Correctness by Construction: Towards Verification in Hierarchical System Development

Mila Majster-Cederbaum, Frank Salger

Universität Mannheim
Fakultät für Mathematik und Informatik,
D7, 27, 68131 Mannheim, Germany
{mcb, fsalger}@pi2.informatik.uni-mannheim.de

Abstract. In many approaches to the verification of reactive systems, operational semantics are used to model systems whereas specifications are expressed in temporal logics. Most approaches however assume, that the initial specification is indeed the intended one. Changing the specification thus necessitates to find an accordingly adapted system and to carry out the verification from scratch. During a systems life cycle however, changes of the requirements and resources necessitate repeated adaptations of specifications. We here propose a method that supports *syntactic action refinement* (SAR) and allows to automatically obtain (a priori) correct systems by hierarchically adding details to the according specifications. More precisely, we give a definition of SAR for formulas φ of the *Modal Mu-Calculus* (denoted by $\varphi[\alpha \rightsquigarrow Q]$) that conforms to SAR for $TCSP$-like process terms P (denoted $P[\alpha \rightsquigarrow Q]$) in the following sense: The system induced by a process term P satisfies a specification φ if and only if the system induced by the refined term $P[\alpha \rightsquigarrow Q]$ satisfies the refined specification $\varphi[\alpha \rightsquigarrow Q]$. *Model checking* is used to decide, whether the initial system satisfies the initial specification. If we are not satisfied with the obtained refinement $P[\alpha \rightsquigarrow Q]$ or $\varphi[\alpha \rightsquigarrow Q]$ we reuse already gained verification information (P satisfies φ that is) as the basis for other refinement steps. This can be conceived as a method to reengineer systems. Syntactic action refinement allows to handle infinite state systems. Further, the system induced by P might be exponentially smaller that the system induced by $P[\alpha \rightsquigarrow Q]$. We explain how our results can thus also be exploited to enhance model checking techniques. Finally, we apply our results to an example.

1 Introduction

Faults of *reactive systems* (like, for example of air traffic control systems) can imply severe consequences, whence proving the correctness of such systems with respect to the expected behaviour is inevitable. We are concerned with a dual language approach to verification in which systems are modelled operationally whereas specifications are given in an appropriate temporal logic. The obvious method to obtain verified systems is to come up with a specification of the intended system and subsequently invest experience and guess work to design an

according system. *Model checking* can then be used for the verification to follow. However, adaptation of the system and subsequent verification has to be undergone repeatedly until the system meets the specification, a time consuming task. Another method uses transformational methods to construct a (a priori correct) system directly from the specification [CE81,MW84,PR89,AE89], 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 systems life cycle however, specifications (and hence the according systems) are most often subject to repeated adaptations actuated by changed requirements or resources. Such changes also emerge in realistic scenarios for system development where the specification is arrived at by successivly enriching the 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 system P satisfies a specification φ (denoted $P \models \varphi$), transforming φ into a modified specification φ' should entail a transformation of P into P' such that $P' \models \varphi'$. This paradigm supports (a priori) correct system maintenance and stepwise *development of correct reactive systems*. Reversely, *reengineering* amounts to the ability to infer $P \models \varphi$ from $P' \models \varphi'$. This allows to reuse verification knowledge that has already been gained through preceding steps in a development sequence.

We here present an action based development/reengineering-technique that exploits the method of *syntactic action refinement* (see [GR99] for a survey), SAR for short. 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 $P[\alpha \rightsquigarrow Q]$. SAR however complicates the task of verification. For example, many behavioural equivalences used for verification [BBR90] are not preserved under SAR [CMP87]. Considering a verification setting based on process algebras and (action based) logics, the following problem arises: Knowing that the system induced by a process term P satisfies a particular formula does not tell us which formulas are satisfied by the system induced by the refined term $P[\alpha \rightsquigarrow Q]$.

To overcome this problem, we define SAR for formulas φ of the Modal Mu-Calculus [Koz83] that conforms to SAR for $TCSP$-like process terms P and 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. The distinguishing features of our approach are

- The use of SAR. This supports hierarchical development of infinite state systems: As opposed to semantic action refinement, SAR is applied to process terms whence state spaces do not have to be handled algorithmically to implement SAR.

- The refinement operator implicitly supplies an *abstraction technique* that, by the syntactic nature of the refinement operator, relates system descriptions. Again, this allows infinite state systems to be considered.
- Using assertion $(*)$, correctly developing (or adapting) a system 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, reengineering 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/reengineering-technique is illustrated by Figure 1.
- As the Modal Mu-Calculus subsumes many other process logics [EL86,Dam94], we believe that our results provide a basis for similar investigations employing these logics and other semantics for concurrency.

