Abstract

HyPA is a formalism that is suitable for the algebraic analysis of hybrid systems, i.e., systems with continuous (physical) as well as discrete (computational) components. Linearization is a useful first step in this analysis, because it reduces the complexity of model descriptions by transforming them into so-called linear form. We present an algorithm for the linearization of hybrid processes modeled in a subset of hybrid process algebra (HyPA) and prove its correctness. This algorithm is able to linearize most HyPA constructs, except recursive parallelism, the empty process, and disrupts that are not a flow prefix. We also extend HyPA with an abstraction operator, which is used in the linearization algorithm.

1 Introduction

A hybrid system is a system with continuous (physical) as well as discrete (computational) components. For instance, a computer program controlling a continuous chemical process is a hybrid system. Both types of components have been studied extensively in isolation, but the interaction between them has only been the subject of research since a few years. This research has led to a number of different formalisms. Among the most well-known formalisms are hybrid automata [1], hybrid I/O automata [2] and hybrid Petri nets [3]. Various process algebra based approaches exist as well, such as Process Algebra for Hybrid Systems [4], hybrid $\chi$ [5] and HyPA [6,7].

Hybrid process algebras attempt to extend the knowledge and experience of the field of process algebra to the field of hybrid systems. This is usually achieved by taking an existing (discrete or timed) process algebra and extending it with elements for modeling continuous behavior. The result is a hybrid formalism that fully supports algebraic reasoning. Moreover, these formalisms...
provide compositionality of all the operators, including the parallel composition. Of the three hybrid process algebras mentioned above, HyPA was chosen for this work. HyPA is a conservative extension of the discrete process algebra ACP [8], with the disrupt operator from LOTOS [9] and with clauses [10] for the description of continuous behavior and discontinuities.

Hybrid process algebras allow a modeler to construct intuitive models of complex hybrid systems and to reason about these models. However, many analysis techniques take (a description of) a state space as input, but it is far from trivial to generate the state space of a given model. Furthermore, compositionality requires an equivalence that is robust under an arbitrary context. This equivalence is necessarily very strong. Certain analysis techniques, such as the safety analysis of [11], cannot be used under such a strong equivalence.

In this paper, we present an algorithm for the linearization of hybrid processes described in the hybrid process algebra HyPA. Linearization in process algebra is a transformation of a (recursive) specification, or model description, into a symbolic representation of the state space. This symbolic representation is expressed as a recursive specification as well, but it uses only a small subset of the full process algebra. Such a specification is said to be linear or in linear form. There are several advantages to this approach. First, it is fairly straightforward to generate the actual state space of a system if its specification is in linear form. Second, the linear form is convenient for exchanging specifications between tools because of its simple structure. Finally, a weaker notion of equivalence can be used on linear specifications, namely the one that is compositional only for the restricted set of operators used in linear specifications.

Linearization was first described in the context of the discrete process algebra $\mu$CRL [12]. We used the linearization algorithm of $\mu$CRL [13,14] as a starting point for our own algorithm. $\mu$CRL and the process algebraic part of HyPA are very similar, so there are many similarities between the linearization algorithms. However, $\mu$CRL does not have any hybrid features, so we adapted the algorithm to incorporate HyPA’s hybrid aspects. On the other hand, HyPA does not have an equivalent of $\mu$CRL’s alternative quantification (denoted $\sum_{d \in D}$ with a possibly infinite domain $D$). This sum operator is essential for linearizing specifications with large or infinite state spaces, so we introduce an operator for abstraction of model variables that can take over the role of this $\mu$CRL operator. Abstraction of model variables is indeed a valuable addition to HyPA in itself.

The structure of this paper is as follows. In Section 2, HyPA is presented. In Section 3, an abstraction operator for model variables is introduced. In Section 4, the linearization algorithm is presented and its correctness is proven. In Section 5, some optimizations are discussed and in Section 6 an example is
given. Finally, in Section 7, we conclude with a discussion of the results and possible directions for future research.

2 Hybrid Process Algebra

In this section the syntax and semantics of HyPA are discussed. The discussion presented here is adapted from [11]. A more detailed explanation of HyPA can be found in [6].

