $P \models_\vartheta \varphi_\kappa$ *for all* $\kappa \in I$

$P \models_\vartheta \bigvee_{\kappa \in I} \varphi_\kappa$ *iff* $P \models_\vartheta \varphi_\kappa$ *for some* $\kappa \in I$

$P \models_\vartheta [\alpha]\varphi$ *iff* $P \in \{E \in R\Sigma | \forall E' \in R\Sigma (E \xrightarrow{\alpha} E' \Rightarrow E' \models_\vartheta \varphi)\}$

$P \models_\vartheta \langle\alpha\rangle\varphi$ *iff* $P \in \{E \in R\Sigma | \exists E' \in R\Sigma (E \xrightarrow{\alpha} E'$ *and* $E' \models_\vartheta \varphi)\}$

$P \models_\vartheta \mu Z.\varphi$ *iff* $P \in \bigcap \left\{\mathcal{E} \subseteq R\Sigma | \{E \in R\Sigma | E \models_{\vartheta[\mathcal{E}/Z]} \varphi\} \subseteq \mathcal{E}\right\}$

$P \models_\vartheta \nu Z.\varphi$ *iff* $P \in \bigcup \left\{\mathcal{E} \subseteq R\Sigma | \mathcal{E} \subseteq \{E \in R\Sigma | E \models_{\vartheta[\mathcal{E}/Z]} \varphi\}\right\}$

$P \models_\vartheta \varphi[\alpha \rightsquigarrow Q]$ *iff* $P \models_\vartheta \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$

The semantics of approximation fixed point formulas is given by "syntactic unrolling" according to Figure 6 (see, for example, [119, 192]). □

$P \models_\vartheta \mu^0 Z.\varphi$ iff $P \models_\vartheta \bot$
$P \models_\vartheta \mu^{\kappa+1} Z.\varphi$ iff $P \models_\vartheta \varphi[\mu^\kappa Z.\varphi/Z]$
$P \models_\vartheta \mu^\kappa Z.\varphi$ iff $P \models_\vartheta \bigvee_{\lambda \in \kappa} \mu^\lambda Z.\varphi$ for any limit ordinal κ.

$P \models_\vartheta \nu^0 Z.\varphi$ iff $P \models_\vartheta \top$
$P \models_\vartheta \nu^{\kappa+1} Z.\varphi$ iff $P \models_\vartheta \varphi[\nu^\kappa Z.\varphi/Z]$
$P \models_\vartheta \nu^\kappa Z.\varphi$ iff $P \models_\vartheta \bigwedge_{\lambda \in \kappa} \nu^\lambda Z.\varphi$ for any limit ordinal κ.

Figure 6: Semantics of Approximation Fixed Point Formulas

It is now possible to capture satisfaction of a fixed point formula by means of approximation fixed point formulas.

Remark 4.23 (See, for example, [119, 192])

1) $P \models_\vartheta \mu Z.\varphi$ *iff* $P \models_\vartheta \mu^\kappa Z.\varphi$ *for some* $\kappa \in \mathcal{O}n$

2) $P \models_\vartheta \nu Z.\varphi$ *iff* $P \models_\vartheta \nu^\kappa Z.\varphi$ *for all* $\kappa \in \mathcal{O}n$ □

We say P *satisfies* φ (with respect to ϑ) iff $P \models_\vartheta \varphi$. For a closed $R\mu\mathcal{L}^{App}$-formula φ we simply write $P \models \varphi$. Minimum- and maximum fixed points always exist by the results of [197].

Example 4.24
Let $\phi = \nu Z.([a]\bot \wedge [b]Z)$ *and* $\psi = \mu Z.(\langle a\rangle\top \vee \langle b\rangle Z)$. *Then* ϕ *intuitively expresses the safety property 'there is no a-action executable on any b-path' and* ψ *expresses the liveness property 'there exists a b-path after which the action a can eventually be executed'. (see also [192]).* □

Example 4.25
Let $P_1 = fix(x = ((\alpha \|_\emptyset \beta); x))$, $P_2 = fix(y = (((\alpha;\beta)+(\beta;\alpha)); y))$, $\varphi = \nu Z.(\langle\alpha\rangle\langle\beta\rangle Z \wedge \langle\beta\rangle\langle\alpha\rangle Z)$, $Q := \gamma[\gamma \rightsquigarrow (\alpha_1;\alpha_2)]$. $P_i \models \varphi$ *and* $P_i[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q]$ *for* $i = 1, 2$ *(to see this note that* $\varphi[\alpha \rightsquigarrow Q]$ *is logically equivalent to the formula* $\nu Z.(\langle\alpha_1\rangle\langle\alpha_2\rangle\langle\beta\rangle Z \wedge \langle\beta\rangle\langle\alpha_1\rangle\langle\alpha_2\rangle Z)$*). In addition* $P_1[\alpha \rightsquigarrow Q]$ *satisfies* $\langle\alpha_1\rangle\langle\beta\rangle\langle\alpha_1\rangle\top$ *which is not satisfied by* $P_2[\alpha \rightsquigarrow Q]$. □

4.1.1 Discussion

We decided to use the Modal Mu-Calculus as our logical framework for the following reasons. The modalities of $\mu\mathcal{L}$ intuitively capture the idea of "computation steps". Furthermore, the two different modalities allow a clean distinction between existential and universal properties [21, 16].

We have seen that the two basic temporal attributes of reactive systems, referred to as *safety*- and *liveness* properties [125, 6] are captured by maximum- and minimum fixed point operators respectively [59]. Nested fixed point operators can complicate an intuitive understanding of formulas. Generally, formulas with a fixed number $n \in \mathbb{N}$ of fixed point operators cannot capture particular properties that can be captured by formulas with $n + 1$ fixed point operators [27]. It has been argued however, that most of the properties interesting for practical purposes can be expressed by formulas with two fixed point operators [72]. The ability to formalize important attributes of reactive systems like, for example, safety-, liveness-, fairness- or cyclic properties in all combinations [192, 26] reinforces the pertinence of $\mu\mathcal{L}$ to reason about $RCSs$-designs. Further, $LTSs$ are generic structures for the interpretation of the logic $\mu\mathcal{L}$ as they are closely related to "Kripke structures" (see, for example, [121]), the original mathematical structure over which modal logics have been interpreted. Moreover, $\mu\mathcal{L}$ logically characterizes strong bisimulation equivalence (see, for example, [192]), the most thoroughly investigated equivalence relation for $LTSs$. Hence, $LTSs$ and $\mu\mathcal{L}$ fit together also from the theoretical point of view.

4.2 Simultaneous Syntactic Action Refinement (SSAR) for the Process Algebra $R\Sigma$ and the Modal Mu-Calculus

In this section we provide the link between SAR in the process algebra $R\Sigma$ and SAR in the logic $R\mu\mathcal{L}$.

Definition 4.26
For a fixed point formula $\varphi = \sigma Z.\varphi' \in \mu\mathcal{L}^{App}$, the closure ordinal *of φ (relative to a valuation function ϑ) is the least ordinal $\lambda \in \mathcal{O}n$ such that*

$$\{P \in R\Sigma \mid P \models_\vartheta \sigma^\lambda Z.\varphi'\} = \{P \in R\Sigma \mid P \models_\vartheta \sigma^{\lambda+1} Z.\varphi'\}.$$

Let $cl_\vartheta(\varphi)$ denote the closure ordinal of φ (relative to ϑ). If φ is closed we simply write $cl(\varphi)$. □

For $\varphi \in \mu\mathcal{L}^{App}$, let $\|\varphi\|_\vartheta$ denote the set $\{P \in R\Sigma \mid P \models_\vartheta \varphi\}$.

Lemma 4.27
Let ϑ be a valuation function and let $\varphi = \sigma Z.\varphi' \in \mu\mathcal{L}^{App}$. Then

- $\|\varphi\|_\vartheta = \|\sigma^{cl_\vartheta(\varphi)} Z.\varphi'\|_\vartheta$,
- $\|\sigma^{cl_\vartheta(\varphi)} Z.\varphi'\|_\vartheta = \|\sigma^\lambda Z.\varphi'\|_\vartheta$ *for each* $\lambda \geq cl_\vartheta(\varphi)$,
- $cl_\vartheta(\varphi) \leq \omega_1$. □

Proof: The second assertion follows directly from Definition 4.26. The third assertion is a consequence of the facts that $cl_\vartheta(\varphi) \leq |R\Sigma|$ for any valuation function ϑ (see, e.g., [26, p.20] or [190, p. 530]) and that $R\Sigma$ is a countable set of processes. The first assertion is a consequence of the fact that any formula $\varphi \in \mu\mathcal{L}^{App}$ determines a monotonic function $\mathcal{E} \mapsto \|\varphi\|_{\vartheta[\mathcal{E}/Z]}$, $\mathcal{E} \subseteq R\Sigma$. ■

The lemma above can be used in order construct for each Modal Mu-Calculus formula $\varphi \in \mu\mathcal{L}$ an approximative Modal Mu-Calculus formula $\psi \in App$ such that $\|\varphi\|_\vartheta = \|\psi\|_\vartheta$. These constructions can be achieved via the interpretation function of Definition 4.20.

Lemma 4.28

Let $\varphi \in \mu\mathcal{L}$. Then $\|\varphi\|_\vartheta = \|\mathcal{I}(\varphi)\|_\vartheta$.

Proof: The proof is by induction on the structure of $\varphi \in \mu\mathcal{L}$. The interesting cases are where φ is a fixed point formula. We only show the case $\varphi = \mu Z.\varphi'$: We have

$$P \models_\vartheta \mu Z.\varphi'$$

$$\text{iff } P \in \bigcap \left\{\mathcal{E} \subseteq R\Sigma | \{E \in R\Sigma | E \models_{\vartheta[\mathcal{E}/Z]} \varphi'\} \subseteq \mathcal{E}\right\}$$

$$\text{iff } P \in \bigcap \left\{\mathcal{E} \subseteq R\Sigma | \{E \in R\Sigma | E \models_{\vartheta[\mathcal{E}/Z]} \mathcal{I}(\varphi')\} \subseteq \mathcal{E}\right\} \quad \text{(By induction)}$$

$$\text{iff } P \models_\vartheta \mu Z.\mathcal{I}(\varphi')$$

$$\text{iff } P \models_\vartheta \mu^{\omega_1} Z.\mathcal{I}(\varphi') \quad \text{(By Lemma 4.27)}$$

$$\text{iff } P \models_\vartheta \mathcal{I}(\mu Z.\varphi').$$

■

Remark 4.29

Consider Definition 4.20: We used the first uncountable ordinal ω_1 to fomulate the transformation $\mathcal{I}$. If we considered only finite state transition systems, we could replace ω_1 with the first transfinite ordinal ω_0. For finite state systems, Lemma 4.28 would then still be valid. □

Definition 4.30

The depth *of a formula $\varphi \in \mu\mathcal{L}^{App}$ (relative to a valuation function ϑ) is given by the function $d_\vartheta : \mu\mathcal{L}^{App} \to \mathcal{O}n$ defined as follows:*

$d_\vartheta(*) := 0$ *where* $* \in \{\top, \bot\} \cup Var$

$d_\vartheta(\bigotimes \varphi_\kappa) := \sup_{\kappa \in I}(d_\vartheta(\varphi_\kappa)) + 1$ *where* $\bigotimes \in \{\bigvee_{\kappa \in I}, \bigwedge_{\kappa \in I}\}$

$d_\vartheta(\triangle_\alpha \varphi) := d_\vartheta(\varphi) + 1$ *where* $\triangle_\alpha \in \{\langle\alpha\rangle, [\alpha]\}$

$d_\vartheta(\sigma^0 Z.\varphi) := 0$

$d_\vartheta(\sigma^{\lambda+1} Z.\varphi) := d_\vartheta(\varphi[\sigma^\lambda Z.\varphi/Z])$

$d_\vartheta(\sigma^\lambda Z.\varphi) := \sup_{\kappa\in\lambda}(d_\vartheta(\sigma^\kappa Z.\varphi)) + 1$ *where* λ *is a limit ordinal.*

$d_\vartheta(\sigma Z.\varphi) := d_\vartheta(\sigma^\lambda Z.\varphi) + 1$ *where* $\lambda = cl_\vartheta(\sigma Z.\varphi)$.

For closed formulas φ *we simply write* $d(\varphi)$. □

Example 4.31
Let $\varphi = \nu^2 Z.([\alpha]\bot \wedge \langle\beta\rangle Z)$. *Then*

$$d(\varphi) = d\Big(([\alpha]\bot \wedge \langle\beta\rangle([\alpha]\bot \wedge \langle\beta\rangle(\nu^0 Z.([\alpha]\bot \wedge \langle\beta\rangle Z))))\Big) = 4.$$

□

The transitive and non-reflexive relation $\prec_{d_\vartheta} \subseteq (\Delta\times\mu\mathcal{L}^{App})^2$ is defined by $(Q_1,\psi) \prec_{d_\vartheta} (Q_2,\varphi)$ iff $|Q_1| < |Q_2|$ or ($|Q_1| = |Q_2|$ and $d_\vartheta(\psi) < d_\vartheta(\varphi)$).

Lemma 4.32
Let $\varphi = \sigma^{\lambda_1+1}Z_1.\sigma^{\lambda_2+1}Z_2.\ldots.\sigma^{\lambda_n+1}Z_n.\psi$ $(n \geq 1)$ *be a* $CG\mu\mathcal{L}^{App}$*-formula. Consider the formula* $\phi = \psi[\phi_1/Z_1]\ldots[\phi_n/Z_n]$ *where, for* $1 \leq i < n$*, the formulas* ϕ_i *are defined by*

$$\phi_1 := \sigma^{\lambda_1}Z_1.\sigma^{\lambda_2+1}Z_2.\ldots.\sigma^{\lambda_n+1}Z_n.\psi,$$

$$\phi_{i+1} := \sigma^{\lambda_{i+1}}Z_{i+1}.\sigma^{\lambda_{i+2}+1}.\ldots.\sigma^{\lambda_n+1}Z_n.\psi[\phi_1/Z_1]\ldots[\phi_i/Z_i].$$

Then for all $P \in R\Sigma$ *we have that*

1) $P \models \varphi$ *iff* $P \models \phi$,

2) For any $Q \in \Delta$ *we have that* $P \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$ *iff* $P \models \mathcal{R}ed(\phi[\alpha \rightsquigarrow Q])$,

3) $d(\varphi) = d(\phi)$.

Proof: 1) Immediate, since $P \models \sigma^{\lambda+1}Z.\varphi'$ iff $P \models \varphi'[\sigma^{\lambda}Z.\varphi'/Z]$.

2) We can derive that $\mathcal{R}ed((\psi[\phi_1/Z_1]\ldots[\phi_n/Z_n])[\alpha \rightsquigarrow Q])$ equals

$$\mathcal{R}ed(\psi[\alpha \rightsquigarrow Q])[\mathcal{R}ed(\phi_1[\alpha \rightsquigarrow Q])/Z_1]\ldots[\mathcal{R}ed(\phi_n[\alpha \rightsquigarrow Q])/Z_n]$$

by n successive applications of Lemma 4.19. In a similar way, we infer that the formula $\mathcal{R}ed(\phi_1[\alpha \rightsquigarrow Q])$ equals

$$\sigma^{\lambda_1}Z_1.\sigma^{\lambda_2+1}Z_2.\ldots.\sigma^{\lambda_n+1}Z_n.\mathcal{R}ed(\psi[\alpha \rightsquigarrow Q]).$$

For $1 \leq m \leq n$, let $[\mathcal{R}ed(\phi[\alpha \rightsquigarrow Q])/Z]^{\leq m}$ abbreviate the sequence of substitutions $[\mathcal{R}ed(\phi_1[\alpha \rightsquigarrow Q])/Z_1]\ldots[\mathcal{R}ed(\phi_m[\alpha \rightsquigarrow Q])/Z_m]$. For $1 \leq i < n$, the formula $\mathcal{R}ed(\phi_{i+1}[\alpha \rightsquigarrow Q])$ equals

$$\sigma^{\lambda_{i+1}}Z_i.\sigma^{\lambda_{i+2}+1}Z_{i+2}.\ldots.\sigma^{\lambda_n+1}Z_n.\mathcal{R}ed((\psi[\phi_1/Z_1]\ldots[\phi_i/Z_i])[\alpha \rightsquigarrow Q])$$

$$= \sigma^{\lambda_{i+1}}Z_i.\sigma^{\lambda_{i+2}+1}Z_{i+2}.\ldots.\sigma^{\lambda_n+1}Z_n.\mathcal{R}ed(\psi[\alpha \rightsquigarrow Q])[\mathcal{R}ed(\phi[\alpha \rightsquigarrow Q])/Z]^{\leq i}$$

which follows by i successive applications of Lemma 4.19. Since $P \models \sigma^{\lambda+1}Z.\varphi'$ iff $P \models \varphi'[\sigma^{\lambda}Z.\varphi'/Z]$ we obtain the following sequence of equivalence assertions:

$$P \models \mathcal{R}ed(\psi[\alpha \rightsquigarrow Q])[\mathcal{R}ed(\phi[\alpha \rightsquigarrow Q])/Z]^{\leq n}$$

$$\Leftrightarrow P \models \sigma^{\lambda_n+1}Z_n.\mathcal{R}ed(\psi[\alpha \rightsquigarrow Q])[\mathcal{R}ed(\phi[\alpha \rightsquigarrow Q])/Z]^{\leq n-1}$$

$$\begin{array}{lll} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{array}$$

$$\Leftrightarrow P \models \sigma^{\lambda_2+1}Z_2.\sigma^{\lambda_3+1}Z_3.\ldots.\sigma^{\lambda_n+1}Z_n.\mathcal{R}ed(\psi[\alpha \rightsquigarrow Q])[\mathcal{R}ed(\phi[\alpha \rightsquigarrow Q])/Z]^{\leq 1}$$

$$\Leftrightarrow P \models \sigma^{\lambda_1+1}Z_1.\sigma^{\lambda_2+1}Z_2.\sigma^{\lambda_3+1}Z_3.\ldots.\sigma^{\lambda_n+1}Z_n.\mathcal{R}ed(\psi[\alpha \rightsquigarrow Q])$$

$$\Leftrightarrow P \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$$

3) By definition we have $d(\sigma^{\lambda+1}Z.\varphi') = d(\varphi'[\sigma^{\lambda}Z.\varphi'/Z])$. ■

The following two "expansion lemmata" formalize the possibility to refine performances of process expressions and formulas by simple process expressions, composed of two performances, without affecting the satisfaction relation. In the proofs of the following two lemmata and Theorem 4.38, the guardedness and closedness conditions will allow to reduce the induction steps for (approximation) fixed point formulas to previous induction steps.

Lemma 4.33 (Expansion lemma for ';')

Let $P \in G\Sigma$ and let $\varphi \in CGApp$ be a formula and $\gamma_1, \gamma_2 \in \mathcal{A}$ such that $\gamma_1 \neq \gamma_2$ and $\gamma_1, \gamma_2 \notin alph(P) \cup \xi(\varphi)$. Then

$$P \models \varphi \Leftrightarrow red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]).$$

Proof: The proof is by transfinite induction on the depth $d(\varphi)$ of $\varphi \in CGApp$. Only the cases where φ is of the form $\langle\alpha\rangle\varphi$ or $[\alpha]\varphi'$ differ conceptually from the proof of Theorem 4.36. We here only show the case where $\varphi = [\alpha]\varphi'$.

$\underline{\varphi = [\beta]\varphi' \text{ where } \alpha \neq \beta:}$

Both directions are proved by an indirect argument.

'⇒':

Assume $P \models [\beta]\varphi'$ and $red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \not\models \mathcal{R}ed(([\beta]\varphi')[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$
From

$$red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \not\models \mathcal{R}ed(([\beta]\varphi')[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$$

we obtain

$$red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \not\models [\beta]\mathcal{R}ed(\varphi'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$$

by Definition 4.5 and Definition 4.13.
Hence we get

$$\exists E' \in R\Sigma\Big(red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\beta} E' \text{ and } E' \not\models \mathcal{R}ed(\varphi'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\Big) \quad (1)$$

by Definition 4.22. Now $\alpha \neq \beta$, $\beta \notin \xi((\gamma_1;\gamma_2))$ due to the condition $\gamma_1, \gamma_2 \notin \xi(\varphi)$ and since $\beta \in \xi(\varphi)$. We obtain

$$\exists P''(P \xrightarrow{\beta} P'' \text{ and } red(P''[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = E') \quad (2)$$

by application of assertion 2) from Lemma 2.28. Now (1) and (2) give

$$red(P''[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \not\models \mathcal{R}ed(\varphi'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$$

whence we get

$$\exists P''(P \xrightarrow{\beta} P'' \text{ and } P'' \not\models \varphi')$$

by the induction hypothesis (note that $d(\varphi') < d(\varphi)$), yielding the contradiction

$$P \not\models [\beta]\varphi'.$$

'$\Leftarrow$':

Assume $red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \models \mathcal{R}ed(([\beta]\varphi')[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$ and $P \not\models [\beta]\varphi'$, that is,

$$\exists E' \in R\Sigma(P \xrightarrow{\beta} E' \text{ and } E' \not\models \varphi')$$

Since $\alpha \neq \beta$ we obtain

$$\exists E' \in R\Sigma(red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\beta} red(E'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \text{ and } E' \not\models \varphi')$$

by application of assertion 1) from Lemma 2.28. But this implies

$$\exists E' \in R\Sigma\Big(red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\beta} red(E'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \text{ and }$$

$$red(E'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \not\models \mathcal{R}ed(\varphi'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\Big)$$

by the induction hypothesis (again we have $d(\varphi') < d(\varphi)$). Hence we get

$$red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \not\models [\beta]\mathcal{R}ed(\varphi'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$$

whence the desired contradiction follows.