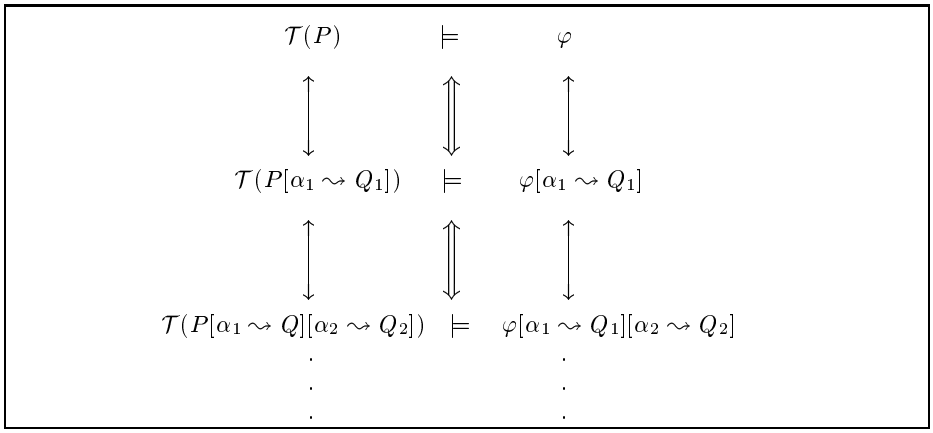

Fig. 1. Developing/Reengineering Correct Systems

If not applied in the context of developing/reengineering reactive systems, assertion $(*)$ can still be usefull to support model checking techniques for systems that could not be handled otherwise due to the (huge or infinite) size of their state spaces: If we can find a process term P_s and a formula φ_s (an abstraction of the formula φ under consideration) 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 explosion problem[1]. We can then apply the model checker to decide $P_s \models \varphi_s$

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

and conclude via assertion $(*)$ whether $P \models \varphi$ holds or not. Thus, our approach is also conceptually related to a large body of research which invertigates techniques to enhance model checking techniques (see Section 5 for an application of our abstraction technique and Section 6 for related methods).

We show how 'metalevel reasoning' (involving bisimulations and logical reasoning) can also be exploited in the development/reengineering procedure. We apply our results to a simple case study.

In Section 2 we introduce a $TCSP$-like process calculus which contains an operator for syntactic action refinement. SAR for the Modal Mu-Calculus is defined in Section 3. Section 4 provides the link between those two refinement concepts. The case study is presented in Section 5. Related work is discussed in Section 6. A summary of the results is given in Section 7. Some elementary definitions are collected in Section 8.

2 Syntactic Action Refinement in the System Model

In this section we fix the framework used to model reactive systems. Let $\alpha, \beta, \ldots$ range over a fixed set Act of *(atomic) actions*. Two languages are used to build up process terms of the form $P[\alpha \rightsquigarrow Q]$. The language $R\Delta$ supplies the terms Q whereas the language $R\Sigma$ provides the terms P. Let $R\Delta$ be the language of process terms generated by the grammar

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

and $R\Sigma$ be the language of process terms 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 x ranges over a fixed set of *identifiers*, $A \subseteq Act$ is a *synchronisation set* and $Q \in R\Delta$.

Let $sync(P)$ denote the union of all synchronisation sets that occur in P. As usual, the term 0 denotes a process that cannot perform any action. 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 1).

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 $E \notin \surd$, that is E can execute an action (see Appendix A1). A term $P \in R\Sigma$ is called *guarded* iff we have that in each subexpression $fix(x = Q)$ occurring in P the identifier x is guarded in the term Q.

A term $P_1 \in R\Sigma$ is called *alphabet-disjoint* from a term $P_2 \in R\Sigma$ iff P_1 and P_2 share no common actions.

A term $P \in R\Sigma$ is called *uniquely synchronized* iff for all terms $(P_1 \|_A P_2)$ that occur in P, $sync(P_i) = A$ for $i = 1, 2$.

To give a meaning to refined terms $P[\alpha \rightsquigarrow Q]$, we make use of a *reduction function* $red : R\Sigma \to \Sigma$ which removes all occurrences of refinement operators

in a process expression by syntactic substitution. To this end we adapt the definitions of [GGR94] for our purposes and extend it, such that the recursion operator fix can be handled (see Appendices A2 and A3). We illustrate the reduction function by the following example.