2.1 Syntax

The syntax of HyPA is an extension of the process algebra ACP [8], with the disrupt operator from LOTOS [9] and with variants of the flow clauses and re-initialization clauses from the event-flow formalism introduced in [10]. The signature of HyPA consists of the following constant and function symbols:

(1) deadlock $\delta$,
(2) empty process $\epsilon$,
(3) discrete actions $a \in A$,
(4) flow clauses $c \in C$,
(5) a family of process re-initialization operators $d \gg$, where $d \in D$,
(6) alternative composition $- \oplus -$,
(7) sequential composition $- \odot -$,
(8) disrupt $- \triangleright -$ and left-disrupt $- \triangleright -$,
(9) parallel composition $- \parallel -$, left-parallel composition $- \parallel -$ and forced-synchronization $- | -$,
(10) a family of encapsulation operators $\partial_H (.)$ where $H \subseteq A$.

The binding order of these operators is as follows: $\odot$, $\triangleright$, $\triangleright$, $d \gg$, $\parallel$, $\parallel$, $|$, $\oplus$, where sequential composition binds strongest and alternative composition binds weakest. These constants and operators are described informally below.

**Deadlock, empty process and discrete actions** The atomic processes $\delta$ (called *deadlock*) and $\epsilon$ (called *empty process*) are used to model a deadlocking process and a (successfully) terminating process, respectively. The atomic *discrete actions* are used as an abstract model for discrete, computational behavior.
Flow clauses  Flow clauses are used to model continuous, never terminating, 
physical behavior by describing how the model variables \( \mathcal{V}_m \) are allowed to 
change through time. A flow clause is a pair \((V \mid P_f)\) of a set of model variables 
\( V \subseteq \mathcal{V}_m \) and a flow predicate \( P_f \in \mathcal{P}_f \). The set \( V \) models which variables are 
not allowed to jump at the beginning of a flow.

The predicate \( P_f \) models the continuous behavior. More precisely, \( P_f \) describes 
a set of flows, where a flow is a (partial) function of time \( T \) to the valuations 
\( \text{Val} \) of model variables. \( T \) has a closed-interval domain starting from 0, and 
\( \text{Val} \) is the set of variable valuations \( \mathcal{V}_m \rightarrow \mathcal{V} \), where \( \mathcal{V} \) is the union of all 
variable domains and is defined as \( \mathcal{V} = \bigcup_{x \in \mathcal{V}_m} \mathcal{V}(x) \). The set of all flows is 
\( \mathcal{F} = \{ f \in T \mapsto \text{Val} \mid \text{dom}(f) = [0, t] \text{ for some } t \in T \} \). The flows that are 
described by a flow predicate are called the solutions of that predicate.

The set of predicates \( \mathcal{P}_f \), the sets \( \mathcal{V}_m \) of model variables and \( T \) of time points, 
and the notion of solution \( \models_f \subseteq \mathcal{F} \times \mathcal{P}_f \) are parameters of the theory. This 
means that the modeler can choose an appropriate method to describe flows, 
for example with differential equations, integrals, algebraic inequalities, etc. 
Finally, the set of all flow clauses is closed under conjunction (\( \land \)) and it is 
assumed that there is a flow predicate \( \text{false} \in \mathcal{P}_f \), which has no solutions.

Re-initializations  A process re-initialization \( d \succ p \) models the behavior 
of \( p \) where the model variables are submitted to a discontinuous change as 
specified by the re-initialization clause \( d \). A re-initialization clause describes a 
set of re-initializations, where a re-initialization is a pair of valuations represent-
ing the values of the model variables prior to and immediately after the 
re-initialization. The set of all re-initializations \( \text{Val} \times \text{Val} \) is denoted \( \mathcal{R} \).

A re-initialization clause is a pair \([V \mid P_r]\) of a set of model variables \( V \subseteq \mathcal{V}_m \) 
and a re-initialization predicate \( P_r \in \mathcal{P}_r \). The set \( V \) models which variables are 
allowed to change. Note that this is precisely opposite to flow clauses, where \( V \) 
denotes those variables that do not change (initially). Predicate \( P_r \) models the 
discontinuous changes. In a predicate, \( x^- \) denotes the valuation of a variable 
\( x \) before re-initialization, and \( x^+ \) denotes the valuation of a variable \( x \) after 
re-initialization. If \( x \) is not allowed to change, ie. \( x \notin V \), we may choose to 
write \( x \) instead of \( x^- \) in a predicate. As with flow clauses, the set of predicates 
\( \mathcal{P}_r \) and the notion of solution \( \models_r \subseteq \mathcal{R} \times \mathcal{P}_r \) are parameters of the theory.