$\underline{\varphi = [\alpha]\varphi':}$

First, we observe that

$$\mathcal{R}ed(([\alpha]\varphi')[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = [\gamma_1][\gamma_2]\mathcal{R}ed(\varphi'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$$

follows by Definition 4.5 and Definition 4.13.

We proceed as follows.

'$\Rightarrow$':

Assume $P \models [\alpha]\varphi'$ and $red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \not\models \mathcal{R}ed(([\alpha]\varphi')[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$

We have:

$red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \not\models \mathcal{R}ed(([\alpha]\varphi')[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$

$\Leftrightarrow red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \not\models [\gamma_1][\gamma_2]\mathcal{R}ed(\varphi'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$
(By the claim)

$\Leftrightarrow \exists P', P'' \in R\Sigma\Big(red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P'$ and $P' \xrightarrow{\gamma_2} P''$ and
$P'' \not\models \mathcal{R}ed(\varphi'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\Big)$ (1) Since $\gamma_1, \gamma_2 \notin \xi(P)$ we can apply assertion 2) of Lemma 2.31 and obtain

$$\exists \tilde{P}(P \xrightarrow{\alpha} \tilde{P} \text{ and } red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) = P'') \quad (2)$$

Taking (1) and (2) together we have $red(\tilde{P}[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \not\models \mathcal{R}ed(\varphi'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$. By the induction hypothesis we obtain

$$\tilde{P} \not\models \varphi' \quad (3)$$

But (2) and (3) imply

$$P \not\models [\alpha]\varphi'.$$

'$\Leftarrow$':

Assume $red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \models \mathcal{R}ed(([\alpha]\varphi')[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])$ and $P \not\models [\alpha]\varphi'$. From the latter we obtain

$$\exists P' \in R\Sigma(P \xrightarrow{\alpha} P' \text{ and } P' \not\models \varphi').$$

By assertion 1) of Lemma 2.31 we get

$$\exists P''(red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} P'' \xrightarrow{\gamma_2} red(P'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])) \quad (1)$$

Further $P' \not\models \varphi'$ implies

$$red(P'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \not\models \mathcal{R}ed(\varphi'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \quad (2)$$

by the induction hypothesis. By (1) and (2)

$$\exists E', E'' \in R\Sigma\Big(red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \xrightarrow{\gamma_1} E' \xrightarrow{\gamma_2} E'' \text{ and}$$

$$E'' \not\models \mathcal{R}ed(\varphi'[\alpha \rightsquigarrow (\gamma_1;\gamma_2)])\Big)$$

and therefore

$$red(P[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]) \not\models \mathcal{R}ed(([\alpha]\varphi')[\alpha \rightsquigarrow (\gamma_1;\gamma_2)]).$$

■

Lemma 4.34 (Expansion lemma for '+')

Let $P \in G\Sigma$. Let $\varphi \in CGApp$ be a formula and $\gamma_1, \gamma_2 \in \mathcal{A}$ be such that $\gamma_1 \neq \gamma_2$ and $\gamma_1, \gamma_2 \notin alph(P) \cup \xi(\varphi)$. Then

$$P \models \varphi \Leftrightarrow red(P[\alpha \rightsquigarrow (\gamma_1 + \gamma_2)]) \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow (\gamma_1 + \gamma_2)]).$$

Proof: In analogy to the proof of the proceeding lemma using Lemma 2.32. ■

In Theorem 4.36 we can meet the conditions that Q is distinct, P is alphabet-disjoint to Q and that φ is ξ-disjoint to Q by renaming the performances of Q in the obvious way. This renaming is consistent with the usual approach to action refinement since a performance α which is to be refined in the term $P[\alpha \rightsquigarrow Q]$ is the abstraction of the term Q whence it should not be considered equal to any performance which occurs in Q itself thereby supporting the separation of different levels of abstraction [88]. Disjoint sets of performances are necessary as can be seen in the following.

Example 4.35
Consider the process expression $P := (a\|_{\{b\}}a)$ and the formula $\varphi := \langle a\rangle\langle a\rangle\top$. We have $P \models \varphi$ but $red(P[a \rightsquigarrow b]) \not\models \mathcal{R}ed(\varphi[a \rightsquigarrow b])$. Note that the process expression P is not $\chi\xi$-disjoint from the process term Q, that is, we have $\chi(P) \cap \xi(Q) \neq \emptyset$. □

Theorem 4.36
Let $P \in G\Sigma$ be a process term and $\varphi \in CGApp$ be a formula. Further let $Q \in \Delta$ be a distinct process term, such that P is alphabet-disjoint from Q and φ is ξ-disjoint from Q. Then $P \models \varphi \Leftrightarrow red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$.

Proof: Induction on the relation $\prec_d$ involving a case discrimination of the structure of $\varphi \in CGApp$ and a subsidiary case discrimination of the structure of $Q \in \Delta$.

First note that the case $\varphi = Z$ cannot occur since Z is neither a guarded nor a closed formula.

Induction Base:

$\underline{\varphi = *, \text{ where } * \in \{\top, \bot\}}$: Trivial.

Induction Hypothesis:

$\forall \alpha \in \mathcal{A}\ \forall P \in G\Sigma\ \forall \tilde{\varphi} \in CGApp\ \forall \tilde{Q} \in R\Delta$ such that $\tilde{Q}$ is distinct, P and $\tilde{\varphi}$ are alphabet-disjoint from $\tilde{Q}$ and $(\tilde{Q}, \tilde{\varphi}) \prec_d (Q, \varphi)$ we have

$$P \models \tilde{\varphi} \Leftrightarrow red(P[\alpha \rightsquigarrow \tilde{Q}]) \models \mathcal{R}ed(\tilde{\varphi}[\alpha \rightsquigarrow \tilde{Q}])$$

Induction Step:

$\underline{\varphi = \bigvee_{\lambda \in I} \varphi_\lambda}$:

We have $P \models \varphi$ iff $P \models \bigvee_{\lambda \in I} \varphi_\lambda$

$$\text{iff } P \models \varphi_\lambda \text{ for some } \lambda \in I$$

(by Definition 4.22)

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\varphi_\lambda[\alpha \rightsquigarrow Q]) \text{ for some } \lambda \in I$$

(By the induction hypothesis. Note that the induction hypothesis is applicable since $d(\varphi_\lambda) \leq \sup_{\lambda \in I}(d(\varphi_\lambda)) < \sup_{\lambda \in I}(d(\varphi_\lambda)) + 1 = d(\varphi)$.

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \bigvee_{\lambda \in I} \mathcal{R}ed(\varphi_\lambda[\alpha \rightsquigarrow Q])$$

(By Definition 4.22)

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \bigvee_{\lambda \in I} \Big(\mathcal{R}ed(\varphi_\lambda)\Big)\{\alpha \rightsquigarrow red(Q)\}$$

(By Definition 4.13)

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \Big(\bigvee_{\lambda \in I} \mathcal{R}ed(\varphi_\lambda)\Big)\{\alpha \rightsquigarrow red(Q)\}$$

(By Definition 4.5)

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \Big(\mathcal{R}ed(\bigvee_{\lambda \in I} \varphi_\lambda)\Big)\{\alpha \rightsquigarrow red(Q)\}$$

(By Definition 4.13)

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed((\bigvee_{\lambda \in I} \varphi_\lambda))[\alpha \rightsquigarrow Q])$$

(By Definition 4.13)

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$$

$\underline{\varphi = \bigwedge_{\lambda \in I} \varphi_\lambda}$: Similarly to the above case.

$\underline{\varphi = [\beta]\varphi' \text{ where } \alpha \neq \beta}$:

Both directions are proved by an indirect argument.

'$\Rightarrow$':

In this case we exploit the condition that the formula φ is ξ-disjoint from the process term Q, that is, $\xi(\varphi) \cap \xi(Q) = \emptyset$.

Assume $P \models [\beta]\varphi'$ and $red(P[\alpha \rightsquigarrow Q]) \not\models \mathcal{R}ed(([\beta]\varphi')[\alpha \rightsquigarrow Q])$
From

$$red(P[\alpha \rightsquigarrow Q]) \not\models \mathcal{R}ed(([\beta]\varphi')[\alpha \rightsquigarrow Q])$$

we obtain

$$red(P[\alpha \rightsquigarrow Q]) \not\models [\beta]\mathcal{R}ed(\varphi'[\alpha \rightsquigarrow Q]).$$

Hence

$$\exists E' \in R\Sigma\Big(red(P[\alpha \leadsto Q]) \xrightarrow{\beta} E' \text{ and}$$

$$E' \not\models \mathcal{R}ed(\varphi'[\alpha \leadsto Q])\Big) \quad (1)$$

by Definition 4.22. Now $\beta \notin \xi(Q)$ due to ξ-disjointness of φ form Q and since $\beta \in \xi(\varphi)$. Hence

$$\exists P''(P \xrightarrow{\beta} P'' \text{ and } red(P''[\alpha \leadsto Q]) = E') \quad (2)$$

by application of assertion 2) from Lemma 2.28. Now (1) and (2) give

$$red(P''[\alpha \leadsto Q]) \not\models \mathcal{R}ed(\varphi'[\alpha \leadsto Q])$$

whence we obtain

$$\exists P''(P \xrightarrow{\beta} P'' \text{ and } P'' \not\models \varphi')$$

by the induction hypothesis: First note that P'' and φ' are alphabet-disjoint from Q. Further, we have that $d(\varphi') < d(\varphi)$. It follows

$$P \not\models [\beta]\varphi'.$$

'$\Leftarrow$':

Assume $red(P[\alpha \leadsto Q]) \models \mathcal{R}ed(([\beta]\varphi')[\alpha \leadsto Q])$ and $P \not\models [\beta]\varphi'$. From $P \not\models [\beta]\varphi'$ we obtain

$$\exists E' \in R\Sigma(P \xrightarrow{\beta} E' \text{ and } E' \not\models \varphi').$$

Since $\alpha \neq \beta$ by the current induction step and $\beta \notin \xi(Q)$ by the condition of of ξ-disjointness we obtain

$$\exists E' \in R\Sigma(red(P[\alpha \leadsto Q]) \xrightarrow{\beta} red(E'[\alpha \leadsto Q]) \text{ and } E' \not\models \varphi')$$

by application of assertion 1) from Lemma 2.28. But this implies

$$\exists E' \in R\Sigma\Big(red(P[\alpha \leadsto Q]) \xrightarrow{\beta} red(E'[\alpha \leadsto Q]) \text{ and}$$

$$red(E'[\alpha \leadsto Q]) \not\models \mathcal{R}ed(\varphi'[\alpha \leadsto Q])\Big)$$

by the induction hypothesis. Hence

$$red(P[\alpha \rightsquigarrow Q]) \not\models [\beta]\mathcal{R}ed(\varphi'[\alpha \rightsquigarrow Q])$$

that is,

$$red(P[\alpha \rightsquigarrow Q]) \not\models \mathcal{R}ed(([\beta]\varphi')[\alpha \rightsquigarrow Q]).$$

$\underline{\varphi = [\alpha]\varphi'}$: We do the proof by a case discrimination on the structure of $Q \in \Delta$.

a) $Q = \beta$:

Both directions are proved by means of an indirect argument.

'⇒':

The condition of alphabet-disjointness of P from Q implies the condition of ξ-disjointness of P from Q, that is, $\xi(P) \cap \xi(Q) = \emptyset$. The latter condition is necessary to carry out this induction step.

Assume $P \models [\alpha]\varphi'$ and $red(P[\alpha \rightsquigarrow \beta]) \not\models \mathcal{R}ed(([\alpha]\varphi')[\alpha \rightsquigarrow \beta])$
From

$$red(P[\alpha \rightsquigarrow \beta]) \not\models \mathcal{R}ed(([\alpha]\varphi')[\alpha \rightsquigarrow \beta])$$

we obtain

$$red(P[\alpha \rightsquigarrow \beta]) \not\models [\beta]\mathcal{R}ed(\varphi'[\alpha \rightsquigarrow \beta]).$$

Hence

$$\exists E' \in R\Sigma\Big(red(P[\alpha \rightsquigarrow \beta]) \xrightarrow{\beta} E' \text{ and}$$
$$E' \not\models \mathcal{R}ed(\varphi'[\alpha \rightsquigarrow \beta])\Big) \quad (1)$$

Now $\beta \in \xi(Q)$ implies $\beta \notin \xi(P)$ whence

$$\exists P''(P \xrightarrow{\alpha} P'' \text{ and } red(P''[\alpha \rightsquigarrow \beta]) = E') \quad (2)$$

by application of assertion 2) from Lemma 2.30. Now (1) and (2) give

$$red(P''[\alpha \rightsquigarrow \beta]) \not\models \mathcal{R}ed(\varphi'[\alpha \rightsquigarrow \beta])$$

whence

$$\exists P''(P \overset{\alpha}{\to} P'' \text{ and } P'' \not\models \varphi')$$

by the induction hypothesis, that is,

$$P \not\models [\alpha]\varphi'.$$

'⇐':

Assume $\mathit{red}(P[\alpha \rightsquigarrow \beta]) \models \mathcal{R}\mathit{ed}(([\alpha]\varphi')[\alpha \rightsquigarrow Q])$ and $P \not\models [\alpha]\varphi'$. From $P \not\models [\alpha]\varphi'$ we obtain

$$\exists E' \in R\Sigma(P \overset{\alpha}{\to} E' \text{ and } E' \not\models \varphi').$$

Hence

$$\exists E' \in R\Sigma(\mathit{red}(P[\alpha \rightsquigarrow \beta]) \overset{\beta}{\to} \mathit{red}(E'[\alpha \rightsquigarrow \beta]) \text{ and } E' \not\models \varphi')$$

by application of assertion 1) from Lemma 2.30. But this implies

$$\exists E' \in R\Sigma\Big(\mathit{red}(P[\alpha \rightsquigarrow \beta]) \overset{\beta}{\to} \mathit{red}(E'[\alpha \rightsquigarrow \beta]) \text{ and}$$

$$\mathit{red}(E'[\alpha \rightsquigarrow \beta]) \not\models \mathcal{R}\mathit{ed}(\varphi'[\alpha \rightsquigarrow \beta])\Big)$$

by the induction hypothesis. Hence

$$\mathit{red}(P[\alpha \rightsquigarrow \beta]) \not\models [\beta]\mathcal{R}\mathit{ed}(\varphi'[\alpha \rightsquigarrow \beta])$$

that is,

$$\mathit{red}(P[\alpha \rightsquigarrow \beta]) \not\models \mathcal{R}\mathit{ed}(([\alpha]\varphi')[\alpha \rightsquigarrow \beta])$$

Let $\gamma, \delta \in Var_{Act}$ be such that $\gamma, \delta \notin \mathit{alph}(P) \cup \xi(Q) \cup \xi(\varphi)$ and $\gamma \neq \delta$.

b) $Q = (Q_1 + Q_2)$:

Let $\hat{P} := \mathit{red}(P[\alpha \rightsquigarrow (\gamma + \delta)]$ and $\hat{\varphi} := \mathcal{R}\mathit{ed}(\varphi[\alpha \rightsquigarrow (\gamma + \delta)])$.

We have $P \models \varphi$

$\Leftrightarrow$ $\hat{P} \models \hat{\varphi}$
(By Lemma 4.34)

$\Leftrightarrow$ $red((red(\hat{P}[\delta \rightsquigarrow Q_2]))[\gamma \rightsquigarrow Q_1]) \models$
$\mathcal{R}ed((\mathcal{R}ed(\hat{\varphi}[\delta \rightsquigarrow Q_2]))[\gamma \rightsquigarrow Q_1])$
(By two applications of the induction hypothesis. Note that
$(Q_i, \psi) \prec_d (Q, \psi')$, $i \in \{1, 2\}$, for any $\psi, \psi' \in CG\mu\mathcal{L}^{App}$.
Since Q is distinct, $red(\hat{P}[\delta \rightsquigarrow Q_2])$ is alphabet-disjoint from Q_1
and $\mathcal{R}ed(\hat{\varphi}[\delta \rightsquigarrow Q_2])$ is ξ-disjoint from Q_1)

$red(((P[\alpha \rightsquigarrow (\gamma + \delta])[\delta \rightsquigarrow Q_2])[\gamma \rightsquigarrow Q_1]) \models$
$\mathcal{R}ed(((\varphi[\alpha \rightsquigarrow (\gamma + \delta)])[\delta \rightsquigarrow Q_2])[\gamma \rightsquigarrow Q_1])$
(By Remark 2.17 and Remark 4.14)

$\Leftrightarrow$ $red(P[\alpha \rightsquigarrow (Q_1 + Q_2)]) \models \mathcal{R}ed(([\alpha]\varphi')[\alpha \rightsquigarrow (Q_1 + Q_2)])$
(By Lemma 2.19 and Lemma 4.15)

c) $Q = (Q_1; Q_2)$:

Let $\hat{P} := red(P[\alpha \rightsquigarrow (\gamma; \delta)]$ and $\hat{\varphi} := \mathcal{R}ed(\varphi[\alpha \rightsquigarrow (\gamma; \delta)])$.

We have $P \models \varphi$

$\Leftrightarrow$ $\hat{P} \models \hat{\varphi}$
(By Lemma 4.33)

$\Leftrightarrow$ $red(red(\hat{P}[\delta \rightsquigarrow Q_2])[\gamma \rightsquigarrow Q_1]) \models \mathcal{R}ed(\mathcal{R}ed(\hat{\varphi}[\delta \rightsquigarrow Q_2])[\gamma \rightsquigarrow Q_1])$
(By two applications of the induction hypothesis)

$\Leftrightarrow$ $red(((P[\alpha \rightsquigarrow (\gamma; \delta)])[\delta \rightsquigarrow Q_2])[\gamma \rightsquigarrow Q_1]) \models$
$\mathcal{R}ed(((\varphi[\alpha \rightsquigarrow (\gamma; \delta)])[\delta \rightsquigarrow Q_2])[\gamma \rightsquigarrow Q_1])$
(By Remark 2.17 and Remark 4.14)

$\Leftrightarrow$ $red(P[\alpha \rightsquigarrow (Q_1; Q_2)]) \models \mathcal{R}ed(([\alpha]\varphi')[\alpha \rightsquigarrow (Q_1; Q_2)])$
(By Lemma 2.19 and Lemma 4.15)

<u>$\varphi = \langle\gamma\rangle\varphi'$:</u>

'$\Rightarrow$': The proof of this direction proceeds in analogy to the '$\Leftarrow$'-direction of the case $\varphi = [\gamma]\varphi'$.

'$\Leftarrow$': The proof of this direction proceeds in analogy to the '$\Rightarrow$'-direction of the case $\varphi = [\gamma]\varphi'$.

<u>$\varphi = \sigma^\lambda Z.\varphi'$:</u>

First consider the case $\lambda = 0$. If $\sigma = \nu$, then φ is eqivalent to $\top$ whereas φ is equivalent to $\bot$ in the case $\sigma = \mu$. Hence we can apply the base case to complete this step.

Now let λ be a limit ordinal and $\sigma = \nu$, i.e. $\varphi = \nu^\lambda Z.\varphi'$. We have $P \models \varphi$

$$\text{iff } P \models \bigwedge_{\kappa\in\lambda} \nu^\kappa Z.\varphi'$$

(By Definition 4.22)

$$\text{iff } \forall\kappa < \lambda (P \models \nu^\kappa Z.\varphi')$$

(By Definition 4.22)

$$\text{iff } \forall\kappa < \lambda\Big(\mathit{red}(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}\mathit{ed}((\nu^\kappa Z.\varphi')[\alpha \rightsquigarrow Q])\Big)$$

(By the induction hypothesis. Note that $d(\nu^\kappa Z.\varphi') < d(\varphi)$)

$$\text{iff } \forall\kappa < \lambda\Big(\mathit{red}(P[\alpha \rightsquigarrow Q]) \models \nu^\kappa Z.\mathcal{R}\mathit{ed}(\varphi[\alpha \rightsquigarrow Q])\Big)$$

(By Definition 4.5 and Definition 4.13)

$$\text{iff } \mathit{red}(P[\alpha \rightsquigarrow Q]) \models \bigwedge_{\kappa\in\lambda} \nu^\kappa Z.\mathcal{R}\mathit{ed}(\varphi[\alpha \rightsquigarrow Q])$$

(By Definition 4.5)

$$\text{iff } \mathit{red}(P[\alpha \rightsquigarrow Q]) \models \nu^\lambda Z.\mathcal{R}\mathit{ed}(\varphi[\alpha \rightsquigarrow Q])$$

(By Definition 4.22)

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed((\nu^\lambda Z.\varphi)[\alpha \rightsquigarrow Q])$$

(By Definition 4.5 and Definition 4.13)

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$$

Let λ be a limit ordinal and $\sigma = \mu$, i.e. $\varphi = \mu^\lambda Z.\varphi'$. The proof of this case proceeds in analogy to the proof of the preceeding case.

Now let λ be a successor ordinal. For $n \geq 1$ and $\lambda_1 + 1 = \lambda$, we prove the claim of the theorem for formulas

$$\varphi = \sigma^{\lambda_1+1} Z_1.\sigma^{\lambda_2+1} Z_2.\dots\sigma^{\lambda_n+1} Z_n.\psi$$

where ψ is of the form $\top$, $\bot$, $\langle\alpha\rangle\varphi'$, $[\alpha]\varphi'$, $\bigvee_{\lambda\in I}\varphi_\lambda$, $\bigwedge_{\lambda\in I}\varphi_\lambda$, $\sigma Z.\varphi'$ or $\sigma^\kappa Z.\varphi'$ (κ a limit ordinal): By assertion 1) of Lemma 4.32 follows

$$P \models \varphi \quad \text{iff} \quad P \models \psi[\phi_1/Z_1]\dots[\phi_n/Z_n]$$

where the formulas ϕ_i $(1 \leq i \leq n)$ are given according to Lemma 4.32. Note that by assertion 3) of Lemma 4.32 we have $d(\varphi) = d(\psi[\phi_1/Z_1]\dots[\phi_n/Z_n])$. In the cases where ψ is of the form $\top$, $\bot$ or $\sigma^\lambda Z.\varphi'$ (where $\lambda \in \mathcal{O}n$ is a limit ordinal) the present case can be readily reduced to these previous cases which gives

$$P \models \varphi \quad \text{iff} \quad red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed((\psi[\phi_1/Z_1]\dots[\phi_n/Z_n])[\alpha \rightsquigarrow Q]).$$

We show the induction argument for the other cases. For $m \in \mathbb{N}$, let $[\phi/Z]^{\leq m}$ abbreviate the sequence of substitutions $[\phi_1/Z_1]\dots[\phi_m/Z_m]$.