Example 1. Consider the process expression $P = ((\alpha;\beta)\|_{\{\alpha\}}\alpha)[\alpha \rightsquigarrow (\alpha_1 + \alpha_2)]$. Then we have that $red(P) = (((\alpha_1 + \alpha_2);\beta)\|_{\{\alpha_1,\alpha_2\}}(\alpha_1 + \alpha_2))$. □

The operational semantics of the language Σ is given as usual (see Appendix A4). The semantics of a process expression P is a *labelled transition system with termination*, that is, a tuple $\mathcal{T}(P) = (P, \Sigma, Act, \rightarrow, \surd)$ where $\surd$ is the termination predicate (see Appendix A1). Since the terms $P[\alpha \rightsquigarrow Q]$ and $red(P[\alpha \rightsquigarrow Q])$ are supposed to behave identically, we define $\mathcal{T}(P) := \mathcal{T}(red(P))$ to supply semantics for terms $P \in R\Sigma$ (see also [AH91]). In what follows we sometimes identify the term P with the transition system $\mathcal{T}(P)$ if the context avoids ambiguity.

Remark 1. 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 [Mil80]). The exclusion of the empty process term 0 from the language $R\Delta$ means that we disallow 'forgetful refinement'[2]. As the refinement of a (terminating) action by some infinite behaviour violates the intuition [GR99], no expression of the form $fix(x = P)$ is allowed to occur in a term $Q \in R\Delta$. □

3 Syntactic Action Refinement in the Modal Mu-Calculus

We use the *Modal Mu-Calculus* [Koz83] $\mu\mathcal{L}$ to specify properties of reactive systems. It is generated by the grammar

$$\Phi ::= \top \mid \bot \mid Z \mid (\Phi_1 \vee \Phi_2) \mid (\Phi_1 \wedge \Phi_2) \mid [\alpha]\Phi \mid \langle\alpha\rangle\Phi \mid \nu Z.\Phi \mid \mu Z.\Phi$$

where α ranges over the set Act of actions and Z ranges over a fixed set Var of *variables*. Let $R\mu\mathcal{L}$ be the language generated by the grammar for $\mu\mathcal{L}$ augmented with the rule $\Phi ::= \Phi[\alpha \rightsquigarrow Q]$ where $Q \in R\Delta$. Let $R\mu\mathcal{L}_{\langle\cdot\rangle}$ ($R\mu\mathcal{L}_{[\cdot]}$) be the language generated by the grammar for $R\mu\mathcal{L}$ without the rule $\Phi ::= [\alpha]\Phi$ ($\Phi ::= \langle\alpha\rangle\Phi$ resp.). We let σ range over the set $\{\mu, \nu\}$.

A *fixed point formula* has the form $\sigma Z.\varphi$ in which σZ *binds* free occurrences of Z in φ. A variable Z is called *free* iff it is not bound. A $R\mu\mathcal{L}$-formula φ is called *closed* iff every variable Z that occurs in φ is bound.

A $R\mu\mathcal{L}$-formula φ is called *guarded* iff every occurrence of a variable Z in φ lies in the scope of a modality $[\alpha]$ or $\langle\alpha\rangle$.

A formula $\varphi \in R\mu\mathcal{L}$ is called *alphabet-disjoint* from a term $P \in R\Sigma$ iff φ and Q share no common actions. Next we introduce a concept of logical substitution which will be used to define the reduction of formulas.

[2] Such refinements cannot be explained by a change in the level of abstraction [vGG89] and are usually avoided.

Definition 1. *Let $Q, Q_1, Q_2 \in \Delta$ and $\phi, \varphi, \psi \in \mu\mathcal{L}$. The operation of* logical substitution, *$(\phi)\{\alpha \rightsquigarrow Q\}$ is defined as follows:*

$$(*)\{\alpha \rightsquigarrow Q\} := * \quad \text{if } * \in \{\top, \bot\} \cup Var$$
$$((\varphi \odot \psi))\{\alpha \rightsquigarrow Q\} := ((\varphi)\{\alpha \rightsquigarrow Q\} \odot (\psi)\{\alpha \rightsquigarrow Q\}) \quad \text{if } \odot \in \{\wedge, \vee\}$$
$$(\triangle_\beta \varphi)\{\alpha \rightsquigarrow Q\} := \triangle_\beta(\varphi)\{\alpha \rightsquigarrow Q\} \quad \text{if } \alpha \neq \beta$$
$$(\triangle_\alpha \varphi)\{\alpha \rightsquigarrow Q\} :=$$

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

where in each clause $\triangle_\epsilon$ means throughout either $\langle\epsilon\rangle$ or $[\epsilon]$ for all $\epsilon \in Act$. We require that γ, δ are fresh actions, that is, γ and δ do neither occur in ϕ nor in Q in the term $(\phi)\{\alpha \rightsquigarrow Q\}$. □

Example 2. Let $\varphi = \mu Z.[\alpha](\langle\beta\rangle Z \vee [\alpha]\bot)$ and $Q = (\delta + \xi)$ where $\alpha, \beta, \delta, \xi \in Act$. Then

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

□

We can now define the reduction for formulas.