It is assumed that there is a flow predicate \( \text{true} \in \mathcal{P}_r \), which satisfies all 
re-initializations in \( \mathcal{R} \), and a predicate \( \text{false} \in \mathcal{P}_r \), which satisfies no re-
initializations in \( \mathcal{R} \). The set of all re-initialization clauses is closed under con-
junction (\( \land \)), disjunction (\( \lor \)) and concatenation (\( \sim \)). Furthermore, there is a 
satisfiability operator (\( d^? \)) on re-initialization clauses \( d \), which does not change 
the valuation of any model variable, but only executes the re-initialized pro-
cess if $d$ can be satisfied in some way. Finally, there is a re-initialization clause $(c_{\text{jmp}})$ derived from a flow clause $c$, which executes the same discontinuities that are allowed initially by the flow clause $c$. These last two operators are mainly used for calculating with process terms.

**Alternative and sequential composition** The alternative composition $p \oplus q$ models a (non-deterministic) choice between the processes $p$ and $q$. The sequential composition $p \odot q$ models a sequential execution of processes $p$ and $q$. The process $q$ is executed after (successful) termination of the process $p$.

**Disrupt and left-disrupt** The disrupt $p \triangleright q$ models a kind of sequential composition where the process $q$ may take over execution from process $p$ at any moment, without waiting for its termination. This composition is essential for modeling two flow clauses executing one after the other, since the behavior of flow clauses never terminates. The left-disrupt $p \triangleright q$ is mainly needed for calculation and axiomatization purposes, rather than for modeling purposes. It first executes a part of the process $p$ and then behaves as a normal disrupt.

**Parallel composition** The parallel composition $p \parallel q$ models concurrent execution of $p$ and $q$. The intuition behind this concurrent execution is that discrete actions are executed in an interleaving manner, with the possibility of synchronization of actions (as in ACP, where synchronization is called communication), while flow clauses are forced to synchronize, and can only synchronize if they accept the same solutions. The synchronization of actions takes place using a (partial, commutative and associative) communication function $\gamma \in \mathcal{A} \times \mathcal{A} \mapsto \mathcal{A}$. For example, if the actions $a$ and $a'$ synchronize, then the resulting action is $a'' = a\gamma a'$. Actions cannot synchronize with flow clauses, and in a parallel composition between those, the action executes first. Re-initializations synchronize only if the processes on which they act synchronize.

As with the left-disrupt, the left-parallel and forced-synchronization operators are mainly introduced for calculation purposes. The left-parallel composition $p \parallel q$ denotes that $p$ performs a discrete action first (if possible), and then behaves as a normal parallel composition. The forced-synchronization $p \mid q$ denotes how the first behavior (either a discrete action or a part of a flow) of $p$ and $q$ is synchronized, after which it behaves as a normal parallel composition as well.

**Encapsulation** Encapsulation $\partial_H(p)$ models that the discrete actions from the set $H \subseteq \mathcal{A}$ are blocked during the execution of process $p$. This opera-
tor is often used in combination with the parallel composition to model that synchronization between discrete actions is enforced.

**Recursion** Terms can be constructed using variables from a given set of process variables $V_p$ (with $V_p \cap V_m = \emptyset$), as usual. The set of all such terms is denoted $T(V_p)$ and these are referred to as terms or open terms. Terms in which no process variables occur are called closed terms. The set of all closed terms is denoted $T$. Finally, all processes should be interpreted in the light of a set $E$ of recursive definitions of the form $X \approx p$, where $X$ is a process variable and $p$ is a term.

### 2.2 Semantics

The formal semantics of HyPA is defined in terms of a Hybrid Transition System. Such a transition system has two different kinds of transitions, namely one associated with computational behavior (i.e. discrete actions), and the other associated with physical behavior (i.e. flow clauses).