$\psi = \bigotimes \varphi_\eta$ where $\bigotimes \in \{\bigvee_{\eta\in I}, \bigwedge_{\eta\in I}\}$:

Then

$$\psi[\phi/Z]^{\leq n} = \Big(\bigotimes \varphi_\eta\Big)[\phi/Z]^{\leq n} = \bigotimes \varphi_\eta[\phi/Z]^{\leq n}$$

Clearly

$$d(\bigotimes \varphi_\eta[\phi/Z]^{\leq n}) = \sup_{\eta\in I}(d(\varphi_\eta[\phi/Z]^{\leq n})) + 1$$

$$> \sup_{\eta \in I}(d(\varphi_\eta[\phi/Z]^{\leq n})) \geq d(\varphi_\eta[\phi/Z]^{\leq n}) \text{ for any } \eta \in I.$$

Hence the induction hypothesis is applicable to $\varphi_\eta[\phi/Z]^{\leq n}$ for any $\eta \in I$. Consequently, we can reduce the current case to previous case where $\varphi = \bigotimes \varphi_\eta$ and $\bigotimes \in \{\bigvee_{\eta \in I}, \bigwedge_{\eta \in I}\}$.

$\psi = \triangle_\alpha \psi'$ where $\triangle_\alpha \in \{\langle\alpha\rangle, [\alpha]\}$. Then

$$\psi[\phi/Z]^{\leq n} = (\triangle_\alpha \psi')[\phi/Z]^{\leq n} = \triangle_\alpha \psi'[\phi/Z]^{\leq n}.$$

Clearly we have

$$d(\triangle_\alpha \psi'[\phi/Z]^{\leq n}) = d(\psi'[\phi/Z]^{\leq n}) + 1 > d(\psi'[\phi/Z]^{\leq n})$$

Hence, the induction hypothesis is applicable to $\psi'[\phi/Z]^{\leq n}$ whence we can reduce the current case to the previous case where $\varphi = \triangle_\alpha \varphi'$ and $\triangle_\alpha \in \{\langle\alpha\rangle, [\alpha]\}$.

By assertion 2) of Lemma 4.32 it follows that $red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$. ■

Theorem 4.37
Let $P \in G\Sigma$ be a process term and $\varphi \in CG\mu\mathcal{L}$ be a formula. Further let $Q \in \Delta$ be a distinct process term, such that P is alphabet-disjoint from Q and φ is ξ-disjoint from Q. Then $P \models \varphi \Leftrightarrow red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$.

Proof: Let $\varphi \in CG\mu\mathcal{L}$. Then

$$P \models \varphi$$

$$\text{iff } P \models \mathcal{I}(\varphi) \quad \text{(By Lemma 4.28)}$$

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\mathcal{I}(\varphi)[\alpha \rightsquigarrow Q])$$

(This follows from Theorem 4.36 since $\mathcal{I}(\varphi) \in CGApp$. Note that $\phi \in \mu\mathcal{L}$ is closed and guarded iff $\mathcal{I}(\phi) \in App$ is closed and guarded. Further, $\phi \in \mu\mathcal{L}$ is alphabet-disjoint from $Q \in \Delta$ iff $\mathcal{I}(\phi) \in App$ is alphabet-disjoint from Q.)

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \mathcal{I}(\mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])) \quad \text{(By Lemma 4.21)}$$

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q]) \quad \text{(By Lemma 4.28)}$$

■

Theorem 4.38
Let $P \in G\Sigma$ be a process term and $\varphi \in CG\mu\mathcal{L}$ be a formula. Further let $Q \in R\Delta$ be a distinct process term, such that P is alphabet-disjoint from Q and φ is ξ-disjoint from Q. Then $P \models \varphi \Leftrightarrow red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$.

Proof: Assume $red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$. We have that $red(Q) = red(red(Q))$. Hence, the assumption is equivalent to

$$red(P[\alpha \rightsquigarrow red(Q)]) \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow red(Q)]) \quad (*)$$

which follows from Definition 2.11, 2.15, 4.5 and Definition 4.13. By Lemma 2.18 we have $\xi(Q) = \xi(red(Q))$. Further, Q is distinct iff $red(Q)$ is distinct. Since $red(Q) \in \Delta$, it follows from Theorem 4.37 that assertion $(*)$ is equivalent to the assertion $P \models \varphi$. ■

The next three lemmata formalize the intuition, that process terms $P[\alpha \rightsquigarrow Q]$ and formulas $\varphi[\alpha \rightsquigarrow Q]$ exhibit the same semantics as the reduced process term $red(P[\alpha \rightsquigarrow Q])$ and $\mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$ respectively.

Lemma 4.39
Let $P \in R\Sigma$ and $\varphi \in \mu\mathcal{L}$. Then $P \models_\vartheta \varphi$ iff $red(P) \models_\vartheta \varphi$.

Proof: $\mathcal{T}(P) = \mathcal{T}(red(P))$ by definition. ■

Lemma 4.40
Let $P \in R\Sigma$ and $\varphi \in R\mu\mathcal{L}$. Then $P \models_\vartheta \varphi \Leftrightarrow P \models_\vartheta \mathcal{R}ed(\varphi)$.

Proof: The proof is by induction on the structure of $\varphi \in R\mu\mathcal{L}$. ■

Corollary 4.41
Let $P \in R\Sigma$ and $\varphi \in R\mu\mathcal{L}$. Then $P \models_\vartheta \varphi \Leftrightarrow red(P) \models_\vartheta \varphi$.

Proof: Follows from Lemma 4.39 and Lemma 4.40. ■

Theorem 4.42 (Simultaneous syntactic action refinement)
Let $P \in GR\Sigma$ be a process term and $\varphi \in CG\mu\mathcal{L}$ be a formula. Further let $Q \in R\Delta$ be a distinct process term such that P is alphabet-disjoint from Q and φ is ξ-disjoint from Q. Then $P \models \varphi \Leftrightarrow P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q]$.

Proof:

$$P \models \varphi$$

$$\text{iff } red(P) \models \varphi \quad \text{(By Corollary 4.41)}$$

$$\text{iff } red(P) \models \mathcal{R}ed(\varphi) \quad \text{(By Lemma 4.40)}$$

$$\text{iff } red(red(P)[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\mathcal{R}ed(\varphi)[\alpha \rightsquigarrow Q]) \quad \text{(By Theorem 4.38)}$$

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q]) \quad \text{(By Remark 2.17 and Remark 4.14)}$$

$$\text{iff } P[\alpha \rightsquigarrow Q] \models \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q]) \quad \text{(By Corollary 4.41)}$$

$$\text{iff } P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q] \quad \text{(By Lemma 4.40)}$$

■

Before proceeding to the case study in the next section, we demonstrate the applicability of Theorem 4.42 by means of a simple example.

Example 4.43 (A priori-verification and abstraction)
Consider the following 'assembly line' P in a car factory shown in Figure 7.

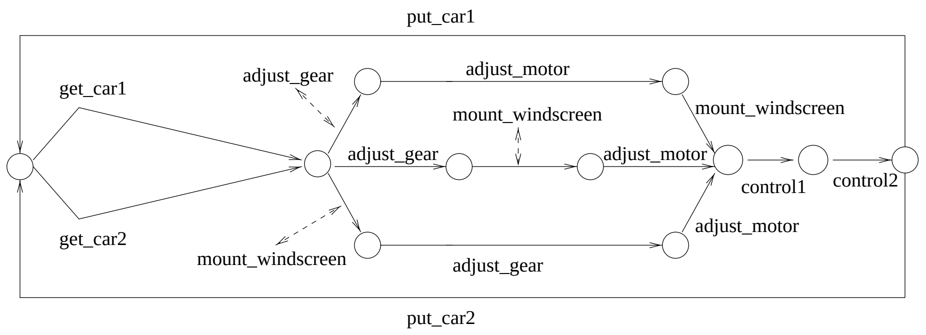

Figure 7: The Process P

The complete description of the assembly line is given by the program P shown in Figure 8. The job of P is to adjust the motor and the gear of a car and to mount the windscreen. Hence P can be modelled by means of a few atomic actions. get_car1 (get_car2): get a car from conveyer band one (two resp.), $adjust\_gear$, $adjust\_motor$, $mount\_windscreen$, put_car1 (put_car2): put the car back on conveyer band one (two resp.). To reach a defined system status before the car is carried back to the conveyer band two control actions are executed by P.

The process P has the temporal property that 'whenever a car is taken from the conveyer band (either get_car1 or get_car2 is executed), the control actions will even-

$fix(x = Q; x)$ where $Q =$

$(get\_car; (((adjust; control)\|_{\{control1, control2\}}(mount\_windscreen; control)); put\_car))$

such that

$get\_car = (get\_car1 + get\_car2)$

$adjust = (adjust\_gear; adjust\_motor)$

$put\_car = (put\_car1 + put\_car2)$

$control = (control1; control2)$

Figure 8: The Process Term P

tually be executed'. We denote this property by the formula[19] $\varphi = \nu Z.(([gc1]\mu Y.(\Phi \wedge \Upsilon) \wedge \Psi) \wedge ([gc2]\mu Y.(\Phi \wedge \Upsilon) \wedge \Psi))$ *where* $\Phi =$

$$\Big((\langle gc1\rangle\top \wedge \langle gc2\rangle\top) \vee (\langle ag\rangle\langle am\rangle\top \vee (\langle mw\rangle\top \vee (\langle c1\rangle\langle c2\rangle\top \vee (\langle pc1\rangle\top \wedge \langle pc2\rangle\top))))\Big)$$

$$\Upsilon = \Big(([gc1]Y \wedge [gc2]Y) \wedge ([ag][am]Y \wedge ([mw]Y \wedge (([pc1]Y \wedge [pc2]Y))))\Big)$$

$$\Psi = \Big(([gc1]Z \wedge [gc2]Z) \wedge ([ag][am]Z \wedge ([mw]Z \wedge ([c1][c2]Z \wedge ([pc1]Z \wedge [pc2]Z))))\Big)$$

in the logic $\mu\mathcal{L}$.

The process P *arises from the process* P_s *shown in Figure 9 by the application of four successive refinement steps, that is,*

$$P_1 = P_s[control \rightsquigarrow (control1; control2)]$$
$$P_2 = P_1[put\_car \rightsquigarrow (put\_car1 + put\_car2)]$$
$$P_3 = P_2[adjust \rightsquigarrow (adjust\_gear; adjust\_motor)]$$
$$P = P_3[get\_car \rightsquigarrow (get\_car1 + get\_car2)]$$

where $P_s = fix(x = (\tilde{Q}; x))$ *and* $\tilde{Q}$ *abbreviates the expression* $(get\_car; (((adjust; control)\|_{\{control\}}(mount\_windscreen; control)); put\_car))$. *Let us assume that we had already established* $P_s \models \varphi_s$ *where* $\varphi_s = \nu Z.([gc]\mu Y.(\Phi_s \wedge \Upsilon_s) \wedge \Psi_s)$ *and*

$$\Phi_s = (\langle gc\rangle\top \vee (\langle a\rangle\top \vee (\langle mw\rangle\top \vee (\langle c\rangle\top \vee (\langle pc\rangle\top)))))$$

[19]Formulas will sometimes be given in a more concise form using the obvious abbreviations for action names like $gc1$ ($pc2$,mw,ag,am,$c1$) for get_car1 ($put\_car2, mount\_windscreen, adjust\_gear, adjust\_motor, control1$ resp.).

$$\Upsilon_s = ([gc]Y \wedge ([a]Y \wedge ([mw]Y \wedge ([pc]Y))))$$

$$\Psi_s = ([gc]Z \wedge ([a]Z \wedge ([mw]Z \wedge ([c]Z \wedge ([pc]Z)))))$$

(Again, the obvious abbreviations for action names of P_s are used in φ_s). Now

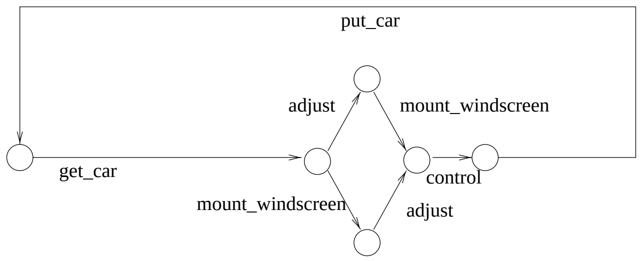

Figure 9: The Process P_s

$$\begin{aligned}\varphi_1 &= \varphi_s[control \leadsto (control1; control2)]\\ \varphi_2 &= \varphi_1[put\_car \leadsto (put\_car1 + put\_car2)]\\ \varphi_3 &= \varphi_2[adjust \leadsto (adjust\_gear; adjust\_motor)]\\ \varphi &= \varphi_3[get\_car \leadsto (get\_car1 + get\_car2)]\end{aligned}$$

whence $P_s \models \varphi_s$ iff $P \models \varphi$ follows by Theorem 4.42. □

It is well known, that the Modal Mu-Calculus induces strong bisimulation equivalence (in the sense of Milner [154]) on the set of (finitely branching) transition systems (see, for example, [192]). To exploit this fact for our approach, we lift bisimulation equivalence to the set $R\Sigma$ by defining $P \sim_b P'$ iff $\mathcal{T}(P) \sim_b \mathcal{T}(P')$. As a direct consequence of Theorem 4.42 we then obtain the following "vertical modularity" result.

Corollary 4.44

Let $P, P' \in R\Sigma$ be guarded process terms and $\varphi \in R\mu\mathcal{L}$ be a closed and guarded formula. Let $Q_1, \ldots, Q_n \in R\Delta$ be distinct and pairwise alphabet-disjoint. Let Q_i be such, that that P and φ are alphabet-disjoint from Q_i, $1 \leq i \leq n$. Let $[\alpha \leadsto Q]_n$ abbreviate $[\alpha_1 \leadsto Q_1], \ldots, [\alpha_n \leadsto Q_n]$. If $P \sim_b P'$ then $P[\alpha \leadsto Q]_n \models \varphi[\alpha \leadsto Q]_n \Leftrightarrow P'[\alpha \leadsto Q]_n \models \varphi[\alpha \leadsto Q]_n$.

Proof: Follows immediately from Theorem 4.42 ■

Corollary 4.44 can thus be used after any development sequence to syntactically interchange the original "target-process term" P with a term P', provided P and P' are strongly bisimular.

Remark 4.45
Clearly, we can replace the premise $P \sim_b P'$ by the premise $P' \models \varphi$. Using model checking however, the best algorithm known hitherto needs in the worst case time $O(alt(\varphi)^2 (N_P + 1)^{\lfloor alt(\varphi)/2 \rfloor + 1})$ to decide $P' \models \varphi$ and space about $N_P^{alt(\varphi)/2}$ where $alt(\varphi)$ is the alternation depth of fixed point operators in φ, and N_P is the number of states of $\mathcal{T}(P)$ (see [134]). In contrast, deciding bisimilarity for two processes P, P' needs time $O(M_P + M_{P'} \log N_P + N_{P'})$ and space $O(M_P + M_{P'} + N_P + N_{P'})$ (see [165]) where M_P is the number of transitions of $\mathcal{T}(P)$. □

The abstraction technique comprised by Theorem 4.42 can be used to abstract those parts of the system description that are irrelevant for the verification at hand. For a given system description P and a property φ we construct a small description P_s and a small formula φ_s, such that we can establish $P = P_s[\alpha_1 \rightsquigarrow Q_1] \ldots [\alpha_n \rightsquigarrow Q_n]$ and $\varphi = \varphi_s[\alpha_1 \rightsquigarrow Q_1] \ldots [\alpha_n \rightsquigarrow Q_n]$. It then suffices to decide $P_s \models \varphi_s$ in order to show whether $P \models \varphi$ holds or not. As the case study in the next section shows, the size of the state space of P_s can be exponentially smaller (with respect to $\sum_{i=1}^{n} |Q_i|$) than the size of the state space of P (see also [139]).

4.2.1 Applications of SSAR: A Case Study

While the application of Theorem 4.42 to develop/re-engineer reactive systems can readily be seen, applying Theorem 4.42 as an *abstraction technique* to enhance model checking might require some further illustration. To this end, we consider a "data processing-environment" (DPE) which consists of a central data base and several users of the data base. Conceptually, our example is similar to Milner's *scheduler* [154] or to the *IEEE Futurebus+* (considered, for example, in [43]) as several structurally equal subsystems are executed in parallel. To ensure the consistency of the data base, it must be accessed in mutual exclusion by the users. Thus, the data base represents a critical section and accessing it is controlled by parameterized read-and write semaphores.

We assume a situation where a DPE has already been implemented and we want to prove, that the given implementation has a desirable property. In order to demonstrate how our approach allows to fix bug's at high levels of abstraction (instead of fixing the bug at the complex concrete level) we deliberately start with a faulty implementation.

Instead of model checking that the concrete system is faulty, we first construct an abstract system and model check that the abstract system contains an according (abstract) bug. Using Theorem 4.42, we then infer that the concrete system is faulty as well. We then fix the bug on the abstract level and model check that the 'abstract' bug has been removed. Finally, Theorem 4.42 is applied again to automatically derive a corrected concrete system from the corrected abstract system.

Let us start with giving some implementation details. The i-th user of the DPE is modelled by the process term[20]

$$User_i := fix((x_i = PD_i; x_i) + (v_i^r; read_i; p_i^r; x_i) + (v_i^w; write_i; p_i^w; x_i)).$$

We define $USER^n := (User_1 ||_\emptyset User_2 ||_\emptyset, \ldots ||_\emptyset User_n)$. $User_i$ can either process (local) data by executing the subsystem PD_i or access the data base (to read or write data) by coordinating with a particular control process $Cont_i$. For user $User_i$ we thus use a control process $Cont_i$, implemented by the process term

$$Cont_i := fix(y_i = (v_i^r; read_i; p_i^r; y_i) + (v_i^w; write_i; p_i^w; y_i)).$$

Let us first consider a faulty control component defined by

$$CONT^n := (Cont_1 ||_\emptyset, Cont_2 ||_\emptyset, \ldots, ||_\emptyset Cont_n).$$

A correct control component is

$$CorrCONT^n := (Cont_1 + Cont_2 + \ldots + Cont_n).$$

We next define a faulty and a correct DPE parameterized with respect to the number of users, that is,

$$DPE(n) = (USER^n ||_{\{v_i^r, v_i^w, read_i, write_i, p_i^r, p_i^w \mid 1 \leq i \leq n\}} CONT^n)$$

and

$$CorrDPE(n) = (USER^n ||_{\{v_i^r, v_i^w, read_i, write_i, p_i^r, p_i^w \mid 1 \leq i \leq n\}} CorrCONT^n).$$

[20]In the example, we sometimes omit parenthesis in order to support readability.

$User_i$ can read data from the data base if $User_i$ and $Cont_i$ can jointly execute v_i^r ($User_i$ occupies the read-semaphore), $read_i$ ($User_i$ reads data) and p_i^r ($User_i$ releases the read-semaphore). As PD_i is assumed to be a 'local subsystem' of $User_i$, it is reasonable to require that PD_i and PD_j contain no common actions for $i \neq j$. Further, we assume PD_i $(1 \leq i \leq n)$ to be distinct. Since the control component $CONT^n$ executes the control processes $Cont_i$ $(1 \leq i \leq n)$ concurrently, mutual exclusive access to the data base is not guaranteed.

We now consider a (faulty) "four user DPE" $DPE(4)$. We would like to prove that $User_1$ and $User_2$ cannot write data at the same time as long as only actions from $User_1$ and $User_2$ are executed by $DPE(4)$. In other words, we would like to show that $DPE(4)$ has no computation sequence (that consists of actions from $User_1$ and $User_2$) which leads to a state where the actions $write_1$ and $write_2$ can both be executed. This amounts to show, that $DPE(4)$ has no such computation path which leads to such a 'bad state'. In order to do this, we try to disprove that $DPE(4)$ has a computation path along which a bad state is reachable. This property can be expressed by the Modal Mu-Calculus formula

$$\phi_{error}^{i,j} = \mu Z.(\langle write_i \rangle \top \wedge \langle write_j \rangle \top) \vee \langle alph(User_i) \cup alph(User_j) \rangle Z$$

for $i = 1$ and $j = 2$. In the above formula, $alph(P)$ denotes the set of actions that occur in a process term P and $\langle A \rangle \varphi$ abbreviates the formula $\langle \alpha_1 \rangle \varphi \vee \langle \alpha_2 \rangle \varphi, \ldots, \langle \alpha_n \rangle \varphi$ for $\alpha_1, \ldots, \alpha_n \in A$.