Definition 2. *Let $Q \in R\Delta$ be a process expression and $\varphi, \psi \in R\mu\mathcal{L}$ be formulas. We define the* logical reduction function *$\mathcal{R}ed : R\mu\mathcal{L} \to \mu\mathcal{L}$ as follows:*

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

$$\mathcal{R}ed((\varphi \odot \psi)) := (\mathcal{R}ed(\varphi) \odot \mathcal{R}ed(\psi)) \quad \text{if } \odot \in \{\wedge, \vee\}$$

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

$$\mathcal{R}ed(\sigma Z.\varphi) := \sigma Z.\mathcal{R}ed(\varphi)$$ □

To cater for refinement, we extend the usual satisfaction relation (see Appendix A5) with the clause $P \models_\vartheta \varphi[\alpha \rightsquigarrow Q]$ iff $P \models_\vartheta \mathcal{R}ed(\varphi[\alpha \rightsquigarrow Q])$. We say P *satisfies* φ (with respect to ϑ) iff $P \models_\vartheta \varphi$. For a closed $R\mu\mathcal{L}$-formula φ we simply write $P \models \varphi$.

Example 3. Let $\phi = \nu Z.([\alpha]\bot \wedge [\beta]Z)$ and $\psi = \mu Z.(\langle\alpha\rangle\top \vee \langle\beta\rangle Z)$. Then ϕ intuitively expresses the (strong) safety property 'there is no α-action executable on any β-path' and ψ expresses the (weak) liveness property 'there exists a β-path along which a state will eventually be reached at which the action α can be executed'. Let $P = fix(x = ((\beta; x); \alpha))$. Then we have $P \models \phi$ and $P \not\models \psi$. □

Example 4. Consider the process terms

$$P_1 = fix(x = ((\alpha \|_\emptyset \beta); x)) \text{ and } P_2 = fix(y = (((\alpha;\beta) + (\beta;\alpha)); y))$$

and the formula

$$\varphi = \nu Z.(\langle\alpha\rangle\langle\beta\rangle Z \wedge \langle\beta\rangle\langle\alpha\rangle Z).$$

Let $Q := \gamma[\gamma \rightsquigarrow (\alpha_1;\alpha_2)]$. Then we have $P_i \models \varphi$ and $P_i[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q]$ for $i = 1, 2$. In addition we have $P_1[\alpha \rightsquigarrow Q] \models \langle\alpha_1\rangle\langle\beta\rangle\langle\alpha_2\rangle\top$ whereas $P_2[\alpha \rightsquigarrow Q] \not\models \langle\alpha_1\rangle\langle\beta\rangle\langle\alpha_2\rangle\top$. □

4 Simultaneous Syntactic Action Refinement

In this section we provide the link between the concept of SAR for the process calculus used and SAR for the logical calculus. Let us first give the general result.

Theorem 1. *Let $P \in R\Sigma$ be a guarded process term and $\varphi \in R\mu\mathcal{L}$ be a closed and guarded formula. Further let $Q \in R\Delta$, such that P and φ are alphabet-disjoint from Q. Then $P \models \varphi \Leftrightarrow P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q]$.*

Proof (Idea). The proof can be achieved by structural induction as follows: Fixed point formulas are treated by 'syntactically unrolling' them. This leads us to the *infinitary Modal Mu-Calculus*[3] since the considered systems might be infinite state. By the condition of closedness and guardedness we obtain a well ordering, along which an argument by transfinite induction carries through. This argument uses a subsidiary induction on the structure of $Q \in R\Delta$, which in turn exploits a series of lemmata that relate the behaviour induced by a process term P with the behaviour induced by the refined term $P[\alpha \rightsquigarrow Q]$. Alphabet-disjointness of P from Q is needed to avoid the introduction and the resolvement of deadlocks through SAR. On the other hand, alphabet-disjointness of φ from Q ensures that φ remains satisfiable under SAR. ■

Remark 2. Note that the equivalence in Theorem 1 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. □

Remark 3. It is clear, that (logical) SAR as used in Theorem 1 is not complete in the sense, that we cannot derive every (interesting) formula ψ from a formula φ. We believe however, that Theorem 1 can always be useful to provide 'basic knowledge' in the overall development procedure. □

It is well known, that the Modal Mu-Calculus induces bisimulation equivalence (in the sense of [Mil80]) on the set of (finitely branching) transition systems. 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 1 we then obtain the following 'vertical modularity' result.