**Definition 1 (Hybrid Transition System)** A hybrid transition system is a tuple $\langle X, A, \Sigma, \mapsto, \sim, \checkmark \rangle$, consisting of a state space $X$, a set of action labels $A$, a set of flow labels $\Sigma$, and transition relations $\mapsto \subseteq X \times A \times X$ and $\sim \subseteq X \times \Sigma \times X$. Lastly, there is a termination predicate $\checkmark \subseteq X$.

For the semantical hybrid transition systems that are associated with HyPA terms, the state space is formed by pairs of process terms and valuations of the model variables, i.e. $X = T(V_p) \times Val$. The set of action labels is formed by pairs of actions and valuations, i.e. $A = A \times Val$, and the set of flow labels is formed by the set of flows, i.e. $\Sigma = F$. Recall that the elements $f \in F$ have a closed-interval domain, possibly a singleton, starting in 0.

The notation $\langle x \rangle \mapsto (x', a)$ is used for a transition $(x, a, x') \in \mapsto$ with $x, x' \in X$ and $a \in A$. Similarly, $\langle x \rangle \sim (x', \sigma)$ is used for a transition $(x, \sigma, x') \in \sim$ with $\sigma \in \Sigma$, and for arbitrary transitions, $\langle x \rangle \overset{l}{\mapsto} (x')$ is used instead of $(x, l, x') \in \mapsto \cup \sim$ and $l \in A \cup F$. Finally, termination is denoted $\langle x \rangle \checkmark$ instead of $x \in \checkmark$.

First, the definition of a solution of a flow clause and a re-initialization clause is given. Then, the semantics of the HyPA constants and function symbols is given, using deduction rules in the style of [15]. See [6] for a detailed explanation of the semantics.

**Definition 2 (Solution of a flow clause)** A pair $(\nu, \sigma) \in Val \times F$, is defined to be a solution of a flow clause $c \in C$, denoted $(\nu, \sigma) \models c$, as follows:
• \( (\nu, \sigma) \models (V|P_f) \) if \( \sigma \models fP_f \), and for all \( x \in V \) we find \( \nu(x) = \sigma(0)(x) \);
• \( (\nu, \sigma) \models c \land c' \) if \( (\nu, \sigma) \models c \) and \( (\nu, \sigma) \models c' \).

**Definition 3 (Solution of a re-initialization clause)** A re-initialization
\( (\nu, \nu') \in \mathcal{R} \) is defined to be a solution of a re-initialization clause \( d \in D \),
denoted \( (\nu, \nu') \models d \), as follows:

• \( (\nu, \nu') \models [V|P_r] \) if \( (\nu, \nu') \models_\pi P_r \) and for all \( x \not\in V \) we find \( \nu(x) = \nu'(x) \);
• \( (\nu, \nu') \models d' \lor d'' \) if \( (\nu, \nu') \models d' \) or \( (\nu, \nu') \models d'' \);
• \( (\nu, \nu') \models d' \land d'' \) if \( (\nu, \nu') \models d' \) and \( (\nu, \nu') \models d'' \);
• \( (\nu, \nu') \models d' \sim d'' \) if there exists \( v \in Val \) with \( (\nu, v) \models d' \) and \( (\nu, v') \models d'' \);
• \( (\nu, \nu') \models d'' \) if \( \nu = \nu' \), and there exists \( v \in Val \) with \( (\nu, v) \models d' \);
• \( (\nu, \nu') \models c_{\text{jump}} \) if there exists \( \sigma \in \Sigma \) such that \( (\nu, \sigma) \models c \) and \( \sigma(0) = \nu' \).

In the Tables 1, 2 and 3, \( p, p', q, q' \) denote process terms, \( a, a', a'' \) denote actions,
\( c \) denotes a flow clause, \( d \) denotes a re-initialization clause, \( H \) denotes a set of
actions, \( X \) denotes a recursion variable, \( \nu, \nu', \nu'' \) denote valuations, \( \sigma \) denotes
a flow, \( t \) denotes a point in time, and \( l \) denotes an arbitrary transition label.