It turns out that the considered implementation of the DPE is faulty, that is, $DPE(4) \models \phi_{error}^{1,2}$. This could be proved directly by using a model checker. However, depending on the terms PD_i $(i = 1, 2, 3, 4)$, the state space of $DPE(4)$ can become tremendous due to the state explosion problem. In order to model check that $DPE(4) \models \phi_{error}^{1,2}$ we first abstract away those implementation details of $DPE(4)$ that are irrelevant for the verification. To this end, we define

$$SmallUser_i := fix(x_i = (pd_i; x_j + (r_i; x_i + w_i; x_i))$$

and

$$SmallCont_i := fix(x_i = r_i; x_i + w_i; x_i).$$

Using these process terms, we define

$$DPE4_{small} = \Big(USER^2 \|_{\emptyset} SmallUser_3 \|_{\emptyset} SmallUser_4 \|_L$$

$$CONT^2 \|_\emptyset SmallCont_3 \|_\emptyset SmallCont_4\Big)$$

where $L = \{v_i^r, v_i^w, read_i, write_i, p_i^r, p_i^w, r_j, w_j \mid i = 1,2 \text{ and } j = 3,4\}$. We can then establish the refinement

$$T = DPE4_{small}[pd_3 \rightsquigarrow PD_3][r_3 \rightsquigarrow v_3^r; read_3; p_3^r][w_3 \rightsquigarrow v_3^w; write_3; p_3^w]$$

$$DPE(4) = T[pd_4 \rightsquigarrow PD_4][r_4 \rightsquigarrow v_4^r; read_4; p_4^r][w_4 \rightsquigarrow v_4^w; write_4; p_4^w].$$

Note that the formula $\phi_{error}^{1,2}$ remains unchanged under the above refinements followed by logical reduction[21]. By Theorem 4.42, it suffices to model check that $DPE4_{small} \models \phi_{error}^{1,2}$ to conclude that $DPE(4) \models \phi_{error}^{1,2}$. In what follows, we let PD_i be implemented by three sequential actions. Then the state space of $DPE4_{small}$ only contains 10 states whence it is about 8 times smaller than the state space of $DPE(4)$.

We can now fix the bug on the abstract level by using the correct control component:

$$CorrDPE4_{small} = \Big(USER^2 \|_\emptyset SmallUser_3 \|_\emptyset SmallUser_4 \|_L$$

$$(CorrCONT^2 + SmallCont_3 + SmallCont_4)\Big)$$

Model checking can now be used on the abstract level to show that we have

$$CorrDPE4_{small} \not\models \phi_{error}^{1,2}.$$

For

$$T' = CorrDPE4_{small}[pd_3 \rightsquigarrow PD_3][r_3 \rightsquigarrow v_3^r; read_3; p_3^r][w_3 \rightsquigarrow v_3^w; write_3; p_3^w]$$

$$CorrDPE(4) = T'[pd_4 \rightsquigarrow PD_4][r_4 \rightsquigarrow v_4^r; read_4; p_4^r][w_4 \rightsquigarrow v_4^w; write_4; p_4^w].$$

we can immediately conclude (using Theorem 1 again) that

$$CorrDPE(4) \not\models \phi_{error}^{1,2}.$$

The example above shows, that those parts of the system description that share no actions with the formula under consideration can be immediately abstracted. We believe that this makes precise, which parts of the system description are completely

[21] Let ψ be the formula that arises by applying the above refinement operators to the formula $\phi_{error}^{1,2}$. Then $\phi_{error}^{1,2} = \mathcal{R}ed(\psi)$ whence $P \models \psi$ iff (by definition) $P \models \mathcal{R}ed(\psi)$ iff $P \models \phi_{error}^{1,2}$.

irrelevant for the actual verification task and that such situations (where the property of interest 'refers' only to a part of the system) often occur in practice.

It is clear, that the state space of $DPE(i)$ grows exponentially in the number i of DPE-users. The state space of $DPE(8)$ contains about 13000 states whereas a system abstracted with the above strategy contained 19 states, a 680-fold reduction of the state space[22]. Note that we can exploit the above sketched strategy to disprove mutual exclusive write-access (in the above sense) of all users $User_i$. This property can be expressed by the formula

$$\Phi^n_{error} = \bigwedge_{i<j\leq n} \phi^{i,j}_{error} \ .$$

The application of model checking to verify all conjuncts in the above formula amounts to check a total of about 530 states in order to prove that $DPE(8) \models \Phi^8_{error}$. In contrary, classical model checking would necessitate to create the whole state space of 13000 states in order to verify this property.

Additional logical reasoning (based on the structure of the system) might be necessary if we want to abstract parts of the process term, that share action with the formula under consideration. For further abstracting the (faulty) $DPE(4)$-example, assume $PD_i = t^i_1; t^i_2; t^i_3$ $(i = 1, 2)$. We can then use the formula

$$\Psi_{error} = \mu Z.(\langle write_1\rangle\langle p^w_1\rangle\top \wedge \langle write_2\rangle\langle p^w_2\rangle\top) \vee \langle t^1_1\rangle\langle t^1_2\rangle\langle t^1_3\rangle\langle t^2_1\rangle\langle t^2_2\rangle\langle t^2_3\rangle Z$$

to carry out some more abstractions since we have that $DPE4_{small} \models \Psi_{error}$ implies $DPE4_{small} \models \phi^{1,2}_{error}$ (showing the validity of this implication is the above mentioned additional logical reasoning). We proceed as follows:
Let $DPE4_{VerySmall}$ be the process term that arises from $DPE4_{small}$ by substituting the process term PD_i by the action pd_i, the term $read_i; p^r_i$ by the action r_i and the term $write_i; p^w_i$ by the action w_i where $i = 1, 2$. If

$$T = DPE4_{VerySmall}[pd_1 \rightsquigarrow PD_1][pd_2 \rightsquigarrow PD_2][r_1 \rightsquigarrow read_1; p^r_1]$$

then

$$DPE4_{small} = T[r_2 \rightsquigarrow read_2; p^r_2][w_1 \rightsquigarrow write_1; p^w_1][w_2 \rightsquigarrow write_2; p^w_2].$$

[22]We used the Edinburgh Concurrency Workbench 7.0 [49] to calculate the size of the state spaces.

Now consider the formula

$$\Theta_{error} = \mu Z.(\langle w_1\rangle\top \wedge \langle w_2\rangle\top) \vee \langle pd_1\rangle\langle pd_2\rangle Z.$$

We have that

$$\psi = \Theta_{error}[pd_1 \rightsquigarrow PD_1][pd_2 \rightsquigarrow PD_2][r_1 \rightsquigarrow read_1; p_1^r]$$

$$\Psi_{error} = \psi[r_2 \rightsquigarrow read_2; p_2^r][w_1 \rightsquigarrow write_1; p_1^w][w_2 \rightsquigarrow write_2; p_2^w].$$

We use model checking to show that $DPE4_{VerySmall} \models \Theta_{error}$. By Theorem 4.42 follows $DPE4_{small} \models \Psi_{error}$ and hence $DPE4_{small} \models \phi^{1,2}_{error}$.

User-guidance (involving additional logical reasoning) seems to be necessary in situations, where system parts that share actions with the formula under consideration are abstracted.

4.2.2 Discussion

The equivalence in Theorem 4.42 guarantees that the reduction functions *red* and $\mathcal{R}ed$ are defined appropriately as it excludes the use of nonsensical reduction functions: Using the definition $\mathcal{R}ed(\varphi) = \top$ would trivially validate the implication from left to right.

It is clear, that (logical) SAR as used in Theorem 4.42 is not complete in the sense, that we cannot derive every (interesting) formula ψ from a formula φ. We believe however, that Theorem 4.42 can always be useful to provide "basic knowledge" in the overall development procedure.

We have argued that alphabet-disjoint process expressions and formulas can be obtained by renaming the actions of Q in the obvious way. Alphabet-disjointness can also be achieved by additional refinements. Consider, for example, the terms $P = (\alpha + \beta)$ and $Q = (\beta; \gamma)$. Then P is not alphabet-disjoint from Q. We can solve this problem by applying additional refinements as follows: We consider $P_1 = P[\alpha \rightsquigarrow \alpha_1][\beta \rightsquigarrow \beta_1]$ instead of P and $Q_2 = Q[\beta \rightsquigarrow \beta_2][\gamma \rightsquigarrow \gamma_2]$ instead of Q where the indices signalize the level of abstraction. Instead of $P[\alpha \rightsquigarrow Q]$ we can then work with $P_1[\alpha_1 \rightsquigarrow Q_2]$ which satisfies the conditions of Theorem 4.42. We have argued that such renamings are consistent with existing approaches to action refinement (see also [88]).

As the case study in Section 4.2.1 demonstrates, the restriction of distinctness of Q is not severe and often ensues naturally. It could be argued further, that distinctness prohibits Q from being non-deterministic. Introducing non-determinism while making a specification more precise (through action refinement) seems to be counterproductive.

However, in Section 4.3 we show that both, the condition of alphabet-disjointness and the condition of distinctness can be dropped completely provided we restrict ourselves to particular fragments of the logic $R\mu\mathcal{L}$.

A potential drawback of Theorem 4.42 is the length of the formulas that are generated by the reduction of $R\mu\mathcal{L}$-formulas: Provided $\varphi \in \mu\mathcal{L}$ and $Q \in \Delta$, we have that $|\mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])| = O(2^{|\varphi|*|Q|})$. If we let $Q' \in \Delta$ we have

$$|\mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q][\beta \rightsquigarrow Q'])| = O(2^{2^{|\varphi|*|Q|}*|Q'|})$$

and so on. Such "exponential blow-ups" will be avoided by using the generalized Modal Mu-Calculus (see Section 4.4) and a suitable reduction function (see Definition 4.54 and Definition 4.55).

4.3 SSAR for Fragments of the Modal Mu-Calculus

Renaming of actions can often be applied successfully in order to meet the conditions of alphabet disjointness. However, this condition rules out the possibility to conduct particular refinement steps which can become important in the development of reactive systems: Suppose the system P can execute the atomic actions a, b. At the current level of abstraction, the action a (b) is considered to be the name of a procedure Q_a (Q_b respectively.) which is not yet implemented. In an intermediate development step, Q_a and Q_b are implemented making use of a common subsystem S which we might assume has been provided by a system library. Hence, alphabet disjointness of Q_a and Q_b does not hold. It is clear that dropping the condition of alphabet-disjointness of the process terms P from Q only makes sense in conjunction with abandoning alphabet-disjointness of φ from Q. Dropping the latter restriction however leads to fundamental problems: Without it, repeated syntactic action refinement might transform an originally satisfiable formula into an unsatisfiable one:

Example 4.46
Let $\varphi = (\langle\alpha_1\rangle\top \wedge [\alpha_2]\bot)$. *Clearly,* φ *is satisfiable. On the other hand, the formula* $\varphi[\alpha_1 \rightsquigarrow \alpha][\alpha_2 \rightsquigarrow \alpha]$ *is not satisfiable.* □

The above example show that we cannot hope for a result like Theorem 4.42 for any fragment $L \subseteq R\mu\mathcal{L}$ in which it is allowed to compose formulas $\varphi \in L$ containing both types of modalities, that is, $\langle\alpha\rangle$ and $[\alpha]$ without accepting any restrictions on alphabet-disjointness. This is the reason why we consider the logics $R\mu\mathcal{L}_{\langle\cdot\rangle}$ and $R\mu\mathcal{L}_{[\cdot]}$ where respectively only one modality type might occur in the formulas. Intuitively, the language $R\mu\mathcal{L}_{\langle\cdot\rangle}$ allows to specify properties of reactive systems where the branching character of computations is neglected, that is, the focus lies on specifying particular computation paths rather than computation trees as in the logic $R\mu\mathcal{L}$ whence it can be used to specify computation paths which must be executable by a system. The expressiveness of these two fragments of the full Modal Mu-Calculus has already been investigated in [21, 16].

Definition 4.47 (The fragments $R\mu\mathcal{L}_{\langle\cdot\rangle}$ and $R\mu\mathcal{L}_{[\cdot]}$)
Let $\mu\mathcal{L}_{\langle\cdot\rangle} \subset \mu\mathcal{L}$ *be the language generated by the grammar*

$$\Phi ::= \top \mid \bot \mid Z \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi) \mid \langle\alpha\rangle\Phi \mid \nu Z.\Phi \mid \mu Z.\Phi$$

and $\mu\mathcal{L}_{[\cdot]} \subset \mu\mathcal{L}$ *be the language generated by the grammar*

$$\Phi ::= \top \mid \bot \mid Z \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi) \mid [\alpha]\Phi \mid \nu Z.\Phi \mid \mu Z.\Phi$$

where α *ranges over the set* $\mathcal{A}$ *and* Z *ranges over a fixed set* Var *of variables.*

Let $R\mu\mathcal{L}_{\langle\cdot\rangle}$ $(R\mu\mathcal{L}_{[\cdot]})$ *be the language generated by the grammar for* $\mu\mathcal{L}_{\langle\cdot\rangle}$ $(\mu\mathcal{L}_{[\cdot]}$ *respectively.) with the additional rule* $\Phi ::= \Phi[\alpha \rightsquigarrow Q]$ *where* $Q \in R\Delta$. □

The logics $R\mu\mathcal{L}_{\langle\cdot\rangle}$ and $R\mu\mathcal{L}_{[\cdot]}$ can be used to express interesting properties of reactive systems:

Example 4.48

- *The unless-property "*Φ *remains true in every computation unless* Ψ*" is true for a process if*

$$P \models \nu Z.(\Psi \vee (\Phi \wedge \Box_{\xi(P)} Z)) \in R\mu\mathcal{L}_{[\cdot]}$$

where $\Box_A \phi := ([\alpha_1]\phi \wedge ([\alpha_2]\phi \wedge (\ldots([\alpha_{n-1}]\phi \wedge [\alpha_n]\phi)\ldots)$ *for* $A = \{\alpha_1, \ldots, \alpha_n\}$.

- *Let the complement Φ^c of the formula Φ (see, for example, [192]) capture the "bad states" which should not be reached by the system P. Then P satisfies the safety-property "P never reaches a bad state whenever Ψ has become true", if*

$$P \models \nu Z.((\Psi^c \vee (\Psi \wedge \nu Y.(\Phi \wedge \Box_{\xi(P)} Y))) \wedge \Box_{\xi(P)} Z) \in R\mu\mathcal{L}_{[\cdot]}.$$

- *The process P satisfies the guarantee-property [142] "eventually Φ in any infinite computation sequence of P" if*

$$P \models \mu Z.(\Phi \vee \Box_{\xi(P)} Z) \in R\mu\mathcal{L}_{[\cdot]}.$$

Further, $R\mu\mathcal{L}_{[\cdot]}$ can be used to express liveness-properties under fairness and cyclic-properties (see [192]). □

Example 4.49 (See [26])

- *If*

$$P \models \nu Y.\mu Z.((\Phi \vee \diamond_{\xi(P)} Z) \wedge \diamond_{\xi(P)} Y) \in R\mu\mathcal{L}_{\langle\cdot\rangle}$$

where $\diamond_A \phi := (\langle\alpha_1\rangle\phi \vee (\langle\alpha_2\rangle\phi \vee (\ldots(\langle\alpha_{n-1}\rangle\phi \vee \langle\alpha_n\rangle\phi)\ldots)$ for $A = \{\alpha_1, \ldots, \alpha_n\}$, then "there exists a computation sequence of P in which Φ holds infinitely often".

- *If*

$$P \models \nu Y.((\mu Z.(\Phi \vee \diamond_{\xi(P)} Z) \wedge \diamond_{\xi(P)} Y) \in R\mu\mathcal{L}_{\langle\cdot\rangle}$$

then "there exists a computation sequence of P along which Φ is always attainable". □

While dropping the conditions on alphabet-disjointness (and distinctness), we can still derive two special cases of Theorem 4.42 for the fragments $R\mu\mathcal{L}_{[\cdot]}$ and $R\mu\mathcal{L}_{\langle\cdot\rangle}$ of the logic $R\mu\mathcal{L}$.

Theorem 4.50 (Developing systems w.r.t. $R\mu\mathcal{L}_{\langle\cdot\rangle}$-properties)

Let $P \in UGR\Sigma$ and $Q \in R\Delta$ be process terms. Let $\varphi \in CGR\mu\mathcal{L}_{\langle\cdot\rangle}$ be a formula. Then we have the following:

1) If P is $\chi\xi$-disjoint from Q and $\alpha \notin \chi(P)$ or

2) if $\xi(\varphi) \subseteq \chi(P)$ *then*

$$P \models \varphi \Rightarrow P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q].$$

Proof: The proof of this theorem proceeds in analogy to the proof of Theorem 4.42: First, one has to prove the claim of the above theorem for process terms $P \in UG\Sigma$ and formulas $\varphi \in CGApp_{\langle\cdot\rangle}$ (the language $App_{\langle\cdot\rangle}$ is generated by the grammar of the language *App* where the rule $\Phi ::= [\alpha]\Phi$ is removed). Most of those induction steps are proved in analogy to the proof of this steps in Theorem 4.36 (of course only the implication from the left hand side to the right hand side is considered here). Hence, we only show the induction steps which differ substantially from the induction steps in Theorem 4.36.

$\varphi = \langle\gamma\rangle\varphi'$ where $\gamma \in \mathcal{A}$.

Case 1: $\varphi = \langle\beta\rangle\varphi'$ where $\beta \neq \alpha$.

Assume $P \models \varphi$. Then

$$\exists P' \in R\Sigma(P \xrightarrow{\beta} P' \wedge P' \models \varphi')$$

Obeying condition 1) of the theorem, we can immediately apply assertion 3) of Lemma 2.28. On the other hand, condition 2) and $\beta \in \xi(\varphi)$ imply $\beta \in \chi(P)$. Hence assertion 3) of Lemma 2.28 is also applicable in this case whence we have

$$red(P[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P'[\alpha \rightsquigarrow Q])$$

Further we have

$$red(P'[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\varphi'[\alpha \rightsquigarrow Q])$$

by the induction hypothesis, that is,

$$red(P[\alpha \rightsquigarrow Q]) \models \langle\beta\rangle\mathcal{R}ed(\varphi'[\alpha \rightsquigarrow Q])$$

by Definition 4.22. We conclude

$$red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed((\langle\beta\rangle\varphi')[\alpha \rightsquigarrow Q])$$

by Definition 2.11 and Definition 4.5.

Case 2: $\varphi = \langle\alpha\rangle\varphi'$.

Case discrimination on the structure of $Q \in R\Delta$.

$Q = \beta$.

Assume $P \models \varphi$. Then

$$\exists P' \in R\Sigma(P \xrightarrow{\alpha} P' \wedge P' \models \varphi')$$

Obeying condition 1), we have that $\beta \notin \chi(P)$ since P has to be $\chi\xi$-disjoint from Q and $\beta \in \xi(Q)$. Hence

$$red(P[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P'[\alpha \rightsquigarrow Q])$$

by the application of assertion 1) of Lemma 2.30. On the other hand, condition 2) ensures that $\alpha \in \chi(P)$ since $\alpha \in \xi(\varphi)$. Hence, we can apply assertion 3) of Lemma 2.30 and likewise obtain

$$red(P[\alpha \rightsquigarrow Q]) \xrightarrow{\beta} red(P'[\alpha \rightsquigarrow Q]).$$

Further we have

$$red(P'[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed(\varphi'[\alpha \rightsquigarrow Q])$$

by the induction hypothesis, that is,

$$red(P[\alpha \rightsquigarrow Q]) \models \langle\beta\rangle\mathcal{R}ed(\varphi'[\alpha \rightsquigarrow Q])$$

by Definition 4.22. We conclude

$$red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed((\langle\alpha\rangle\varphi')[\alpha \rightsquigarrow Q])$$

by Definition 2.11 and Definition 4.5.

$Q \in \{(Q_1; Q_2), (Q_1 + Q_2), Q_1[\gamma \rightsquigarrow Q_2]\}$. These cases are proved in analogy to the proof of those steps in Theorem 4.38 and Theorem 4.36. Here of course, we only prove the implication from the left hand side to the right hand side. Please note that exploiting condition 1), the induction hypothesis is applicable since $\chi\xi$-disjointness of P from Q_1 and Q_2 and $\alpha \notin \chi(P)$ implies that $P[\alpha \rightsquigarrow Q_1]$ remains $\chi\xi$-disjoint from

Q_2. In contrary to Theorem 4.38, the condition that Q is distinct is thus not necessary to carry out these induction steps. Using condition 2), the induction hypothesis is applicable since $\xi(\varphi) \subseteq \chi(P)$ implies $\xi(\varphi[\alpha \rightsquigarrow Q]) \subseteq \chi(P[\alpha \rightsquigarrow Q])$.