[3] Please consult [Sti96] for the relevant definitions.

Corollary 1. *Let $P, P' \in R\Sigma$ be guarded process terms and $\varphi \in R\mu\mathcal{L}$ be a closed and guarded formula. Let $Q \in R\Delta$, such that P and φ are alphabet-disjoint from Q. Let $[\alpha \rightsquigarrow Q]_n$ abbreviate $[\alpha_1 \rightsquigarrow Q_1], \ldots, [\alpha_n \rightsquigarrow Q_n]$. If $P \sim_b P'$ then $P[\alpha \rightsquigarrow Q]_n \models \varphi[\alpha \rightsquigarrow Q]_n \Leftrightarrow P'[\alpha \rightsquigarrow Q]_n \models \varphi[\alpha \rightsquigarrow Q]_n$.*

Corollary 1 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. 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 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 [LBC+94]). In contrary, 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 [PT87]) where M_P is the number of transitions of $\mathcal{T}(P)$. □

In Theorem 1, we can meet the conditions that P and φ are alphabet-disjoint from Q by renaming the actions of Q in the obvious way. This renaming is consistent with the usual approach to action refinement since an action 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 action that occurs in Q itself. This supports the separation of different levels of abstraction [GGR94]. Disjoint sets of actions are necessary as can be seen in the following.

Example 5. Consider the process expression $P := (\alpha \|_{\{\beta\}} \alpha)$ and the formula $\varphi := \langle\alpha\rangle\langle\alpha\rangle\top$. We have $P \models \varphi$ but $P[\alpha \rightsquigarrow \beta] \not\models \varphi[\alpha \rightsquigarrow \beta]$. Note that P is not alphabet-disjoint from Q. □

Though renaming of action can often be applied successfully, alphabet disjointness 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 resp.) 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. However, while dropping the conditions on alphabet-disjointness, we can still derive two special cases of Theorem 1. For the following let $alph(\varphi)$ be the set of actions that occur in a formula $\varphi \in R\mu\mathcal{L}$.

Theorem 2. *Let $P \in R\Sigma$ be a guarded and uniquely synchronized process term and $\varphi \in R\mu\mathcal{L}_{\langle\cdot\rangle}$ be a closed and guarded formula. Further let $Q \in R\Delta$, such that no action in $sync(P)$ occurs in Q. If $\alpha \notin sync(P)$ or $alph(\varphi) \subseteq sync(P)$ then $P \models \varphi \Rightarrow P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q]$.*

Theorem 3. *Let $P \in R\Sigma$ be a guarded and uniquely synchronized process term and $\varphi \in R\mu\mathcal{L}_{[\cdot]}$ be a closed and guarded formula. Further let $Q \in R\Delta$, such that*

no action in $sync(P)$ *occurs in* Q. *If* $\alpha \notin sync(P)$ *or* $alph(\varphi) \subseteq sync(P)$ *then* $P \models \varphi \Leftarrow P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q]$.

It is clear, that we cannot hope for a result like Theorem 1 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, i.e. $\langle\alpha\rangle$ and $[\alpha]$ without accepting any restrictions on alphabet disjointness. This is the reason why we considered the logics $R\mu\mathcal{L}_{\langle\cdot\rangle}$ and $R\mu\mathcal{L}_{[\cdot]}$ where only one modality type might occur in the formulas.

The logic $R\mu\mathcal{L}_{[\cdot]}$ can be used to express interesting properties of reactive systems, like unless-properties, for example 'φ remains true in every computation unless ψ holds' or safety properties such as 'φ never holds again whenever ψ has become true'. Moreover, $R\mu\mathcal{L}_{[\cdot]}$ can be used to express liveness-properties under fairness and cyclic-properties (see [Sti96]). $R\mu\mathcal{L}_{\langle\cdot\rangle}$-formulas can be used to formalize properties like for example 'there exists a computation sequence of P in which φ holds infinitely often' or 'there exists a computation sequence of P along which φ is always attainable.'

Whereas Theorem 2 can still be used to develop (correct) systems, the contrapositive form of Theorem 3 can be used to debug a complex (concrete) system $P[\alpha \rightsquigarrow Q]$ (with respect to $\varphi[\alpha \rightsquigarrow Q]$) by debugging the (abstract) system P with respect to φ.