**Table 1**
Operational semantics of HyPA

\[
\begin{align*}
\langle \epsilon, \nu \rangle \equiv & \quad (1) \\
\langle a, \nu \rangle \xrightarrow{a,v} \langle \epsilon, \nu \rangle \quad (2) \\
\langle \nu, \sigma \rangle \models c, \quad \text{dom}(\sigma) = [0, t] \quad (3) \\
\lambda \nu, \nu' \models d, \quad \langle \nu, \nu' \rangle \models \lambda \nu, \nu' \models d, \\
\langle \nu, \nu' \rangle \models d, \quad \langle \nu, \nu' \rangle \equiv & \quad (4) \\
\langle \nu, \nu' \rangle \models d, \quad \langle \nu, \nu' \rangle \models d, \quad \langle \nu, \nu' \rangle \equiv & \quad (5) \\
\langle p, \nu \rangle \equiv & \quad (6) \\
\langle p \oplus q, \nu \rangle \equiv & \quad (7) \\
\langle q \oplus p, \nu \rangle \equiv & \quad (8) \\
\langle p \oplus q, \nu \rangle \equiv & \quad (9) \\
\langle q \oplus p, \nu \rangle \equiv & \quad (10) \\
\end{align*}
\]

2.3 Robust Bisimilarity and Axiomatization

In this section the equivalence notion of robust bisimilarity is presented. First,
the notion of bisimilarity of hybrid transition systems is defined. This notion
is then lifted to process terms.
Table 2
Operational semantics of HyPA, disrupt and left-disrupt

\[ \langle p, \nu \rangle \checkmark \quad (11) \]
\[ \langle p \triangleright q, \nu \rangle \checkmark \quad (12) \]
\[ \langle p \triangleright q, \nu \rangle \checkmark \quad (13) \]
\[ \langle q, \nu \rangle \checkmark \quad (14) \]

Definition 4 (Bisimilarity on hybrid transition systems)
Let \( \langle X, A, \Sigma, \rightarrow, \sim, \checkmark \rangle \) be a hybrid transition system. A relation \( R \subseteq X \times X \) on the state space, is a bisimulation relation if

- for all \( x, y \in X \) such that \( x R y \), we find \( \langle x \rangle \checkmark \) implies \( \langle y \rangle \checkmark \);
- for all \( x, y \in X \) such that \( x R y \), we find \( \langle y \rangle \checkmark \) implies \( \langle x \rangle \checkmark \);
- for all \( x, x', y \in X \) such that \( x R y \) and \( l \in A \cup \Sigma \), we find \( \langle x \rangle \overset{l}{\rightarrow} \langle x' \rangle \) implies there exists \( y' \) such that \( \langle y \rangle \overset{l}{\rightarrow} \langle y' \rangle \) and \( x' R y' \);
- for all \( x, y, y' \in X \) such that \( x R y \) and \( l \in A \cup \Sigma \), we find \( \langle y \rangle \overset{l}{\rightarrow} \langle y' \rangle \) implies there exists \( x' \) such that \( \langle x \rangle \overset{l}{\rightarrow} \langle x' \rangle \) and \( x' R y' \).

Two states \( x, y \in X \) are bisimilar, notation \( x \leftrightarrow y \), if there exists a bisimulation relation that relates \( x \) and \( y \).

In HyPA, model variables are shared by all processes executing in parallel. Therefore, a process can cause interference with another (parallel) process through these shared variables. In order for the equivalence to be robust with respect to this interference, it is required that process terms are related for all valuations that can be obtained through interference. An interference can be modeled as a function \( \iota : Val \rightarrow Val \).

Definition 5 (Robust) A relation \( R \subseteq (T(V_p) \times Val) \times (T(V_p) \times Val) \) is called robust if for all \( \langle p, \nu \rangle, \langle p', \nu' \rangle \in X \) such that \( \langle p, \nu \rangle R \langle p', \nu' \rangle \) we find \( \langle p, \iota(\nu) \rangle R \langle p', \iota(\nu') \rangle \) for all interferences \( \iota \in Val \rightarrow Val \).

Definition 6 (Robust bisimilarity) Two process terms \( p, q \in T(V_p) \) are robustly bisimilar, denoted \( p \leftrightarrow_r q \), if there exists a robust bisimulation relation \( R \subseteq (T(V_p) \times Val) \times (T(V_p) \times Val) \) such that \( \langle p, \nu \rangle R \langle q, \nu \rangle \) for all valuations \( \nu \in Val \).