The proof of Theorem 4.50 is then carried out in analogy to the proof of Theorem 4.42 (note that we have $P \in UGR\Sigma$ iff $red(P) \in UG\Sigma$). ■

Theorem 4.51 (Debugging systems w.r.t. $R\mu\mathcal{L}_{[.]}$-properties)

Let $P \in UGR\Sigma$ and $Q \in R\Delta$ be process terms. Let $\varphi \in CGR\mu\mathcal{L}_{[.]}$ be a formula. Then we have the following:

1) If P is $\chi\xi$-disjoint from Q and $\alpha \notin \chi(P)$ or

2) if $\xi(\varphi) \subseteq \chi(P)$ then

$$P \models \varphi \Leftarrow P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q].$$

Proof: In analogy to the proof of Theorem 4.50. The induction step where $\varphi = [\gamma]\varphi'$ is proved by a contrapositive argument: Assuming that $P \not\models \varphi$ we have that $\exists P'(P \xrightarrow{\gamma} P' \wedge P' \not\models \varphi')$. By similar reasoning as in Theorem 4.50, we can easily derive that $red(P[\alpha \rightsquigarrow Q]) \not\models \mathcal{R}ed(P[\alpha \rightsquigarrow Q])$ for the two cases $\gamma \neq \alpha$ and $\gamma = \alpha$. ■

As an application of the Theorems given above we consider a ("lock-step") solution of a two process mutual exclusion problem:

Example 4.52
The process terms P_α and P_β are given by □

$$fix(x = (((\alpha_1; (\alpha; \alpha_2)) + (\beta_1; (\beta; \beta_2))); x))$$

and

$$fix(y = (((\beta_1; (\beta; \beta_2)) + (\alpha_1; (\alpha; \alpha_2))); y))$$

respectively. It is easy to see that the process $ME_{\alpha\beta} = (P_\alpha \|_{\{\alpha_i,\beta_i,\alpha,\beta\}} P_\beta)$ where $i = 1, 2$ enters the (abstract) critical sections α and β in mutual exclusion. For $A \subseteq \mathcal{A}$ we let $\diamond_A\varphi$ abbreviate the formula $\bigvee_{\alpha\in A}\langle\alpha\rangle\varphi$ and $\Box_A\varphi$ abbreviate the formula $\bigwedge_{\alpha\in A}[\alpha]\varphi$. We have that

$$ME_{\alpha\beta} \models \varphi = \nu Y.(\mu Z.(\langle\alpha\rangle\top \vee \diamond_{\xi(ME_{\alpha\beta})}Z) \wedge \diamond_{\xi(ME_{\alpha\beta})}Y),$$

that is, there exists a computation sequence of $ME_{\alpha\beta}$ along which it is always possible to reach a state where the performance α can be executed. Now let $Q = (\gamma_1 + \gamma_2)$. Then we have that

$$ME_{\alpha\beta}[\alpha \rightsquigarrow Q][\beta \rightsquigarrow Q] \models \phi[\alpha \rightsquigarrow Q][\beta \rightsquigarrow Q]$$

via Theorem 4.50 which says that $ME_{\alpha\beta}[\alpha \rightsquigarrow Q][\beta \rightsquigarrow Q]$ can execute a computation sequence along which it is always possible to reach a state where all performances of Q can be executed. This holds, though another performance (β in our case) is also refined by Q. $\square$

Please note that Theorem 4.50 and Theorem 4.51 can both be established without the need to consider restrictions of alphabet-disjointness (and distinctness).

4.3.1 Discussion

Amongst the applications of Theorem 4.42 to the verification of reactive systems, the concept of a priori-verification might be the most interesting one. There, every property φ of a system P which can be expressed in $R\mu\mathcal{L}$ "carries over" (in its refined form $\varphi[\alpha \rightsquigarrow Q]$) to the refined system $P[\alpha \rightsquigarrow Q]$, in particular this holds for all safety and liveness properties. Strong safety properties (which involve the box modality $[\alpha]$) might not be carried over (in the above sense) when dropping the restriction of alphabet disjointness. Theorem 4.50 shows however, that it is still possible to verify systems 'a priori' with respect to properties expressible in the logic $R\mu\mathcal{L}_{\langle\cdot\rangle}$, without the need to ensure alphabet disjointness of the considered process terms and formulas. In essence, these properties are "existential" properties, that is, weak safety and liveness properties (according to [192]).

In its contrapositive form, Theorem 4.51 can be used to 'debug' a (concrete) reactive system by means of debugging an abstract system (where the abstraction is based on syntactic action refinement between those systems): If $P \not\models \varphi \in R\mu\mathcal{L}_{[\cdot]}$ then $P[\alpha \rightsquigarrow Q] \not\models \varphi[\alpha \rightsquigarrow Q]$. In the case we cannot prove $P \models \varphi$, no information about satisfaction of $\varphi[\alpha \rightsquigarrow Q]$ by $P[\alpha \rightsquigarrow Q]$ can be inferred. However, Theorem 4.51 might also be used to support model checking techniques for systems that otherwise would remain unfeasable due to the size of their state spaces: If P is such a system then no information at all about satisfaction with respect to any property φ can be established by means of model checking techniques. In particular we could not show $P \not\models \varphi$. However, if we could establish appropriate abstractions P_s and φ_s, i.e. $P = P_s[\alpha_1 \rightsquigarrow Q_1]\dots[\alpha_n \rightsquigarrow Q_n]$ and $\varphi = \varphi_s[\alpha_1 \rightsquigarrow Q_1]\dots[\alpha_n \rightsquigarrow Q_n]$ then P_s might well become manageable by a model checker since the state space of P_s might be exponentially smaller then the state space of P due to the well known state explosion problem[23]. Then we could apply the model checker to prove $P_s \not\models \varphi_s$ and conclude $P \not\models \varphi$ via Theorem 4.51. Various interesting properties can be expressed in $R\mu\mathcal{L}_{[\cdot]}$ (and therefore be used in the debug-procedure described above), in particular strong safety properties of a system P like, that is, $\nu Z.(\varphi \wedge \Box_{\xi(P)} Z)$ meaning "φ holds in every state of P".

4.4 SSAR for the Generalized Modal Mu-Calculus

In this section we present the generalized Modal Mu-Calculus. The syntax of this logic is similar to the standard Modal Mu-Calculus except for the generalizations of the modalities. In order to introduce the new production rules we next define an extension of the language Δ (see Definition 2.5): Let Δ_0 be the language generated by the grammar for Δ with the additional rule $Q ::= 0$.

Definition 4.53 (Generalized Modal Mu-Calculi $\mu\mathcal{L}_g$ and $R\mu\mathcal{L}_g$)
The generalized Modal Mu-Calculus $\mu\mathcal{L}_g$ is generated by the grammar for the (standard) Modal Mu-Calculus (see Definition 4.1) in which the rules $\varphi ::= [\alpha]\varphi$ and $\varphi ::= \langle\alpha\rangle\varphi$ are replaced by the rules $\varphi ::= [E]\varphi$ and $\varphi ::= \langle E\rangle\varphi$ (where $E \in \Delta_0$). Let $R\mu\mathcal{L}_g$ be the language generated by the above grammar with the additional rule $\varphi ::= \varphi[\alpha \rightsquigarrow Q]$ where $Q \in R\Delta$. $\Box$

For $E \in \Delta_0$ we let $\triangle_E \in \{\langle E\rangle, [E]\}$ and adapt the length of formulas (as defined on page 81) by $|\triangle_E\varphi| := |\varphi| + |E|$ where $\varphi \in \mu\mathcal{L}_g$. A formula $\varphi \in R\mu\mathcal{L}_g$ is called *simple* iff for all modalities $\langle E\rangle$ and $[E]$ that occur in φ we have that $E \in \Delta$. A $R\mu\mathcal{L}_g$-formula φ is called *guarded* iff every occurrence of a variable Z in φ lies in the scope of a modality $[E]$ or $\langle E\rangle$ where $E \notin \surd$. The notions of closedness is as for the standard

[23]A linear reduction of the number of performances in a term $P \in R\Sigma$ can entail an exponential reduction of the number of reachable states of $\mathcal{T}(P)$. Please note that model checking algorithms are based on the investigation of the involved system state spaces, regardless whether those spaces are represented explicitly or implicitly (e.g. using *BDDs* [31]).

Modal Mu-Calculus (see page 79). The notion of ξ-disjointness (see Definition 4.4) is as for the standard Modal Mu-Calculus where $\xi(\triangle_E\varphi) := \xi(E) \cup \xi(\varphi)$.

We now introduce a logical substitution operation for the generalized Modal Mu-Calculus $\mu\mathcal{L}_g$ by which we are able to define the reduction function for $R\mu\mathcal{L}_g$-formulas.

Definition 4.54 (Logical substitution for $\mu\mathcal{L}_g$)
Let $Q \in \Delta$ and $E \in \Delta_0$ be process expressions and $\varphi, \psi \in \mu\mathcal{L}_g$ be formulas. The operation of (generalized) logical substitution*, $(\varphi)\{\alpha \rightsquigarrow Q\}_g$ is defined as follows:*

$$(*)\{\alpha \rightsquigarrow Q\}_g := * \quad \textit{if } * \in \{\top, \bot\} \cup Var\ , \qquad (\sigma Z.\varphi)\{\alpha \rightsquigarrow Q\}_g := \sigma Z.(\varphi)\{\alpha \rightsquigarrow Q\}_g$$

$$((\varphi \odot \psi))\{\alpha \rightsquigarrow Q\}_g := ((\varphi)\{\alpha \rightsquigarrow Q\}_g \odot (\psi)\{\alpha \rightsquigarrow Q\}_g) \quad \textit{if } \odot \in \{\wedge, \vee\}$$

$$([E]\varphi)\{\alpha \rightsquigarrow Q\}_g := [E\{Q/\alpha\}](\varphi)\{\alpha \rightsquigarrow Q\}_g$$

$$(\langle E\rangle\varphi)\{\alpha \rightsquigarrow Q\}_g := \langle E\{Q/\alpha\}\rangle(\varphi)\{\alpha \rightsquigarrow Q\}_g$$ □

Definition 4.55 (Logical reduction function for $R\mu\mathcal{L}_g$)
Let $Q \in R\Delta$ and $E \in \Delta_0$ be process expressions and $\varphi, \psi \in R\mu\mathcal{L}_g$ be formulas. We define the (generalized) logical reduction function $\mathcal{R}ed_g : R\mu\mathcal{L}_g \to \mu\mathcal{L}_g$ *as follows:*

$$\mathcal{R}ed_g(*) := * \quad \textit{if } * \in \{\top, \bot\} \cup Var\ , \quad \mathcal{R}ed_g(\varphi[\alpha \rightsquigarrow Q]) := (\mathcal{R}ed_g(\varphi))\{\alpha \rightsquigarrow red(Q)\}_g$$

$$\mathcal{R}ed_g((\varphi \odot \psi)) := (\mathcal{R}ed_g(\varphi) \odot \mathcal{R}ed_g(\psi)) \quad \textit{if } \odot \in \{\wedge, \vee\}$$

$$\mathcal{R}ed_g([E]\varphi) := [E]\mathcal{R}ed_g(\varphi)\ , \quad \mathcal{R}ed_g(\langle E\rangle\varphi) := \langle E\rangle\mathcal{R}ed_g(\varphi)$$

$$\mathcal{R}ed_g(\sigma Z.\varphi) := \sigma Z.\mathcal{R}ed_g(\varphi)$$ □

The distinguishing feature of the above presented reduction function is, that

$$|\mathcal{R}ed_g(\varphi[\alpha \rightsquigarrow Q])| = O(|\varphi| * |Q|),$$

provided we have $\varphi \in \mu\mathcal{L}_g$ and $Q \in \Delta$. It is thus possible to derive (non hierarchical) $\mu\mathcal{L}_g$-formulas from the according hierarchical specifications in polynomial time, as opposed to hierarchical formulas of the Modal Mu-Calculus (see Section 4.2.2).

Lemma 4.56

Let $\varphi \in R\mu\mathcal{L}_g$ and $Q \in R\Delta$. Then $\mathcal{R}ed_g((\mathcal{R}ed_g(\varphi))[\alpha \rightsquigarrow Q]) = \mathcal{R}ed_g(\varphi[\alpha \rightsquigarrow Q])$.

Proof: Follows from Definition 4.55 and the fact that $\mathcal{R}ed_g(\varphi) \in \mu\mathcal{L}_g$. ■

We now define the satisfaction relation for the generalized Modal Mu-Calculus.

Definition 4.57 (Satisfaction for $R\mu\mathcal{L}_g$)

Let $P \in R\Sigma$, $Q \in R\Delta$, $E \in \Delta_0$, $\varphi, \psi \in R\mu\mathcal{L}_g$ and $Z \in Var$.

$$
\begin{array}{lll}
P \models^g_\vartheta \top \quad , & P \not\models^g_\vartheta \bot \quad , & P \models^g_\vartheta Z \textit{ iff } P \in \vartheta(Z) \\
P \models^g_\vartheta (\varphi \wedge \psi) & \textit{iff } P \models^g_\vartheta \varphi \textit{ and } P \models^g_\vartheta \psi & \\
P \models^g_\vartheta (\varphi \vee \psi) & \textit{iff } P \models^g_\vartheta \varphi \textit{ or } P \models^g_\vartheta \psi & \\
P \models^g_\vartheta [E]\varphi & \textit{iff } E \in \surd \textit{ implies } P \models^g_\vartheta \varphi \textit{ and } E \xrightarrow{\alpha} E' \textit{ implies} & \\
 & P \in \{F \in R\Sigma \mid \forall F'(F \xrightarrow{\alpha} F' \Rightarrow F' \models^g_\vartheta [E']\varphi\} & \\
P \models^g_\vartheta \langle E\rangle\varphi & \textit{iff } E \in \surd \textit{ implies } P \models^g_\vartheta \varphi \textit{ and } E \xrightarrow{\alpha} E' \textit{ implies} & \\
 & P \in \{F \in R\Sigma \mid \exists F'(F \xrightarrow{\alpha} F' \textit{ and } F' \models^g_\vartheta \langle E'\rangle\varphi\} & \\
P \models^g_\vartheta \mu Z.\varphi & \textit{iff } P \in \bigcap\left\{\mathcal{E} \subseteq R\Sigma | \{E \in R\Sigma | E \models^g_{\vartheta[\mathcal{E}/Z]} \varphi\} \subseteq \mathcal{E}\right\} & \\
P \models^g_\vartheta \nu Z.\varphi & \textit{iff } P \in \bigcup\left\{\mathcal{E} \subseteq R\Sigma | \mathcal{E} \subseteq \{E \in R\Sigma | E \models^g_{\vartheta[\mathcal{E}/Z]} \varphi\}\right\} & \\
P \models^g_\vartheta \varphi[\alpha \rightsquigarrow Q] & \textit{iff } P \models^g_\vartheta \mathcal{R}ed_g(\varphi[\alpha \rightsquigarrow Q]) &
\end{array}
$$

□

Assertions about subprocesses of systems can be expressed in $\mu\mathcal{L}_g$ in a more intuitive and concise way than in $\mu\mathcal{L}$.

Example 4.58

Let $\phi = \mu Z.(\langle E\rangle\top \vee [\alpha]Z)$. A system P satisfies ϕ if any "α-path" of P eventually leads to a state (a subsystem of P) where P can execute any computation step executable by the system E. □

The semantics of $\mu\mathcal{L}_g$ coincides with the semantics of the Modal Mu-Calculus if we restrict the generalized modalities $[E]$ and $\langle E\rangle$ such that $E \in Act$. In order to relate the logics $\mu\mathcal{L}$ and $\mu\mathcal{L}_g$ with each other we need the following results. In what follows, $\triangle_E$ means either $\langle E\rangle$ or $[E]$ throughout the statements.

Lemma 4.59

Let $P \in R\Sigma$ and $E_1, E_2 \in \Delta_0$ be process expressions and $\varphi \in R\mu\mathcal{L}_g$ be a formula. Then

1) $P \models^g_\vartheta \triangle_{(E_1+E_2)}\varphi$ and $E_1 \notin \surd$ imply $P \models^g_\vartheta \triangle_{E_1}\varphi$,

2) $P \models^g_\vartheta \triangle_{E_1}\varphi$ and $E_2 \in \surd$ imply $P \models^g_\vartheta \triangle_{(E_1+E_2)}\varphi$,

3) $P \models^g_\vartheta \triangle_{E_1}\varphi$ and $P \models^g_\vartheta \triangle_{E_2}\varphi$ imply $P \models^g_\vartheta \triangle_{(E_1+E_2)}\varphi$.

Proof:
1) Assume $P \models^g_\vartheta \langle(E_1 + E_2)\rangle\varphi$. W.l.o.g. let $E_1 \xrightarrow{\alpha} E_1'$. Then

$$\exists P'(P \xrightarrow{\alpha} P' \text{ and } P' \models^g_\vartheta \langle E_1'\rangle\varphi).$$

Hence $P \models^g_\vartheta \langle E_1\rangle\varphi$.
Now assume $P \models^g_\vartheta [(E_1 + E_2)]\varphi$ and $P \not\models^g_\vartheta [E_1]\varphi$. The latter implies

$$\exists E_1'(E_1 \xrightarrow{\alpha} E_1') \text{ and } \exists P'(P \xrightarrow{\alpha} P' \text{ and } P' \not\models^g_\vartheta [E_1']\varphi)$$

by Definition 4.57 (note that $E_1 \notin \surd$). But $E_1 \xrightarrow{\alpha} E_1'$ implies $(E_1 + E_2) \xrightarrow{\alpha} E_1'$ by the operational semantics. It follows $P \not\models^g_\vartheta [(E_1 + E_2)]\varphi$, contradiction.
2) By the operational semantics we have that $E_2 \in \surd$ implies $((E_1 + E_2) \xrightarrow{\alpha} E'$ iff $E_1 \xrightarrow{\alpha} E')$. Assertion 2) then follows directly from Definition 4.57.
3) Assume $P \models^g_\vartheta \langle E_1\rangle\varphi$ and $P \models^g_\vartheta \langle E_2\rangle\varphi$. Case i): $E_2 \in \surd$. The claim follows directly from assertion 2). Case ii): $E_2 \notin \surd$. For $E_1 \in \surd$, the claim follows by assertion 2). Now let $E_1 \notin \surd$. Assume $E_1 \xrightarrow{\alpha} E_1'$. Then $\exists P'(P \xrightarrow{\alpha} P'$ and $P' \models^g_\vartheta \langle E_1'\rangle\varphi)$ by the premise. Further assume $E_2 \xrightarrow{\beta} E_2'$. Then $\exists P''(P \xrightarrow{\beta} P''$ and $P'' \models^g_\vartheta \langle E_2'\rangle\varphi)$ by the premise. By the operational semantics and Definition 4.57 follows $P \models^g_\vartheta \langle(E_1+E_2)\rangle\varphi$. The proof for the box modality proceeds analogously. ■

From Lemma 4.59 easily follows

Corollary 4.60
Let $P \in R\Sigma$ and $E_1, E_2 \in \Delta_0$ be process expressions and $\varphi \in R\mu\mathcal{L}_g$ be a formula. If $(E_1 \in \surd$ iff $E_2 \in \surd)$ then $((P \models^g_\vartheta \triangle_{E_1}\varphi$ and $P \models^g_\vartheta \triangle_{E_2}\varphi)$ iff $P \models^g_\vartheta \triangle_{(E_1+E_2)}\varphi)$.

Proof: Case 1: $E_1, E_2 \notin \surd$. Then the claim follows from assertion 1) and assertion 3) of from Lemma 4.59. Case 2: $E_1, E_2 \in \surd$. Then the claim follows directly from Definition 4.57. ■

Lemma 4.61
Let $P \in R\Sigma$ and $E_1, E_2 \in \Delta_0$ be process expressions and $\varphi \in R\mu\mathcal{L}_g$ be a formula. If $E_1 \in \surd$, then $P \models^g_\vartheta \triangle_{E_1}\triangle_{E_2}\varphi$ iff $P \models^g_\vartheta \triangle_{(E_1;E_2)}\varphi$.

Proof: Follows directly from Definition 2.21 and Definition 4.57. ■

Lemma 4.62
Let $P \in R\Sigma$ and $E_{11}, E_{12}, E_2 \in \Delta_0$ be process expressions and $\varphi \in R\mu\mathcal{L}_g$ be a formula. Then

1) $P \models^g_\vartheta \triangle_{((E_{11}+E_{12});E_2)}\varphi$ and $E_{11} \notin \surd$ imply $P \models^g_\vartheta \triangle_{(E_{11};E_2)}\varphi$,

2) $P \models^g_\vartheta \triangle_{(E_{11};E_2)}\varphi$ and $P \models^g_\vartheta \triangle_{(E_{12};E_2)}\varphi$ imply $P \models^g_\vartheta \triangle_{((E_{11}+E_{12});E_2)}\varphi$,

3) $P \models^g_\vartheta \triangle_{(E_{11};E_2)}\varphi$ and $E_{12} \in \surd$ imply $P \models^g_\vartheta \triangle_{((E_{11}+E_{12});E_2)}\varphi$.

Proof: Follows from Definition 2.21, Definition 4.57, Corollary 4.60 and Lemma 4.61. We only show assertion 2) for the diamond-modality:

Case 1: $E_{11}, E_{12} \in \surd$. Then the claim follows from Corollary 4.60 and Lemma 4.61.

Case 2: $E_{11} \in \surd$ and $E_{12} \notin \surd$. Assume the premise holds. Suppose we have $((E_{11} + E_{12}); E_2) \xrightarrow{\alpha} E'$. Since $E_{11} \in \surd$, it follows by Definition 2.21 that $E' = (E'_{12}; E_2)$ where $E_{12} \xrightarrow{\alpha} E'_{12}$. Hence $(E_{12}; E_2) \xrightarrow{\alpha} (E'_{12}; E_2)$. By Definition 4.57 follows $\exists P'(P \xrightarrow{\alpha} P'$ and $P' \models^g_\vartheta \langle(E'_{12}; E_2)\rangle\varphi)$. We conclude that $P \models^g_\vartheta \langle((E_{11} + E_{12}); E_2)\rangle\varphi$.

Case 3: $E_{11}, E_{12} \notin \surd$. Assume the premise holds. We have $((E_{11} + E_{12}); E_2) \xrightarrow{\alpha} E'$. W.l.o.g., suppose $E_{11} \xrightarrow{\alpha} E'_{11}$ and $E' = (E'_{11}; E_2)$. Hence $\exists P'(P \xrightarrow{\alpha} P'$ and $P' \models^g_\vartheta \langle(E'_{11}; E_2)\rangle\varphi)$. We conclude that $P \models^g_\vartheta \langle((E_{11} + E_{12}); E_2)\rangle\varphi$ by Definition 4.57. ■

Lemma 4.63
Let $P \in R\Sigma$ and $E_1, E_2 \in \Delta_0$ be process expressions and $\varphi \in R\mu\mathcal{L}_g$ be a formula. Then $P \models^g_\vartheta \triangle_{E_1}\triangle_{E_2}\varphi$ iff $P \models^g_\vartheta \triangle_{(E_1;E_2)}\varphi$.