5 The Case Study

While the application of Theorem 1 to develop/reengineer reactive systems can readily be seen, applying Theorem 1 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* [Mil80] or to the *IEEE Futurebus+* (considered for example in [CFJ93]) 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 approch 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 1, 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 1 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[4]

$$User_i := fix((x_i = PD_i; x_i) + (v_i^r; read_i; p_i^r; x_i) + (v_i^w; write_i; p_i^w; x_i)).$$

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

$$CorrDPE(n) = (USER^n \|_{\{v_i^r, v_i^w, read_i, write_i, p_i^r, p_i^w \mid 1 \leq i \leq n\}} CorrCONT^n).$$

$User_i$ 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 $i \neq j$. 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

$$\phi_{error}^{i,j} = \mu Z.(\langle write_i \rangle \top \wedge \langle write_j \rangle \top) \vee \langle alph(User_i) \cup alph(User_j) \rangle Z$$

[4] In the example, we sometimes omit pharenthesis in order to support readability.

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, $DPE(4) \models \phi^{1,2}_{error}$. 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 $DPE(4) \models \phi^{1,2}_{error}$ 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

$$DPE4_{small} = \Big(USER^2 \|_\emptyset SmallUser_3 \|_\emptyset SmallUser_3 \|_L$$

$$CONT^2 \|_\emptyset SmallCont_3 \|_\emptyset SmallCont_4\Big)$$

where $L = \{v^r_i, v^w_i, read_i, write_i, p^r_i, p^w_i, r_j, w_j \mid i = 1, 2 \text{ and } j = 3, 4\}$. We can then establish the refinement

$$T = DPE4_{small}[pd_3 \rightsquigarrow PD_3][r_3 \rightsquigarrow v^r_3; read_3; p^r_3][w_3 \rightsquigarrow v^w_3; write_3; p^w_3]$$

$$DPE(4) = T[pd_4 \rightsquigarrow PD_4][r_4 \rightsquigarrow v^r_4; read_4; p^r_4][w_4 \rightsquigarrow v^w_4; write_4; p^w_4].$$

Note that the formula $\phi^{1,2}_{error}$ remaines unchanged under the above refinements followed by logical reduction[5]. By Theorem 1 it suffices to model check that $DPE4_{small} \models \phi^{1,2}_{error}$ to conclude that $DPE(4) \models \phi^{1,2}_{error}$. 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:

$$CorrDPE4_{small} = \Big(USER^2 \|_\emptyset SmallUser_3 \|_\emptyset SmallUser_4 \|_L$$

$$(CorrCONT^2 + SmallCont_3 + SmallCont_4)\Big)$$

[5] Let ψ be the formula that arises by applying the above refinement operators to the formula $\phi^{1,2}_{error}$. Then $\phi^{1,2}_{error} = \mathcal{R}ed(\psi)$ whence $P \models \psi$ iff (by definition) $P \models \mathcal{R}ed(\psi)$ iff $P \models \phi^{1,2}_{error}$.

Model checking can now be used on the abstract level to show that we have $CorrDPE4_{small} \not\models \phi^{1,2}_{error}$. For

$$T' = CorrDPE4_{small}[pd_3 \rightsquigarrow PD_3][r_3 \rightsquigarrow v^r_3; read_3; p^r_3][w_3 \rightsquigarrow v^w_3; write_3; p^w_3]$$

$$CorrDPE(4) = T'[pd_4 \rightsquigarrow PD_4][r_4 \rightsquigarrow v^r_4; read_4; p^r_4][w_4 \rightsquigarrow v^w_4; write_4; p^w_4].$$

we can immediatly conclude (using Theorem 1 again) that

$$CorrDPE(4) \not\models \phi^{1,2}_{error}.$$

The example above shows, that those parts of the system description that share no actions with the formula under consideration can be immediatly abstracted. We believe that this makes precise, which parts of the system description are completely 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. We conjecture, that the above sketched strategy can be automated efficiently.

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[6]. 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 neccessary 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

$$T = DPE4_{VerySmall}[pd_1 \rightsquigarrow PD_1][pd_2 \rightsquigarrow PD_2][r_1 \rightsquigarrow read_1; p^r_1]$$

[6] We used the Edinburgh Concurrency Workbench 7.0 (see for example [Cle93]) to calculate the size of the state spaces.

then

$$DPE4_{small} = T[r_2 \rightsquigarrow read_2; p_2^r][w_1 \rightsquigarrow write_1; p_1^w][w_2 \rightsquigarrow write_2; p_2^w].$$

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 1 follows $DPE4_{small} \models \Psi_{error}$ and hence $DPE4_{small} \models \phi_{error}^{1,2}$.

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. We intend to investigate, to what extend techniques from compiler optimization (for example, exploiting common subexpressions) can support the presented method.