If two process terms are robustly bisimilar then they describe equivalent transition systems, hence they describe the same process. The axioms of HyPA are
Table 3
Operational semantics of HyPA, parallel composition, encapsulation and recursion

\[
\begin{align*}
\langle p, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} \langle q, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} (15) & \quad \langle p, \nu \rangle \overset{\sigma}{\rightsquigarrow} \langle p', \nu' \rangle, \langle q, \nu \rangle \overset{\sigma}{\rightsquigarrow} \langle q', \nu' \rangle \quad (16) \\
\langle p \parallel q, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} & \quad \langle p \parallel q, \nu \rangle \overset{\sigma}{\rightsquigarrow} \langle p' \parallel q', \nu' \rangle \\
\langle p \mid q, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} & \quad \langle p \mid q, \nu \rangle \overset{\sigma}{\rightsquigarrow} \langle p' \parallel q', \nu' \rangle \\
\langle p, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} \langle p', \nu' \rangle, \langle q, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} & \quad \langle p, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} \langle p', \nu' \rangle \quad (17) \\
\langle p \parallel q, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} & \quad \langle p \parallel q, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} \langle p' \parallel q', \nu' \rangle \\
\langle q \parallel p, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} & \quad \langle q \parallel p, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} \langle q' \parallel p', \nu' \rangle \\
\langle p \mid q, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} & \quad \langle p \mid q, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} \langle p' \parallel q', \nu' \rangle \\
\langle q \mid p, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} & \quad \langle q \mid p, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} \langle q' \parallel p', \nu' \rangle
\end{align*}
\]

\[
\begin{align*}
\langle p, \nu \rangle \overset{a \gamma a'}{\rightsquigarrow} \langle p', \nu'' \rangle, \langle q, \nu \rangle \overset{a \gamma a'}{\rightsquigarrow} \langle q', \nu''' \rangle, \quad a'' = a \gamma a' & \quad (18) \\
\langle p \parallel q, \nu \rangle \overset{a \gamma a'}{\rightsquigarrow} \langle p' \parallel q', \nu'' \rangle \\
\langle p \parallel q, \nu \rangle \overset{a \gamma a'}{\rightsquigarrow} \langle p' \parallel q', \nu'' \rangle \\
\langle p \mid q, \nu \rangle \overset{a \gamma a'}{\rightsquigarrow} \langle p' \parallel q', \nu'' \rangle \\
\langle q \mid p, \nu \rangle \overset{a \gamma a'}{\rightsquigarrow} \langle q' \parallel p', \nu'' \rangle \\
\langle p, \nu \rangle \overset{a \gamma a'}{\rightsquigarrow} \langle p', \nu'' \rangle, \quad a \notin H & \quad (20) \\
\langle \partial_H (p), \nu \rangle \overset{a \gamma a'}{\rightsquigarrow} \langle \partial_H (p'), \nu'' \rangle \\
\langle \partial_H (p), \nu \rangle \overset{a \gamma a'}{\rightsquigarrow} \langle \partial_H (p'), \nu'' \rangle & \quad (21) \\
\langle p, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} \langle p', \nu' \rangle & \quad (22) \\
\langle \partial_H (p), \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} \langle \partial_H (p'), \nu' \rangle
\end{align*}
\]

\[
\begin{align*}
\langle p, \nu \rangle \overset{\check{\cdot}}{\rightsquigarrow} X & \approx p \in E \quad \langle p, \nu \rangle \overset{\rightarrow}{\rightarrow} \langle p', \nu' \rangle \quad (23) \\
\langle X, \nu \rangle \overset{\rightarrow}{\rightarrow} \langle p', \nu' \rangle & \quad (24)
\end{align*}
\]

presented in Table 4 and Table 5 below. The derivation rules of HyPA that define in which way equivalences can be derived from the axioms are given in [6,11]. In each of the axioms, \(x, y, z\) denote arbitrary terms. The letters \(a, a'\) denote actions, while \(c, c'\) denote flow clauses and \(d, d'\) denote re-initialization clauses. Unlike what is usual for ACP, one may not choose \(\delta\) when \(a\) is written in an axiom.