Proof: Define the function $dt : \Delta_0 \to \mathbb{N}_0$ by $dt(0) = 0$ and $dt(\alpha) = 1$ and $dt((E_1\ op\ E_2)) = dt(E_1) + dt(E_2) + 1$ where $op \in \{;, +\}$. The proof is by induction on $dt((E_1; E_2))$ within a case discrimination on the structure of $E_1 \in \Delta_0$. We only show the proof for the diamond-modality. The proof for the box-modality proceeds analogously.

$\underline{E_1 = 0}$: Follows from Lemma 4.61.

<u>$E_1 = \alpha$:</u> Then $P \models^g_\vartheta \langle (E_1; E_2) \rangle \varphi$

$$\text{iff } \exists P'(P \overset{\alpha}{\to} P' \text{ and } P' \models^g_\vartheta \langle (0; E_2) \rangle \varphi) \quad \text{(By Definition 4.57)}$$

$$\text{iff } \exists P'(P \overset{\alpha}{\to} P' \text{ and } P' \models^g_\vartheta \langle 0 \rangle \langle E_2 \rangle \varphi) \quad \text{(By the induction hypothesis)}$$

$$\text{iff } P \models^g_\vartheta \langle E_1 \rangle \langle E_2 \rangle \varphi \quad \text{(By Definition 4.57)}$$

<u>$E_1 = (E_{11} + E_{12})$:</u>

Case 1: $E_{11}, E_{12} \in \surd$. Then the claim follows from Lemma 4.61.

Case 2: $E_{11} \notin \surd$ and $E_{12} \in \surd$ (the case $E_{11} \in \surd$ and $E_{12} \notin \surd$ is proved analogously). Then $P \models^g_\vartheta \langle ((E_{11} + E_{12}); E_2) \rangle \varphi$

$$\text{iff } P \models^g_\vartheta \langle (E_{11}; E_2) \rangle \varphi \quad \text{(By assertion 1) and 3) of Lemma 4.62)}$$

$$\text{iff } P \models^g_\vartheta \langle E_{11} \rangle \langle E_2 \rangle \varphi$$

(By the induction hypothesis: $dt(((E_{11}+E_{12}); E_2)) = dt(E_{11}) + dt(E_{12}) + 1 + dt(E_2) + 1 > dt(E_{11}) + dt(E_2) + 1 = dt((E_{11}; E_2)))$

$$\text{iff } P \models^g_\vartheta \langle (E_{11} + E_{12}) \rangle \langle E_2 \rangle \varphi \quad \text{(By assertion 1) and 2) of Lemma 4.59)}$$

Case 3: $E_{11}, E_{12} \notin \surd$. Then $P \models^g_\vartheta \langle ((E_{11} + E_{12}); E_2) \rangle \varphi$

$$\text{iff } P \models^g_\vartheta \langle (E_{11}; E_2) \rangle \varphi \text{ and } P \models^g_\vartheta \langle (E_{12}; E_2) \rangle \varphi$$

(By assertion 1) and 2) of Lemma 4.62)

$$\text{iff } P \models^g_\vartheta \langle E_{11} \rangle \langle E_2 \rangle \varphi \text{ and } P \models^g_\vartheta \langle E_{12} \rangle \langle E_2 \rangle \varphi$$

(By the induction hypothesis: $dt(((E_{11}+E_{12}); E_2)) = dt(E_{11}) + dt(E_{12}) + 1 + dt(E_2) + 1 > dt(E_{1i}) + dt(E_2) + 1 = dt((E_{1i}; E_2))$ for $i = 1, 2$)

$$\text{iff } P \models^g_\vartheta \langle (E_{11} + E_{12}) \rangle \langle E_2 \rangle \varphi \quad \text{(By Corollary 4.59)}$$

<u>$E_1 = (E_{11}; E_{12})$:</u>

Case 1: $E_1 \in \surd$. Then the claim follows from Lemma 4.61.

Case 2: $E_1 \notin \surd$. Assume $P \models^g_\vartheta \langle((E_{11}; E_{12}); E_2)\rangle\varphi$. Suppose $(E_{11}; E_{12}) \xrightarrow{\alpha} E'$. This is equivalent to $((E_{11}; E_{12}); E_2) \xrightarrow{\alpha} (E'; E_2)$ by Definition 2.21 since $E_1 \notin \surd$. Hence $\exists P'(P \xrightarrow{\alpha} P'$ and $P' \models^g_\vartheta \langle(E'; E_2)\rangle\varphi)$ by Definition 4.57. By the induction hypothesis, this is equivalent to the assertion $\exists P'(P \xrightarrow{\alpha} P'$ and $P' \models^g_\vartheta \langle E'\rangle\langle E_2\rangle\varphi)$ since $F \xrightarrow{\alpha} F' \Rightarrow dt(F') < dt(F)$ for any $F \in \Delta_0$. By Definition 4.57 follows $P \models^g_\vartheta \langle(E_{11}; E_{12})\rangle\langle E_2\rangle\varphi$. ■

It should be noted that for $E_i \notin \surd$ (i=1,2) we have that $P \models^g_\vartheta \langle(E_1 + E_2)\rangle\varphi$ iff $P \models^g_\vartheta (\langle E_1\rangle\varphi \wedge \langle E_2\rangle\varphi)$ which is in contrast to $P \models^{PDL} \langle(a + b)\rangle\varphi$ iff $P \models^{PDL} (\langle a\rangle\varphi \vee \langle b\rangle\varphi)$ in the logic PDL. Also note, that the generalized modalities $[E]$ and $\langle E\rangle$ are not necessarily dual to each other for $E \notin \surd$.

A formula φ in which no terminated process term occurs (this holds for all simple formulas) clearly satisfies the condition that $E_1 \in \surd$ iff $E_2 \in \surd$ for all modalities $\triangle_{(E_1+E_2)}$ that occur in φ. We will thus confine our consideration to simple formulas in what follows.

Lemma 4.64
Let $\varphi \in R\mu\mathcal{L}_g$ be simple and $P \in R\Sigma$. Then $P \models^g_\vartheta \varphi$ iff $P \models^g_\vartheta \mathcal{R}ed_g(\varphi)$.

Proof: The proof is by induction on $|\varphi|$ within a case discrimination on the structure of simple formulas $\varphi \in R\mu\mathcal{L}_g$ using Corollary 4.60 and Lemma 4.63. ■

We can now give a translation t that takes simple $\mu\mathcal{L}_g$-formulas to $\mu\mathcal{L}$-formulas.

Definition 4.65 (Translation of simple $\mu\mathcal{L}_g$-formulas into $\mu\mathcal{L}$-formulas)
The translation t of simple $\mu\mathcal{L}_g$-formulas into $\mu\mathcal{L}$-formulas is given by

$$t(*) := * \textit{ for } * \in \{\top, \bot\} \cup Var\ , \qquad t(\sigma Z.\varphi) := \sigma Z.t(\varphi)$$

$$t((\varphi_1 \odot \varphi_2)) := (t(\varphi_1) \odot t(\varphi_2)) \textit{ for } \odot \in \{\wedge, \vee\}$$

$$t(\triangle_E\varphi) := \begin{cases} \triangle_E t(\varphi) & \textit{if } E \in Act \\ \\ (\triangle_\gamma t(\varphi))\{\gamma \rightsquigarrow E\} \textit{ where } \gamma \notin \xi(t(\varphi)) & \textit{if } E \notin Act \end{cases}$$

□

Remark 4.66

Let $\varphi \in \mu\mathcal{L}_g$ be a simple formula. Then clearly $t(\varphi) \in \mu\mathcal{L}$. $\square$

Refinements in the two logics are related via the following lemma.

Lemma 4.67

Let $\varphi \in \mu\mathcal{L}_g$ be a simple formula and $Q \in R\Delta$ be a process term. Then we have $\mathcal{R}ed((t(\varphi))[\alpha \rightsquigarrow Q]) = t(\mathcal{R}ed_g(\varphi[\alpha \rightsquigarrow Q]))$.

Proof:

<u>Claim:</u> For $\varphi \in \mu\mathcal{L}_g$ and $Q \in \Delta$ we have

$$(t(\varphi))\{\alpha \rightsquigarrow Q\} = t((\varphi)\{\alpha \rightsquigarrow Q\}_g).$$

Assume the claim holds. Then

$$\mathcal{R}ed((t(\varphi))[\alpha \rightsquigarrow Q])$$

$$= (\mathcal{R}ed(t(\varphi)))\{\alpha \rightsquigarrow red(Q)\} \quad \text{(Definition 4.13)}$$

$$= (t(\varphi))\{\alpha \rightsquigarrow red(Q)\} \quad (t(\varphi) \in \mu\mathcal{L} \text{ by Lemma 4.66})$$

$$= t((\varphi)\{\alpha \rightsquigarrow red(Q)\}_g) \quad \text{(by the claim above)}$$

$$= t((\mathcal{R}ed_g(\varphi))\{\alpha \rightsquigarrow red(Q)\}_g) \quad (\varphi \in \mu\mathcal{L}_g)$$

$$= t(\mathcal{R}ed_g(\varphi[\alpha \rightsquigarrow Q])) \quad \text{(Definition 4.55).}$$

It remains to prove the claim. This is done by induction on $|\varphi|$ within a case discrimination on the structure of simple formulas $\varphi \in \mu\mathcal{L}_g$. We only show the case $\varphi = [E]\varphi'$ where $E \notin \surd$. The proof for the generalized diamond modality proceeds analogously. The other induction steps follow immediately from the relevant definitions and the induction hypothesis.

$\varphi = [E]\varphi'$ and $E \notin Act$: $(t([E]\varphi'))\{\alpha \rightsquigarrow Q\}$

$$= (([\gamma]t(\varphi'))\{\gamma \rightsquigarrow E\})\{\alpha \rightsquigarrow Q\} \quad (*) \quad \text{(by Definition 4.65)}$$

where $\gamma \notin \xi(t(\varphi'))$. W.l.o.g. let $\gamma \notin \xi(Q)$ and $\gamma \neq \alpha$. Then assertion $(*)$ equals

$$([\gamma](t(\varphi'))\{\alpha \rightsquigarrow Q\})\{\gamma \rightsquigarrow E\{Q/\alpha\}\} \quad \text{(by the choice of } \gamma)$$

$$= ([\gamma]t((\varphi')\{\alpha \rightsquigarrow Q\}_g))\{\gamma \rightsquigarrow E\{Q/\alpha\}\} \quad \text{(by induction)}$$
$$= t([E\{Q/\alpha\}](\varphi')\{\alpha \rightsquigarrow Q\}_g) \quad \text{(Definition 4.65)}$$
$$= t(([E]\varphi')\{\alpha \rightsquigarrow Q\}_g) \quad \text{(Definition 4.54)}$$

■

Lemma 4.68
Let $\varphi \in \mu\mathcal{L}_g$ be a simple formula and $E_1, E_2 \in \Delta$. Then

1) $t(\triangle_{(E_1;E_2)}\varphi) = t(\triangle_{E_1}\triangle_{E_2}\varphi)$,

2) $t(\triangle_{(E_1+E_2)}\varphi) = t((\triangle_{E_1}\varphi \wedge \triangle_{E_2}\varphi))$.

Proof: Follows from Definition 4.54, Definition 4.65, and Lemma 4.67. ■

The following lemma relates the satisfaction relations of the logics $\mu\mathcal{L}$ and $\mu\mathcal{L}_g$.

Lemma 4.69
Let $P \in R\Sigma$ be a process term and $\varphi \in \mu\mathcal{L}_g$ be a simple formula. Then $P \models^g_\vartheta \varphi$ iff $P \models_\vartheta t(\varphi)$.

Proof: The proof is by induction on $|\varphi|$ involving a case discrimination on the structure of simple formulas $\varphi \in \mu\mathcal{L}_g$ using Lemma 4.59, Corollary 4.60, Lemma 4.63 and Lemma 4.67.

$\underline{\varphi \in \{\top, \bot\} \cup Var}$: Trivial.

$\underline{\varphi = (\varphi_1 \odot \varphi_2) \text{ where } \odot \in \{\vee, \wedge\}}$: Immediately from the satisfaction relation of the Modal Mu-Calculus, Definition 4.57, Definition 4.65 and the induction hypothesis.

$\underline{\varphi = \triangle_E\varphi'}$: Case discrimination on the structure of $E \in \Delta$. Note that $E = 0$ cannot occur since φ is simple.
$E = \alpha$: We only show the case $\varphi = [\alpha]\varphi'$. The case where $\varphi = \langle\alpha\rangle\varphi'$ is proved analogously. Suppose $P \models^g_\vartheta [\alpha]\varphi$. Then

$$\forall P'(P \xrightarrow{\alpha} P' \Rightarrow P' \models^g_\vartheta [0]\varphi') \quad \text{(By Definition 4.57)}$$

$$\text{iff } \forall P'(P \xrightarrow{\alpha} P' \Rightarrow P' \models^g_\vartheta \varphi') \quad \text{(By Definition 4.57)}$$

$$\text{iff } \forall P'(P \overset{\alpha}{\to} P' \Rightarrow P' \models_\vartheta t(\varphi')) \quad \text{(By the induction hypothesis)}$$

$$\text{iff } P \models_\vartheta [\alpha]t(\varphi) \quad \text{(By Definition 4.57)}$$

$$\text{iff } P \models_\vartheta t([\alpha](\varphi)) \quad \text{(By Definition 4.65)}$$

$E = (E_1 + E_2)$: Then $P \models^g_\vartheta \triangle_{(E_1+E_2)}\varphi$

$$\text{iff } P \models^g_\vartheta \triangle_{E_1}\varphi \text{ and } P \models^g_\vartheta \triangle_{E_2}\varphi \quad \text{(By Lemma 4.59)}$$

$$\text{iff } P \models_\vartheta t(\triangle_{E_1}\varphi) \text{ and } P \models_\vartheta t(\triangle_{E_2}\varphi) \quad \text{(By the induction hypothesis)}$$

$$\text{iff } P \models_\vartheta (t(\triangle_{E_1}\varphi) \wedge t(\triangle_{E_2}\varphi)) \quad \text{(By Definition 4.57)}$$

$$\text{iff } P \models_\vartheta t((\triangle_{E_1}\varphi \wedge \triangle_{E_2}\varphi)) \quad \text{(By Definition 4.65)}$$

$$\text{iff } P \models_\vartheta t(\triangle_{(E_1+E_2)}\varphi) \quad \text{(By Lemma 4.68)}$$

$E = (E_1; E_2)$: Then $P \models^g_\vartheta \triangle_{(E_1;E_2)}\varphi$

$$\text{iff } P \models^g_\vartheta \triangle_{E_1}\triangle_{E_2}\varphi \quad \text{(By Lemma 4.63)}$$

$$\text{iff } P \models_\vartheta t(\triangle_{E_1}\triangle_{E_2}\varphi) \quad \text{(By the induction hypothesis)}$$

$$\text{iff } P \models_\vartheta t(\triangle_{(E_1;E_2)}\varphi) \quad \text{(By Lemma 4.68)}$$

$\underline{\varphi = \mu Z.\varphi'}$: Then we have $P \models^g_\vartheta \mu Z.\varphi'$

$$\text{iff } P \in \bigcap \left\{ \mathcal{E} \subseteq R\Sigma | \{E \in R\Sigma | E \models^g_{\vartheta[\mathcal{E}/Z]} \varphi'\} \subseteq \mathcal{E} \right\} \quad \text{(Definition 4.57)}$$

$$\text{iff } P \in \bigcap \left\{ \mathcal{E} \subseteq R\Sigma | \{E \in R\Sigma | E \models_{\vartheta[\mathcal{E}/Z]} t(\varphi')\} \subseteq \mathcal{E} \right\} \quad \text{(by induction)}$$

$$\text{iff } P \models_\vartheta \mu Z.t(\varphi') \quad \text{(By the satisfaction relation of } R\mu\mathcal{L})$$

$$\text{iff } P \models_\vartheta t(\mu Z.\varphi') \quad \text{(Definition 4.65)}$$

$\underline{\varphi = \nu Z.\varphi'}$: In analogy to the above step. ■

We can now give the refinement theorem for (simple) $R\mu\mathcal{L}_g$-formulas.

Theorem 4.70

Let $P \in GR\Sigma$ be a process term, $\varphi \in CGR\mu\mathcal{L}_g$ be a simple formula and $Q \in R\Delta$ be a distinct process term, such that P and φ are alphabet-disjoint from Q. Then we have $P \models^g \varphi \Leftrightarrow P[\alpha \rightsquigarrow Q] \models^g \varphi[\alpha \rightsquigarrow Q]$.

Proof:

$$P \models^g \varphi$$

$$\text{iff } P \models^g \mathcal{R}ed_g(\varphi) \quad \text{(By Lemma 4.64)}$$

$$\text{iff } P \models t(\mathcal{R}ed_g(\varphi)) \quad \text{(By Lemma 4.69)}$$

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models \mathcal{R}ed((t(\mathcal{R}ed_g(\varphi)))[\alpha \rightsquigarrow Q])$$

(Follows by Theorem 4.42 and Remark 4.66)

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models t(\mathcal{R}ed_g((\mathcal{R}ed_g(\varphi))[\alpha \rightsquigarrow Q])) \quad \text{(By Lemma 4.67)}$$

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models^g \mathcal{R}ed_g((\mathcal{R}ed_g(\varphi))[\alpha \rightsquigarrow Q]) \quad \text{(By Lemma 4.69)}$$

$$\text{iff } red(P[\alpha \rightsquigarrow Q]) \models^g \mathcal{R}ed_g(\varphi[\alpha \rightsquigarrow Q]) \quad \text{(By Lemma 4.56)}$$

$$\text{iff } P[\alpha \rightsquigarrow Q] \models^g \varphi[\alpha \rightsquigarrow Q]$$

(Follows from Lemma 4.64 and, by definition, $\mathcal{T}(P[\alpha \rightsquigarrow Q]) = \mathcal{T}(red(P[\alpha \rightsquigarrow Q]))$)

■

Example 4.71

Consider the term $P = (Q_1 \| (Q_2; \alpha))$ where we assume that $Q_1 \in \Delta$ is alphabet-disjoint from $Q_2 \in \Delta$ and Q_i $(i = 1, 2)$ is distinct. The system P satisfies the formula $\varphi = \mu Z.(\langle\alpha\rangle\top \vee [Q_2]Z)$ expressing the property that "every computation path that emerges by recursively executing Q_2 a finite number of times can also be executed by P and leads P to a state where P can execute α". Using classical model checking techniques, the whole state space of P (the size of which can be exponential in $|P|$) has to be computed. Now we have that $P = red(P_s[\beta \rightsquigarrow Q_1][\gamma \rightsquigarrow Q_2])$ where $P_s = (\beta \| (\gamma; \alpha))$ has six states. Further, $\varphi = \mathcal{R}ed_g(\varphi_s[\beta \rightsquigarrow Q_1][\gamma \rightsquigarrow Q_2])$ where $\varphi_s = \mu Z.(\langle\alpha\rangle\top \vee [\gamma]Z)$. We can now use a model checker for the (standard) Modal Mu-Calculus (see, for example, [18]) to decide whether $P_s \models^g \varphi_s$. If Q_i $(i = 1, 2)$ does not contain the action α, we can use Theorem 4.70 to decide $P \models^g \varphi$. □

4.4.1 Discussion

Using the logic $R\mu\mathcal{L}_g$ together with the reduction function $\mathcal{R}ed_g$ instead of the logic $R\mu\mathcal{L}$ solves the problem of the "exponential blow-up" that arises for $R\mu\mathcal{L}$-formula reduction discussed in Section 4.2.2: Provided we consider terms $Q \in \Delta$ and formulas $\varphi \in \mu\mathcal{L}_g$ (process terms $P \in \Sigma$), the formula $\mathcal{R}ed_g(\varphi[\alpha \rightsquigarrow Q])$ (process term $red(P[\alpha \rightsquigarrow Q])$ resp.) has size at most $O(|\varphi| * |Q|)$ ($O(|P| * |Q|)$ resp.). Computing reductions thus takes time $O(|\varphi| * |Q|)$ and $O(|P| * |Q|)$ for the reduction of formulas and for the reduction of process terms respectively. Hence, after each refinement step, implementation near process terms and low level specifications can be efficiently derived by applying the according reduction functions.

We have seen, that the generalized Modal Mu-Calculus $\mu\mathcal{L}_g$ allows to formalize properties of reactive systems in a very concise and intuitive way. Finally, $\mu\mathcal{L}_g$-formulas can be easily translated into logically equivalent $\mu\mathcal{L}$-formulas. This allows existing methods and tools for the Modal Mu-Calculus to be integrated into our framework.

4.5 Related Work

Addressing a development/re-engineering-paradigm, [105] showed that a *synchronization structure* $\mathcal{S}$ satisfies a formula φ if and only if a (semantical) refinement of $\mathcal{S}$ satisfies a particular refinement of φ. It is not clear however, to what extend this approach can be used in practice: Recursive behaviour can only be modelled by infinite synchronizations structures. It thus seems to be questionable whether an effective implementation of the involved method of semantic action refinement can be given. Further, a linear time temporal logic is used whereas we use the branching time Modal Mu-Calculus.

[178] discusses a development technique based on so called "vertical implementation relations". Starting from a specification T and an implementation U, it is shown that T equals U modulo a particular vertical implementation relation $\lesssim^r$ (denoted by $T \lesssim^r U$) iff T is observation congruent (in the sense of Milner [154]) to $U \Uparrow_r$ (denoted by $T \simeq U \Uparrow_r$). There, T, U and $U \Uparrow_r$ are $TCSP$-like process expressions and $U \Uparrow_r$ arises from U by the application of a so called r-abstraction. In contrary to our approach, both, the specifications (T) and the implementations (U and $U \Uparrow_r$) are expressed in an operational fashion. Consequently, the valuable features of dual-language approaches to verification [96] can not be exploited. Furthermore, the specification T remains fixed and can not be adapted to changing recources or requirements, a situation which we wanted to overcome with our approach.