6 Related Work

Adressing a dual language development/reengineering-paradigm, [Huh96] showed that a *synchronisation 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: Reactive behaviour can only be modelled by infinite synchronisations structures. This does not allow to give a straightforward implementation of the involved method of semantic action refinement. Further, a linear time temporal logic is used wheres we use the branching time Modal Mu-Calculus.

If not used to develop and reengineer systems, assertion $(*)$ 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 as was illustrated by the case study in section 5. Thus, our approach is also conceptually related to a large body of research which invertigates techniques to enhance model checking techniques for huge or infinite state spaces [CAV]. *'On the fly' model checking* [SW91,BS92,Hun94,Sti95] focusses 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 [Pel98]) or *binary decision diagrams* [Bry86] 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 [CC92,Cou96]). Theorem 1 relates process terms and formulas with syntactic

refinements of them. The abstractions used in [CGL94,Gra94,BLO98,SS99] are established on the system description as well.

Syntactic action refinement allows to create hierarchical system descriptions. In [AHR98], 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 analysed 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 literatur are only conservative: Let $S^{\mathcal{A}}$ be the abstraction of the system S. Then we cannot infere $S \not\models \varphi$ from $S^{\mathcal{A}} \not\models \varphi$ if the involved abstraction is only conservative. 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 [CFJ93,ES93]). 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).

7 Conclusion

We defined syntactic action refinement (SAR) for formulas φ of the Modal Mu-Calculus and showed that the presented definition conforms to SAR for $TCSP$-like process terms P in the sense that

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

The operator $\cdot[\alpha \rightsquigarrow Q]$ denotes syntactic action refinement both on formulas and process expressions. Assertion $(*)$ is valid provided some particular conditions on alphabet-disjointness are obeyed. However, two special cases of assertion $(*)$ which do not rely upon the condition of alphabet-disjointness were presented.

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 $P[\alpha \rightsquigarrow Q] \models \varphi[\alpha \rightsquigarrow Q]$.

Further, we explained how the obtained results can be used as an abstraction technique, allowing to model check systems that would remain unfeasable otherwise.

We explained that assertion $(*)$ can be combined with model checkers. Hence, assertion $(*)$ extends this verification technique which leads to settings, that allow to automatically develop/reengineer formally correct reactive systems by hierarchically enriching/abstracting specifications with details.

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.

Further case studies are necessary to determine the practical applicability of our approach. Defining an explicit abstraction operator and investigations to what extend our abstraction technique can be fully automated are a future topic of our research. Work is already in progress that extends the above results: We study whether the conditions of alphabet-disjointness can be further relaxed and how the reduction of formulas can be determined efficiently. The consequences of introducing the 'hiding'-operator to the process algebra used will be investigated.

8 Appendix

A1. Terminated States:

To evaluate the semantics of the operator ';' it is common to use a special predicate $\surd$: Let $\surd \subseteq R\Sigma$ be the least set which contains the term 0 and is closed under the rules $(P_1 \in \surd \wedge P_2 \in \surd) \Rightarrow (P_1\ op\ P_2) \in \surd$ where $op \in \{\|_A, +, ;\}$ and $(P \in \surd) \Rightarrow fix(x = P) \in \surd$ and $(P \in \surd) \Rightarrow P[\alpha \rightsquigarrow Q] \in \surd$.

A2. Syntactic Substitution:

Let $P, P_1, P_2 \in \Sigma$ and $Q \in \Delta$ be process expressions and let $alph(Q)$ denote the set of actions that occur in Q. Syntactic substitution, denoted $(P)\{Q/\alpha\}$ is defined as follows:

$(*)\{Q/\alpha\} := *$ where $* \in \{0\} \cup Idf$

$$(\alpha)\{Q/\beta\} := \begin{cases} Q \text{ if } \alpha = \beta \\ \alpha \text{ otherwise} \end{cases}$$

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

$$((P_1 \|_A P_2))\{Q/\alpha\} := \begin{cases} ((P_1)\{Q/\alpha\} \|_{A\setminus\{\alpha\}\cup alph(Q)} (P_2)\{Q/\alpha\}) & \text{if } \alpha \in A \\ ((P_1)\{Q/\alpha\} \|_A (P_2)\{Q/\alpha\}) & \text{if } \alpha \notin A \end{cases}$$

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

A3. Reduction Function:

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

$red(*) := *$ for $* \in \{0\} \cup Idf \cup Act$

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

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

$red(fix(x = P)) := fix(x = red(P))$

A4. Operational Semantics:

Let $P, Q \in \Sigma$ be process expressions.