If not used to develop and re-engineer systems, Theorem 4.42, Theorem 4.50 and Theorem 4.51 can still be used to support model checking techniques for systems that could not be handled otherwise due to the (huge or infinite) size of their state spaces. Thus, our approach is also conceptually related to a large body of research which investigates techniques to enhance model checking techniques for huge or infinite state spaces [1]. *"On the fly" model checking* [194, 28, 106, 191] focuses on generating only those parts of the state space that are relevant for the property under consideration. Other techniques exploit *partial order reduction* (surveyed in [167]) or *binary decision diagrams* [31] with the aim to compactify state spaces without loosing information about the systems.

Closest to our approach are the widely investigated *abstraction techniques*, that are mostly based on the framework of *abstract interpretations* (see, for example, [55, 54]). Theorem 4.42 relates process terms and formulas with syntactic refinements of them. The abstractions used in [44, 91, 17, 182] are established on the system description as well.

Syntactic action refinement allows to create hierarchical system descriptions. In [8], a model checking technique is presented that directly exploits the hierarchical structure of the considered systems: The BDD-based algorithm traverses "abstract" transitions by expanding the according "concrete" transition systems on the fly. Hence, the system is analyzed at different levels of abstraction which alleviates the state explosion problem.

Those abstraction techniques differ from our approach in that only the systems are subject to abstractions whereas both, systems and formulas are abstracted in our approach. Furthermore, our abstraction technique is exact whereas most abstraction techniques found in literature are only conservative: Let $S^{\mathcal{A}}$ be the abstraction of the system S. Then we cannot infer $S \not\models \varphi$ from $S^{\mathcal{A}} \not\models \varphi$ if the involved abstraction is only conservative. In our approach, no distinction is made between the treatment of safety, liveness, universal and existential properties. On the other hand, some of the above mentioned approaches allow to create abstract finite state systems from concrete infinite state systems which is not possible using our results. Another method to enhance model checking exploits symmetries which are often exhibited by concurrent systems (see, for example, [43, 74]). Whereas those methods aim to "merge" the symmetries that occur in the transition graph of a system, our technique exploits the structural equalities that occur in the process descriptions (process terms, that is).

In [81], action refinement for an object based temporal logic has been investigated. There, actions are conceived as propositions in the temporal language. Action refinement then amounts to mappings of propositions p to theories Λ which describe the functionality of p on a lower level of abstraction. The main aim of this work is to establish a proof-theory which can be used to show that a "concrete" description ψ (the implementation) of a system is correctly related to an "abstract" description ϕ (the specification) of the system, in the sense, that $\phi \Rightarrow \psi$. Both, ϕ and ψ are formulas of the temporal language. Hence, this approach lacks the features of dual-language formalisms (see [96]). As opposed to our refinement operator-based approach, this work is based on the idea of "hierarchies of designs" (see, e.g., [90]). Again, the specification ϕ is to be fixed and can thus not be subject of any adaptation.

4.6 Future Directions

There remain some interesting problems still to be investigated.

Probably the most challenging problem is to determine under which conditions Theorem 4.42, Theorem 4.50 and Theorem 4.51 hold when those actions that are considered to be "system-internal" are abstracted to "non-observable" actions, so called τ-actions (see, for example [154]). The τ-actions can then, for example, be employed to hide internal synchronization activity of the system from the environment of the system. In the process algebra $R\Sigma$, abstractions of internal system behaviour can be carried out by means of the so called "hiding-operator" of $TCSP$ [162]. This operator transforms observable actions into τ-actions. The presence of the hiding-operator in $R\Sigma$ necessitates to deal with a computational phenomenon called *divergence* (see, for example, [5]). Intuitively, divergence amounts to infinite internal chatter of a system which embodies another computational phenomenon called *livelock*. As opposed to a deadlock where a system is unable to perform any activity (in order to leave an undesired "termination state"), a livelock does not entirely disable a system from any activity. However, a livelock disables the environment to interact with a system as it engages the system in exclusively internal activities. We remember, that continuous interaction with its environment is a prime feature of reactive systems. From the viewpoint of an external observer (the environment), a "livelocked system" thus behaves like a "deadlocked system". Apart from dealing with livelocks on the system level, the specification formalism has to be adapted: An intuitionistic modal logic (see, for example, [189]) has to be used instead of the classical Modal Mu-Calculus of Kozen [119]: For diverging systems P, there exist formulas φ of the Modal Mu-Calculus for which neither $P \models \varphi$ nor $P \not\models \varphi$ holds, that is, the law of excluded middle fails (see, for example, [189]).

Introducing an explicit abstraction operator into the process algebra $R\Sigma$ and investigations to what extent the abstraction technique embodied in Theorem 4.42 and Theorem 4.70 can be fully automated are other interesting questions. In Section 4.2.1, we have seen that using Theorem 4.42 as an abstraction technique amounts to "forgetting about refinement operator sequences". Deciding $P_s \models \varphi_s$ automatically decides $P \models \varphi$ where $P = P_s[\alpha_1 \rightsquigarrow Q_1]\ldots[\alpha_n \rightsquigarrow Q_n]$ and $\varphi = \varphi_s[\alpha_1 \rightsquigarrow Q_1]\ldots[\alpha_n \rightsquigarrow Q_n]$. The important point is, that the state space of P_s can be exponentially smaller than the state space of P. We expect that an algorithm can be devised which, given a process term P and a formula φ, computes a (abstract) process term $P_s \in \Sigma$, an according formula $\varphi_s \in \mu\mathcal{L}_g$, and appropriate refinement sequences $\cdot[\alpha_1 \rightsquigarrow Q_1]\ldots[\alpha_n \rightsquigarrow Q_n]$ and $\cdot[\alpha_1 \rightsquigarrow Q_1]\ldots[\alpha_n \rightsquigarrow Q_n]$ in time, polynomial in $|P|$ (the size of the process expression P) and $|\varphi|$ (the size of the $\mu\mathcal{L}_g$-formula φ). In order to give some reasons for our conjecture, we note that the computation of appropriate refinement sequences (in the sense above) boils down to the computation of such sub-terms $Q_1, \ldots, Q_n \in \Delta$ of P and φ which meet the conditions of Theorem 4.42. Multiple occurrences of such sub-terms $Q_1, \ldots, Q_n \in \Delta$ can be detected in polynomial time by *partition refinement algorithms* (see, for example, [165][24]). In order to achieve substantial state space reductions we can restrict our search for sub-terms $Q_1, \ldots, Q_n$ of P to terms of the form $(P_1\|_A P_2)$: Such terms are the source of state space explosion problems. An obvious bottleneck of any abstraction algorithm that is based on Theorem 4.42 or Theorem 4.70 will be the test that $Q \in \Delta$ (as this is one condition for valid refinements in the aforementioned Theorems). However, as Δ is generated by a context-free grammar, the algorithm of [66] can be used to decide $Q \in \Delta$ in time $O(|Q|^3)$. Considering that all known model checking algorithms for the Modal Mu-Calculus need exponential time to decide whether a system P satisfies

[24]Of course we have to represent process terms P and formulas φ by their parse trees in order to apply these algorithms. However, the size of the parse trees of P and φ grow linearly with respect to $|P|$ and $|\varphi|$.

a specification φ (with respect to $|P|$ and $|\varphi|$), a polynomial time algorithm which implements our abstraction technique would be very useful: The time needed to compute abstractions would be entirely subsumed by the time required by the model checking algorithm used.

Another interesting question is for which process algebras $\mathcal{P}$, logics $\mathcal{L}_1, \mathcal{L}_2$ and refinement operators $\cdot[\alpha \rightsquigarrow \psi]$, an assertion like

$$\forall P \in \mathcal{P}\ \forall \varphi \in \mathcal{L}_1\ \forall \psi \in \mathcal{L}_2 (P \models \varphi \text{ and } Q \models \psi \Leftrightarrow P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow \psi])$$

can be proved. This would allow (partial) specifications ψ of processes Q to be "logically merged" into φ (in Theorem 4.42 and Theorem 4.70 the "whole logical structure" of Q is merged into φ). There, the most important question to be answered is whether the Modal Mu-Calculus is powerful enough to represent a reasonable semantics for the operator $\cdot[\alpha \rightsquigarrow \psi]$.

5 Conclusion

We defined syntactic action refinement for formulas φ of the Modal Mu-Calculus (Section 4.1) and showed that the presented definition conforms to syntactic action refinement for the process algebra $R\Sigma$ (which contains the parallel composition operator of $TCSP$ and recursion) in the sense, that for process terms $P \in R\Sigma$ the assertion

$$P \models \varphi \Leftrightarrow P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q] \quad (*)$$

is valid (see Theorem 4.42). The operator $\cdot[\alpha \rightsquigarrow Q]$ denotes syntactic action refinement both on formulas and process expressions. The development/re-engineering-technique embodied in assertion $(*)$ is called simultaneous syntactic action refinement.

Assertion $(*)$ is valid provided some particular conditions of alphabet-disjointedness and distinction are obeyed. However, two special cases of assertion $(*)$ which do not rely upon these conditions were presented (see Theorem 4.50 and Theorem 4.51).

Assertion $(*)$ can be applied in various ways to the verification of reactive systems one of which is the (a priori) correct transformation of systems induced by the syntactic refinement of specifications: Provided we know $P \models \varphi$, refining φ into

$\varphi[\alpha \rightsquigarrow Q]$ automatically yields $P[\alpha \rightsquigarrow Q]$ such that $P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q]$ (see Example 4.43, the case study in Section 4.2.1 and Example 4.52).

Further, we explained how the obtained results can be used as an abstraction technique (see Example 4.43, the case study in Section 4.2.1 and Example 4.71) and that that the results can sometimes make it possible to model check systems that would remain infeasible otherwise.

We explained that assertion $(*)$ can be combined with classical verification techniques like, for example, with model checking algorithms (see the case study in Section 4.2.1). Hence, assertion $(*)$ extends classical verification technique which leads to settings, that allow to automatically develop/re-engineer formally correct reactive systems by hierarchically enriching/abstracting specifications with details.

In order to obtain an efficient development/re-engineering-technique, we introduced the generalized Modal Mu-Calculus (Definition 4.53 and Definition 4.57) and defined a reduction function for this logic (Definition 4.54 and Definition 4.55). Computing reductions takes time $O(|\varphi| * |Q|)$ for hierarchical specifications $\varphi[\alpha \rightsquigarrow Q] \in R\mu\mathcal{L}_g$ and $O(|P| * |Q|)$ for hierarchical process terms $P[\alpha \rightsquigarrow Q] \in R\Sigma$, provided the refinements are not nested (that is, φ, P and Q contain no refinement operators). Hence, after each refinement step, implementation near process terms and low level specifications can be efficiently derived via the application of the reduction functions. Theorem 4.70 embodies this efficient development/re-engineering-technique.

We used the expressive Modal Mu-Calculus as specification formalism and the intuitive notion of transition systems as the semantic model for reactive systems. We thus believe that our results can provide a basis for similar investigations that employ other logics and semantic models.

The obtained results have been applied to a serial of examples and a more thorough case study is carried out in Section 4.2.1. Particular parts of this thesis are also available in a more condensed form (see [136, 137, 138, 139, 140]).

Index

References

[1] *International Conf. on Computer-Aided Verification*, volume (LNCS) 407 (1989), 531 (1990), 575 (1991), 663 (1992), 697 (1993), 818 (1994), 939 (1995), 1102 (1996), 1254 (1997), 1427 (1998), 1633 (1999), New York, NY, USA. Springer-Verlag Inc.

[2] In W. P. De Roever G. Rozenberg J. W. De Bakker, editor, REX Workshop on *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness,* Mook, The Netherlands, May/June 1989, volume 430 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[3] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Automata, Languages and Programming. 16th Int. Colloquium Proceedings*, number 372 in Lecture Notes in Computer Science, pages 1–17, Stresa, Italy, July 1989. Springer-Verlag.

[4] L. Aceto and M. Hennessy. Adding action refinement to a finite process algebra. *Lecture Notes in Computer Science*, 510:506–519, 1991.

[5] L. Aceto and M. Hennessy. Termination, deadlock, and divergence. *Journal of the ACM*, 39(1):147–187, January 1992.

[6] B. Alpern and F. B. Schneider. Recognizing Safety and Liveness. *Distributed Comp.*, 2:117–126, 1987.

[7] R. Alur, R. K. Brayton, T. A. Henzinger, and S. Qadeer. Partial-order reduction in symbolic state space exploration. *Lecture Notes in Computer Science*, 1254:340–351, 1997.

[8] R. Alur, T. A. Henzinger, and S. K. Rajamani. Symbolic exploration of transition hierarchies. *Lecture Notes in Computer Science*, 1384:330–344, 1998.

[9] H. R. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal μ-calculus. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 144–153, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[10] E. Astesiano and G. Reggio. Formalism and Method. In M. Bidoit and M. Dauchet, editors, *Proc. Conf. on Theory and Practice of Sw Development (TAPSOFT 97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 93–114, Lille, France, 1997. Springer-Verlag, Berlin.

[11] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar sequential processes (extended abstract). In ACM, editor, *POPL '89. Proceedings of the sixteenth annual ACM symposium on Principles of programming languages, January 11–13, 1989, Austin, TX*, pages 191–201, New York, NY, USA, 1989. ACM Press.

[12] C. Baier and M. Majster-Cederbaum. The connection between an event structure semantics and an operational semantics for TCSP. *Acta Informatica*, 31(1):81–104, 1994.

[13] J. Barwise. Mathematical proofs of computer systems correctness. *Notices of the American Mathematical Society*, 36:844–851, 1989.

[14] I. Beer, S. Ben-David, C. Eisner, and D. Geist. RuleBase: Model checking at IBM. *Lecture Notes in Computer Science*, 1254:480–483, 1997.

[15] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20(3):207–226, December 1983.

[16] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property Preserving Simulations. In G.V. Bochmann and D.K. Probst, editors, *Proceedings of the Fourth Workshop on Computer-Aided Verification*, July 1992.

[17] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331, Vancouver, Canada, June 1998. Springer-Verlag.

[18] S. Berezin, E. Clarke, S. Jha, and W. Marrero. Model checking algorithms for the mu-calculus. Technical Report CMU-CS-96-180, September 1996. ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-180.ps.

[19] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, May 1985.

[20] G. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In *Proceedings, 11*$^{\text{th}}$ *Annual IEEE Symposium on Logic in Computer Science*, pages 304–312, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[21] A. Bouajjani, J. C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. *Lecture Notes in Computer Science*, 510:76–92, 1991.

[22] A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In G. v. Bochmann and D. K. Probst, editors, *Computer-Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 96–108. Springer-Verlag, 1992.

[23] G. Boudol and I. Castellani. Flow models of distributed computations: Three equivalent semantics for CCS. *Information and Computation*, 114(2):247–314, 1 November 1994.

[24] J. Bowen and V. Stavridou. The industrial take-up of formal methods in safety-critical and other areas: A perspective. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 183–195. Formal Methods Europe, Springer-Verlag, April 1993. Lecture Notes in Computer Science 670.

[25] J. P. Bowen and V. Stavridou. Formal methods and software safety. In H. H. Frey, editor, *Safety of computer control systems 1992 (SAFECOMP'92), Computer Systems in Safety-critical Applications, Proc. IFAC Symposium, Zürich, Switzerland, 28–30 October 1992*, pages 93–98. Pergamon Press, 28–30 October 1992.

[26] J. C. Bradfield. *Verifying Temporal Proporties of Systems.* Birkhaeuser, Boston, 1992.

[27] J. C. Bradfield. The modal mu-calculus alternation hierarchy is strict. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 233–246, Pisa, Italy, 26–29 August 1996. Springer-Verlag.

[28] J. C. Bradfield and C. Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science*, 96(1):157–174, April 1992.

[29] S. D. Brookes. On the relationship of CCS and CSP. In Josep Díaz, editor, *Automata, Languages and Programming, 10th Colloquium*, volume 154 of *Lecture Notes in Computer Science*, pages 83–96, Barcelona, Spain, 18–22 July 1983. Springer-Verlag.

[30] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[31] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[32] A. Bundy. A survey of automated deduction. Informatics Research Report EDI-INF-RR-0001, Scotland, April 1999.

[33] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, Pennsylvania, 4–7 June 1990. IEEE Computer Society Press.

[34] O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Theoretical Computer Science*, 221(1–2):251–270, June 1999.

[35] I. Castellani and M. Hennessy. Distributed bisimulations. *Journal of the ACM*, 36(4):887–911, October 1989.

[36] L. Castellano, G. De Michelis, and L. Pomello. Concurrency vs interleaving: an instructive example. *Bulletin of the European Association for Theoretical Computer Science*, 31:12–15, February 1987. Technical Contributions.

[37] B. F. Chellas. *Modal Logic, an Introduction.* Cambridge University Press, Cambridge, 1980.

[38] S. Christensen, H. Hüttel, and C. Stirling. Bisimulation equivalence is decidable for all context-free processes. *Information and Computation*, 121(2):143–148, September 1995.

[39] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[40] E. M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Particial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 428–437, Noordwigherhout, Netherland, May–June 1988. Springer-Verlag.

[41] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[42] E. M. Clarke and E. A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Logics of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[43] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis, editor, *Proceedings of The Fifth Workshop on Computer-Aided Verification*, June/July 1993.

[44] E. M. Clarke, O. Grumberg, and D. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[45] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[46] E. M. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[47] E. M. Clarke, K. L. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. *Lecture Notes in Computer Science*, 1102:419–422, 1996.

[48] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178, September 1996. ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-178.ps.

[49] R. Cleaveland. The concurrency workbench: A semantics-based verification tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[50] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The concurrency factory: a development environment for concurrent systems. *Lecture Notes in Computer Science*, 1102:398–401, 1996.

[51] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 24–37, Berlin, June 1990. Springer.

[52] R. Cleaveland and S. Sims. The NCSU concurency workbench. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[53] R. Cleaveland and S. A. Smolka. Strategic directions in concurrency research. *ACM Computing Surveys*, 28(4):607–625, December 1996.

[54] P. Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324–328, June 1996.

[55] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[56] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering: An International Journal*, 6(1):69–95, January 1999.

[57] Max J. Cresswell and G. E. Hughes. *An Introduction to Modal Logic.* Methuen, London, 1968.

[58] M. Dam. CTL$*$ and ECTL$*$ as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, April 1994.

[59] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[60] A. A. S. Danthine. Protocol representation with finite-state models. *IEEE trans. on commun.*, COM-28:632–643, 1980.

[61] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24(2):211–237, April 1987.

[62] V. Diekert and G. Rozenberg. *The Book of Traces.* World Scientific, Singapore, 1995.

[63] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, October 1971. Reprinted in *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrot, Eds., Academic Press, 1972, pp. 72–93. This paper introduces the classical synchronization problem of Dining Philosophers.

[64] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[65] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 54–69, Liege, Belgium, July 1995. Springer Verlag.

[66] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[67] A. Th. Eiriksson and K. L. McMillan. Using formal verification/analysis methods on the critical path in system design: a case study. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 367–380, Liege, Belgium, July 1995. Springer Verlag.

[68] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072, Amsterdam, 1990. Elsevier Science Publishers.

[69] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30:1–24, 1985.

[70] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In IEEE, editor, *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 368–377, San Juan, Porto Rico, October 1991. IEEE Computer Society Press.

[71] E. A. Emerson and C. L. Lei. Modalities for model checking: Branching time strikes back. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, Louisiana, January 13–16, 1985. ACM SIGACT-SIGPLAN, ACM Press. Extended abstract.

[72] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Symposium on Logic in Computer Science (LICS '86)*, pages 267–278, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[73] E. A. Emerson and A. P. Sistla. Deciding full branching time logic. *Information and Control*, 61(3):175–201, June 1984.

[74] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In C. Courcoubetis, editor, *Proceedings of The Fifth Workshop on Computer-Aided Verificaton*, June/July 1993.

[75] J. Srinivasan E. A. Emerson. Branching time temporal logic. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 123–172, Berlin, May 30–June 3 1989. Springer.

[76] U. Engberg, P. Grønning, and L. Lamport. Mechanical verification of concurrent systems with TLA. In G. v. Bochmann and D. K. Probst, editors, *Proceedings of the 4th International Workshop on Computer Aided Verification,* Montreal, Canada, volume 663 of *Lecture Notes in Computer Science*, pages 44–55. Springer-Verlag, 1992.

[77] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[78] European Space Agency. ARIANE 5 flight 501 failure, July 1996. Report by the Inquiry Board.

[79] S. Feferman, J. W. Dawson, S. C. Kleene, G. H. Moore, R. M. Solovay, and J. van Heijenoort. *Kurt Gödel: Collected Works. Vol. 1: Publications 1929-1936.* Oxford University Press, Walton Street, Oxford OX2 6DP, UK, 1986.

[80] J. H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

[81] J. L. Fiadeiro and T. Maibaum. Sometimes “tomorrow” is “sometime” — action refinement in a temporal logic of objects. *Lecture Notes in Computer Science*, 827:48–66, 1994.

[82] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, April 1979.

[83] D. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic.* Clarendon Press, Oxford, 1994.

[84] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time logic model checking. *INFCTRL: Information and Computation (formerly Information and Control)*, 150, 1999.

[85] G. De Giacomo and M. Lenzerini. Concept language with number restrictions and fixpoints, and its relationship with Mu-calculus. In A. G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 411–415, Chichester, August 8–12 1994. John Wiley and Sons.

[86] P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke, editor, *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90), Rutgers, New Jersey, 1990*, number 531 in Lecture Notes in Computer Science, pages 176–185, Berlin-Heidelberg-New York, 1991. Springer.