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

$$\frac{Q \stackrel{\alpha}{\to} Q'}{(P;Q) \stackrel{\alpha}{\to} Q'} \text{ if } P \in \surd \qquad \frac{P \stackrel{\alpha}{\to} P'}{(P;Q) \stackrel{\alpha}{\to} (P';Q)}$$

$$\frac{P \stackrel{\alpha}{\to} P'}{(P \|_A Q) \stackrel{\alpha}{\to} (P' \|_A Q)} \text{ if } \alpha \notin A \qquad \frac{Q \stackrel{\alpha}{\to} Q'}{(P \|_A Q) \stackrel{\alpha}{\to} (P \|_A Q')} \text{ if } \alpha \notin A$$

$$\frac{P[fix(x=P)/x] \stackrel{\alpha}{\to} Q}{fix(x=P) \stackrel{\alpha}{\to} Q} \qquad \frac{P \stackrel{\alpha}{\to} P' \quad Q \stackrel{\alpha}{\to} Q'}{(P \|_A Q) \stackrel{\alpha}{\to} (P' \|_A Q')} \text{ if } \alpha \in A$$

A5. Satisfaction:

Let $P \in R\Sigma$, $Q \in R\Delta$ be process expressions, $\varphi, \psi \in R\mu\mathcal{L}$ be formulas, $Z \in Var$ be a variable and $\vartheta : Var \to 2^{R\Sigma}$ be a valuation function. 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 $\vartheta'(Z) = \mathcal{E}$.

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

$P \models_\vartheta (\varphi \wedge \psi)$ iff $P \models_\vartheta \varphi$ and $P \models_\vartheta \psi$

$P \models_\vartheta (\varphi \vee \psi)$ iff $P \models_\vartheta \varphi$ or $P \models_\vartheta \psi$

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

$P \models_\vartheta \langle\alpha\rangle\varphi$ iff $P \in \{E \in R\Sigma | \exists E' \in R\Sigma (E \stackrel{\alpha}{\to} E'$ 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 $P \in \bigcup \left\{\mathcal{E} \subseteq R\Sigma | \mathcal{E} \subseteq \{E \in R\Sigma | E \models_{\vartheta[\mathcal{E}/Z]} \varphi\}\right\}$

References

[AE89] 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.

[AH91] L. Aceto and M. Hennessy. Adding action refinement to a finite process algebra. *Lecture Notes in Computer Science*, 510:506–519, 1991.

[AHR98] R. Alur, T. A. Henzinger, and S. K. Rajamani. Symbolic exploration of transition hierarchies. *Lecture Notes in Computer Science*, 1384:330–344, 1998.

[BBR90] 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.

[BLO98] 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.

[Bry86] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BS92] J. Bradfield and C. Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science*, 96(1):157–174, April 1992.

[CAV] *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.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CE81] 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.

[CFJ93] 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.

[CGL94] E. Clarke, D. Grumberg, and D. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[Cle93] 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.

[CMP87] 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.

[Cou96] P. Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324–328, June 1996.

[Dam94] M. Dam. CTL$*$ and ECTL$*$ as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, April 1994.

[EL86] 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.

[ES93] 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.

[GGR94] U. Goltz, R. Gorrieri, and A. Rensink. On syntactic and semantic action refinement. *Lecture Notes in Computer Science*, 789:385–404, 1994.

[GR99] R. Gorrieri and A. Rensink. Action refinement. Technical Report UBLCS-99-9, University of Bologna (Italy). Department of Computer Science., April 1999.

[Gra94] 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.

[Huh96] M. Huhn. Action refinement and property inheritance in systems of sequential agents. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 639–654, Pisa, Italy, 26–29 August 1996. Springer-Verlag.

[Hun94] H. Hungar. Local model checking for parallel compositions of context-free processes. *Lecture Notes in Computer Science*, 836:114–128, 1994.

[Koz83] D. Kozen. Results on the propositional mu -calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.

[LBC+94] 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.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Springer, Berlin, 1 edition, 1980.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes form temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6:68–93, 1984.

[Pel98] D. Peled. Ten years of partial order reduction. *Lecture Notes in Computer Science*, 1427:17–28, 1998.

[PR89] 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.

[PT87] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987.

[SS99] 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.

[Sti95] 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.

[Sti96] C. Stirling. Modal and temporal logics for processes. *Lecture Notes in Computer Science*, 1043:149–237, 1996.

[SW91] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, October 1991.

[vGG89] R. van Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions. In A. Kreczmar and G. Mirkowska, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science*, volume 379 of *LNCS*, pages 237–248, Berlin, August28 September–1 1989. Springer.