[87] U. Goltz. On representing CCS programs by finite Petri nets. In Michael P. Chytil, Ladislav Janiga, and Václav Koubek, editors, *Mathematical Foundations of Computer Science 1988*, volume 324 of *Lecture Notes in Computer Science*, pages 339–350, Carlsbad, Czechoslovakia, 29 August–2 September 1988. Springer.

[88] U. Goltz, R. Gorrieri, and A. Rensink. On syntactic and semantic action refinement. *Lecture Notes in Computer Science*, 789:385–404, 1994.

[89] U. Goltz and A. Mycroft. On the relationship of CCS and Petri nets. In Jan Paredaens, editor, *Automata, Languages and Programming, 11th Colloquium*, volume 172 of *Lecture Notes in Computer Science*, pages 196–208, Antwerp, Belgium, 16–20 July 1984. Springer-Verlag.

[90] R. Gorrieri and A. Rensink. Action refinement. Technical Report UBLCS-99-9, University of Bologna (Italy). Department of Computer Science., April 1999.

[91] S. Graf. Verification of distributed cache memory by using abstractions. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 207–219, Standford, California, USA, June 1994. Springer-Verlag.

[92] S. Graf and B. Steffen. Compositional minimisation of finite state processes. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 186–196. Springer-Verlag, 1990.

[93] J. F. Groote and H. Hüttel. Undecidable equivalences for basic process algebra. *Information and Computation*, 115(2):354–371, December 1994.

[94] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO, ASI Series*, pages 447–498. Springer-Verlag, New York, 1985.

[95] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In M. C. Gaudel and J. Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer-Verlag, 1996.

[96] C. Heitmeyer, J. Kirby, Jr., B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

[97] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and J. van Leeuwen, editors, *Proceedings* 7^{th} *ICALP,* Noordwijkerhout, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer-Verlag, July 1980. This is a preliminary version of: Algebraic laws for nondeterminism and concurrency. *JACM*, 32(1):137–161, 1985.

[98] M. Hennessy and C. Stirling. The power of the future perfect in program logics. *Information and Control*, 67(1–3):23–52, October/November/December 1985.

[99] T. A. Henzinger, O. Kupferman, and S. Qadeer. From pre-historic to post-modern symbolic model checking. *Lecture Notes in Computer Science*, 1427:195–206, 1998.

[100] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proceedings*

REX Workshop on Real-Time: Theory in Practice, Mook, The Netherlands, June 1991, volume 600 of *Lecture Notes in Computer Science*, pages 226–251. Springer-Verlag, 1992.

[101] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[102] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[103] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.

[104] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, London, 1979.

[105] M. Huhn. Action refinement and property inheritance in systems of sequential agents. In *CONCUR '96*, volume 1119 of *Lecture Notes in Computer ScienceS*, pages 639–654. Springer-Verlag, aug 1996.

[106] H. Hungar. Local model checking for parallel compositions of context-free processes. *Lecture Notes in Computer Science*, 836:114–128, 1994.

[107] D. Janin and I. Walukiewicz. On the expressive completeness of the modal mu-calculus w.r.t. monadic second order logic. In *Proc. CONCUR'96*, volume 1119 of *Lecture Notes in Computer Science*, 1996.

[108] B. Jonsson and Y. K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167(1–2):47–72, 30 October 1996.

[109] A. Joyal, M. Nielsen, and G. Winskel. Bisimulation and open maps. In Robert L. Constable, editor, *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science*, pages 418–427, Montreal, Canada, June 1993. IEEE Computer Society Press.

[110] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of the Second Annual ACM*

SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pages 228–240, Montreal, Quebec, Canada, 17–19 August 1983.

[111] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.

[112] S. Katz and D. Peled. Interleaving set temporal logic. *Theoretical Computer Science*, 75(3):21–43, 1991. Preliminary versions appeared in 6th Annual ACM Symposium on Distributed Computing 1987, and in LNCS 398, Temporal Logic in Specification, 1988.

[113] P. Kelb, D. Dams, and R. Gerth. Efficient symbolic model checking of the full μ-calculus using compositional abstractions. Computing Science Reports 95/31, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, October 1995.

[114] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, July 1976.

[115] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. *Lecture Notes in Computer Science*, 803:273–346, 1994.

[116] S. Kimura, A. Togashi, and N. Shiratori. Extension of synthesis algorithm of recursive processes to μ-calculus. *Information Processing Letters*, 58(2):97–104, May 1996.

[117] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies, Annals of Math. Studies 34*, pages 3–40. Princeton, New Jersey, 1956.

[118] M. Klein, J. Knoop, D. Koschützki, and B. Steffen. DFA and OPT-METAFrame: A tool kit for program analysis and optimization. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 422–426. Springer-Verlag, 1996.

[119] D. Kozen. Results on the propositional mu -calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.

[120] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. The MIT Press, New York, N.Y., 1990.

[121] S. Kripke. A semantical analysis of modal logic I: normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963. Announced in *Journal of Symbolic Logic*, **24**, 1959, p. 323.

[122] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *JACM: Journal of the ACM*, 47, 2000.

[123] O. Kupferman and M.Y. Vardi. μ-calculus synthesis. In *Proc. 25th International Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[124] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton Univ. Press, 1994.

[125] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[126] L. Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, Systems Research Centre, 25 December 1991.

[127] F. Laroussinie, S. Pinchinat, and Ph. Schnoebelen. Translations between modal logics of reactive systems. *Theoretical Computer Science*, 140(1):53–71, 20 March 1995.

[128] F. Laroussinie and Ph. Schnoebelen. A hierarchy of temporal logics with past. *Theoretical Computer Science*, 148(2):303–324, 1995.

[129] K. G. Larsen. Modal specifications. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems: Proceedings*, LNCS 407, pages 232–246. Springer-Verlag, 1989.

[130] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, September 1991.

[131] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[132] F. Levi. A compositional μ-calculus proof system for statecharts processes. *Theoretical Computer Science*, 216(1–2):271–310, March 1999.

[133] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In R. Parikh, editor, *Proceedings 3rd Workshop on Logics of Programs, Brooklyn, NY, USA, 17–19 June 1985*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer-Verlag, Berlin, 1985.

[134] D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. R . Marrero. An improved algorithm for the evaluation of fixpoint expressions. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 338–350, Standford, California, USA, June 1994. Springer-Verlag.

[135] R. Loogen and U. Goltz. Modelling nondeterministic concurrent processes with event structures. *Fundamenta Informaticae*, XIV(1):39–74, January 1991.

[136] M. Majster-Cederbaum and F. Salger. On syntactic action refinement and logic. Technical Report 7/99, Fakultät für Mathematik und Informatik, Universität Mannheim, D7.27, Germany, 1999.

[137] M. Majster-Cederbaum and F. Salger. Syntactic action refinement in the modal mu-calculus and its applications to the verification of reactive systems (in proceedings of the 11th nordic workshop on programming theory). Technical Report 1999-008, Department of Information Technology, Uppsala University, Sweden, oct 1999.

[138] M. Majster-Cederbaum and F. Salger. A verification technique based on syntactic action refinement in a TCSP-like process algebra and the Hennessy-Milner-Logic. In *Proceedings of the Asian Computing Science Conference*, volume 1742 of *Lecture Notes in Computer Science*, pages 379–380. Springer, dec 1999.

[139] M. Majster-Cederbaum and F. Salger. Correctness by construction: Towards verification in hierarchical system development. In *Proceedings of the 7th SPIN Workshop on Model Checking of Software*, volume 1885 of *Lecture Notes in Computer Science*, pages 163–180. Springer, sep 2000.

[140] M. Majster-Cederbaum, F. Salger, and M. Sorea. A priori verification of reactive systems. In *Formal Methods for Distributed System Development (Proc. FORTE/PSTV 2000)*. Kluwer Academic Publishers, oct 2000.

[141] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *REX School and Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency,* Noordwijkerhout, The Netherlands, May/June 1988, volume 354 of *Lecture Notes in Computer Science*, pages 201–284. Springer-Verlag, 1989.

[142] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *In Proc. 9th ACM Symp. on Princ. of Dist. Comp.*, pages 377–408, 1990.

[143] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer, 1991.

[144] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems - Safety -*. Springer-Verlag, New York, 1995.

[145] Z. Manna and P. Wolper. Synthesis of communicating processes form temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6:68–93, 1984.

[146] F. E. Marschner. Practical challenges for industrial formal verification tools. In Orna Grumberg, editor, *Proceedings of Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 1–2, Berlin, Germany, June 1997. Springer.

[147] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[148] A. Mazurkiewicz. Basic notions of trace theory. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 285–363, Berlin, May 30–June 3 1989. Springer.

[149] W. S. McCulloch and W. Pitts. A logical calculus of the idea immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.
The original paper on McCulloch-Pitts neurons.

[150] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992. CMU-CS-92-131.

[151] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *13th Annual Symposium on Switching and Automata Theory*, pages 125–129, The University of Maryland, 25–27 October 1972. IEEE.

[152] G. J. Milne. CIRCAL and the representation of communication, concurrency, and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, April 1985.

[153] R. Milner. A modal characterization of observable machine-behaviour. In E. Astesiano and C. Böhm, editors, *Proceedings CAAP '81*, volume 112 of *Lecture Notes in Computer Science*, pages 25–34, Genoa, March 1981. Springer-Verlag.

[154] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.

[155] F. Moller. Infinite results. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 195–216, Pisa, Italy, 26–29 August 1996. Springer-Verlag.

[156] A. P. Moore. Using CSP to develop trustworthy hardware. In *Compass '90: 5th Annual Conference on Computer Assurance*, pages 126–134, Gaithersburg, Maryland, 1990. National Institute of Standards and Technology.

[157] H. M. Müller, C. M. Holt, A. Watters, and J. H. Fetzer. More on the very idea (letters). *Communications of the ACM*, 32(4):506–512, April 1989.

[158] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. *Lecture Notes in Computer Science*, 600:526–548, 1992.

[159] M. Nielsen, V. Sassone, and G. Winskel. Relationships between models of concurrency. *Lecture Notes in Computer Science*, 803:425–476, 1994.

[160] D. Niwinski. On fixed-point clones. In Laurent Kott, editor, *Proceedings of the 13th International Colloquium on Automata, Languages and Programming*, volume 226 of *LNCS*, pages 464–473, Rennes, France, July 1986. Springer.

[161] D. Niwiński. Fixed points vs. infinite generation. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 402–409, Edinburgh, Scotland, 5–8 July 1988. IEEE Computer Society.

[162] E. R. Olderog. TCSP: Theory of communicating sequential processes. In *Advances in Petri Nets 1987, ed. Grzegorz Rozenberg, LNCS 266; Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, LNCS 254-255, 1987, LNCS 188 (1984), LNCS 340 (1988), LNCS 483 (1991).* 1986.

[163] L. Osterweil. Strategic directions in software quality. *ACM Computing Surveys*, 28(4):738–750, December 1996.

[164] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[165] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987.

[166] D. Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science: 5th GI-Conference, Karlsruhe*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Berlin, Heidelberg, and New York, March 1981. Springer-Verlag.

[167] D. Peled. Ten years of partial order reduction. *Lecture Notes in Computer Science*, 1427:17–28, 1998.

[168] W. Penczek. A temporal logic for event structures. *Fundamenta Informaticae*, XI (3):297–326, 1988.

[169] W. Penczek. Branching time and partial order in temporal logic. Technical Report UMCS–91–3–3, Department of Computer Science, Manchester University, Oxford Rd., Manchester M13 9PL, UK, 1991.

[170] D. Perrin. Finite automata. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol b, Formal Models and Semantics*, pages 1–57. Elsevier, Amsterdam, The Netherlands, 1990.

[171] J. C. Pleasant, L. Paulson, A. Cohen, M. G. Bevier, M. K. Smith, W. D. Young, T. R. Clune, S. Savitzky, and J. H. Fetzer. Correspondence on fetzer, program verification: The very idea. *Communications of the ACM*, 32(3):374–381, March 1989.

[172] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[173] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey and current trends. In J. E. de Bakker et al., editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, New York, 1986.

[174] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In ACM, editor, *POPL '89. Proceedings of the sixteenth annual ACM symposium on Principles of programming languages, January 11–13, 1989, Austin, TX*, pages 179–190, New York, NY, USA, 1989. ACM Press.

[175] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[176] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer.Math.Soc.*, 141:1–35, 1969.

[177] A. Rabinovich. Complexity of equivalence problems for concurrent systems of finite agents. *Information and Computation*, 139(2):111–129, 15 December 1997.

[178] A. Rensink and R. Gorrieri. Action refinement as an implementation relation. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 772–786. Springer-Verlag, 1997.

[179] M. Roggenbach and M. Majster-Cederbaum. Towards a unified view of bisimulation: a comparative study. *Theoretical Computer Science*, 238(1–2):81–130, May 2000.

[180] H. Rudin. Network protocols and tools to help produce them. *ANNREVCS: Annual Review of Computer Science*, 2, 1987.

[181] J. Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1995. Also available as NASA Contractor Report 4673, August 1995, and to be issued as part of the *FAA Digital Systems Validation Handbook* (the guide for aircraft certification).

[182] H. Saïdi and N. Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454, Trento, Italy, July 1999. Springer-Verlag.

[183] V. Sassone, M. Nielsen, and G. Winskel. Models for concurrency: Towards a classification. *Theoretical Computer Science*, 170(1–2):297–348, 15 December 1996.

[184] D. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

[185] J. Sifakis. Deadlocks and livelocks in transition systems. In P. Dembinski, editor, *Mathematical Foundations of Computer Science 1980, Proceedings of the 9th Symposium*, volume 88 of *Lecture Notes in Computer Science*, pages 587–600, Rydzyna, Poland, 1–5 September 1980. Springer.

[186] J. Sifakis. Research directions for concurrency. *ACM Computing Surveys*, 28(4es):55, December 1996.

[187] E. W. Stark. Concurrent transition systems. *Theoretical Computer Science*, 64(3):221–269, May 1989.

[188] B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21(2):115–139, October 1993.

[189] C. Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49(2-3):311–347, 1987.

[190] C. Stirling. Modal and temporal logics. In S. Abramsky, Dov M. Gabbay, and T. S.Ẽ. Maibaum, editors, *Handbook of Logic in Computer Science. Volume 2. Background: Computational Structures*, pages 477–563. Oxford University Press, 1992.

[191] C. Stirling. Local model checking games (extended abstract). In Insup Lee and Scott A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International Conference*, volume 962 of *Lecture Notes in Computer Science*, pages 1–11, Philadelphia, Pennsylvania, 21–24 August 1995. Springer-Verlag.

[192] C. Stirling. Modal and temporal logics for processes. *Lecture Notes in Computer Science*, 1043:149–237, 1996.

[193] C. Stirling. The joys of bisimulation. *Lecture Notes in Computer Science*, 1450:142–151, 1998.

[194] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, October 1991.

[195] R. S. Streett and E. A. Emerson. The propositional mu-calculus is elementary. In Jan Paredaens, editor, *Automata, Languages and Programming, 11th Colloquium*, volume 172 of *Lecture Notes in Computer Science*, pages 465–472, Antwerp, Belgium, 16–20 July 1984. Springer-Verlag.

[196] R. S. Streett and E. A. Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81(3):249–264, June 1989.

[197] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[198] D. Taubner. *Finite representations of CCS and TCSP programs by automata and Petri nets*, volume 369 of *Lecture Notes in Computer Science.* Springer-Verlag Inc., New York, NY, USA, 1989.

[199] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), North-Holland, 1989.

[200] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[201] R. van Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions. In *Proc. of the MFCS*, volume 379 of *Lecture Notes in Computer Science*, pages 237–248, Berlin, August28 September–1 1989. Springer.

[202] R. J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90: Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297, Amsterdam, The Netherlands, 27–30August 1990. Springer-Verlag.

[203] R. J. van Glabbeek. The linear time–branching time spectrum II: The semantics of sequential systems with silent moves (extended abstract). In Eike Best, editor, *CONCUR '93: 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81, Hildesheim, Germany, 23–26August 1993. Springer-Verlag.

[204] R. J. van Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. Hildesheimer Informatik-Berichte, Germany, April 1998.

[205] M. Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. In *Proceedings, Symposium on Logic in Computer Science*, pages 167–176, Ithaca, New York, 22–25 June 1987. The Computer Society of the IEEE.

[206] M. Y. Vardi. Linear vs. branching time: A complexity-theoretic perspective. In *LICS: IEEE Symposium on Logic in Computer Science*, 1998.

[207] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS '86)*, pages 332–345, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[208] I. Walukiewicz. A complete deductive system for the μ-calculus. Technical Report RS-95-6, BRICS, Dept. of Computer Science, Univ. of Århus, Denmark, January 1995.

[209] I. Walukiewicz. Completeness of Kozen's axiomatisation of the propositional μ-calculus. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 14–24, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.

[210] I. Wegener. Worst case examples for operations on OBDDs. *IPL: Information Processing Letters*, 74, 2000.

[211] B. Willems and P. Wolper. Partial-order methods for model checking: From linear time to branching time. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 294–303, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[212] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–193. ACM, ACM, January 1986.

[213] P. Wolper. On the relation of programs and computations to models of temporal logic. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors, *Proceedings of Temporal Logic in Specification, Oxford 1987*, pages 75–123. Springer-Verlag, 1989.

[214] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In Eike Best, editor, *CONCUR '93: 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 233–246, Hildesheim, Germany, 23–26August 1993. Springer-Verlag.

[215] D. Yu. A view on three R's (3rs): Reuse, reengineering, and reverse-engineering. *SIGSOFT Software Engineering Notes*, 16(3):69, July 1991.

[216] J. Yuan, J. Shen, J. Abraham, and A. Aziz. On combining formal and informal verification. *Lecture Notes in Computer Science*, 1254:376–387, 1997.

[217] J. Zucker. The propositional μ-calculus and its use in model checking. In Peter E. Lauer, editor, *Papers from Intl. Lecture Series on Functional Programming, Concurrency, Simulation and Automated Reasoning, McMaster Univ., Hamilton, Ont., Canada, 1991–1992*, volume 693 of *Lecture Notes in Computer Science*, pages 118–126. Springer-Verlag, Berlin, 1993.

Zusammenfassung

Die intensiv untersuchte Methode der syntaktischen Aktionsverfeinerung ermöglicht den formalen und hierarchischen Entwurf reaktiver Systeme in (prozess-) algebraischen Entwicklungssprachen: Mit Hilfe der syntaktischen Aktionsverfeinerung können (atomare) Aktionen eines abstrakten Systementwurfs durch komplexe (nicht-atomare) Aktivitäten beschrieben werden.

Eigenschaften von Systementwürfen werden häufig in Temporaler Logik spezifiziert. Eine besonders ausdrucksstarke Temporale Logik ist der Modale Mu-Kalkül.

Unter der Verifikation eines Systementwurfs wird der mathematische Nachweis erwünschter Eigenschaften des Systementwurfs verstanden.

Diese Arbeit beschäftigt sich mit der Integration herkömmlicher Verifikationstechniken (wie zum Beispiel der Methode der Modell-Prüfung - engl.: model checking) in den hierarchischen, auf der syntaktischen Aktionsverfeinerung basierenden, Entwurf reaktiver Systeme. Dazu wird zunächst eine bereits existierende Methode zur syntaktischen Aktionsverfeinerung für die Prozess-Algebra TCSP vorgestellt und für unsere Zwecke erweitert. Anschließend definieren wir eine Methode zur syntaktischen Aktionsverfeinerung für Formeln des Modalen Mu-Kalküls. Daraufhin wird nachgewiesen, dass die Methode zur syntaktischen Aktionsverfeinerung für den Modalen Mu-Kalkül in kanonischer Weise zu der Methode zur syntaktischen Aktionsverfeinerung für TCSP passt: Unter bestimmten Voraussetzungen gilt, dass ein abstrakter Systementwurf P eine abstrakte Spezifikation φ genau dann erfüllt, wenn ein detaillierter Systementwurf P' eine detaillierte Spezifikation φ' erfüllt. Dabei entsteht φ' aus φ durch die Anwendung der Methode zur syntaktischen Aktionsverfeinerung im Modalen Mu-Kalkül. Diese Verfeinerung induziert eine syntaktische Aktionsverfeinerung in TCSP welche P nach P' überführt. Vorausgesetzt, dass der Systementwurf P die Spezifikation φ erfüllt, liefert uns also die Verfeinerung von φ in φ' automatisch einen "a priori" korrekten Systementwurf P' (in dem Sinn, dass P' die Spezifikation φ' erfüllt).

Anschließend wird gezeigt, dass jeweils eine der beiden Implikationen in der obigen Äquivalenzaussage für bestimmte Fragmente des Modalen Mu-Kalküls unter sehr schwachen Voraussetzungen bewiesen werden kann.

Desweiteren führen wir eine Erweiterung des Modalen Mu-Kalküls ein und definieren eine Methode zur syntaktischen Aktionsverfeinerung für diese Logik. Es wird gezeigt, dass die obige Äquivalenzaussage auch bei Verwendung der Erweiterung des Modalen Mu-Kalküls gültig ist. Die Erweiterung des Modalen Mu-Kalküls erlaubt eine kompaktere Spezifizierung von Systemeigenschaften als dies mit dem herkömmlichen Modalen Mu-Kalkül möglich ist. Dieser Umstand wird ausgenutzt, um eine effiziente Entwicklung von a priori korrekten Systementwürfen auf der Basis unserer Methode bereitzustellen. Darüber hinaus wird eine Anwendung unserer Methode beschrieben, welche zur Beschleunigung der Entwurfsverifikation mittels Modell-Prüfung benutzt werden kann. Eine Fallstudie demonstriert die Anwendung der Ergebnisse dieser Arbeit.

Verification in the Hierarchical Development of Reactive Systems.