These axioms are sound with respect to robust bisimilarity and enable equational reasoning for the analysis of hybrid transition systems. Furthermore, using these axioms it is possible to rewrite a closed term into a normal form
and it can be shown that HyPA is a conservative extension of the process algebra ACP [7,6].

2.4 Example

In this section an example is given to illustrate HyPA’s syntax. This example is a slightly modified version of the temperature controller in [16]. In Figure 1, a hybrid automaton model for this example is given.

The temperature in a room is controlled by a thermostat, which continuously monitors the temperature and turns a heater on and off. The thermostat has to keep the temperature $x$ in the interval $[x_{\text{min}}, x_{\text{max}}]$. When the heater is turned off, the temperature decreases according to the differential equation...
Table 5
Axioms of HyPA, continued

<table>
<thead>
<tr>
<th>Axiom 1</th>
<th>Axiom 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d \gg x \oplus d' \gg x \approx (d \lor d') \gg x$</td>
<td>$(d \gg a) \circ x \approx d \gg a \circ x$</td>
</tr>
<tr>
<td>$d \gg (x \oplus y) \approx d \gg x \oplus d \gg y$</td>
<td>$(d \gg \epsilon) \circ x \approx d' \gg x$</td>
</tr>
<tr>
<td>$(d \gg x) \triangleright y \approx d \gg x \triangleright y$</td>
<td>$d \gg x \parallel y \approx d \gg (x \parallel y)$</td>
</tr>
<tr>
<td>$d \gg (d' \gg x) \approx (d \sim d') \gg x$</td>
<td></td>
</tr>
<tr>
<td>$\partial_H (x \oplus y) \approx \partial_H (x) \oplus \partial_H (y)$</td>
<td>$\partial_H (x \circ y) \approx \partial_H (x) \circ \partial_H (y)$</td>
</tr>
<tr>
<td>$\partial_H (x \triangleright y) \approx \partial_H (x) \triangleright \partial_H (y)$</td>
<td>$\partial_H (d \gg x) \approx d \gg \partial_H (x)$</td>
</tr>
<tr>
<td>$\partial_H (c) \approx c$</td>
<td>$\partial_H (\epsilon) \approx \epsilon$</td>
</tr>
<tr>
<td>$\partial_H (a) \approx a$ if $a \notin H$</td>
<td>$\partial_H (a) \approx \delta$ if $a \in H$</td>
</tr>
</tbody>
</table>

$\dot{x} = -x$. When the heater is turned on, the temperature increases according to the differential equation $\dot{x} = -x + 4$. Initially, the heater is on.

**HeaterOn** $\approx (x | \dot{x} = -x + 4) \leftarrow rOff \circ HeaterOff$

**HeaterOff** $\approx (x | \dot{x} = -x) \leftarrow rOn \circ HeaterOn$

**Thermostat** $\approx (x_{min} \leq x \leq x_{max}) \leftarrow$

$$
\begin{cases}
  [x = x_{min}] \gg sOn \circ Thermostat \\
  \oplus \\
  [x = x_{max}] \gg sOff \circ Thermostat
\end{cases}
$$

11
Controller $\approx \partial_H (\text{HeaterOn} \parallel \text{Thermostat})$

\begin{align*}
\gamma(s_{\text{On}}, r_{\text{On}}) &= \text{On} \\
\gamma(s_{\text{Off}}, r_{\text{Off}}) &= \text{Off} \\
H &= \{s_{\text{Off}}, s_{\text{On}}, r_{\text{Off}}, r_{\text{On}}\}
\end{align*}

3 Abstraction of Model Variables

3.1 Syntax

HyPA is extended with an abstraction operator, which is denoted $[ V \mid p ]$. Here, $V$ is a (possibly infinite) set of model variables, namely the variables to abstract from. Furthermore, $p$ is an arbitrary HyPA process term, in which variables in $V$ can occur as well as variables defined outside the scope of this abstraction operator (including the global model variables). Intuitively, the effect of abstracting from a certain variable is that this variable is not visible on the ‘outside’, i.e. the variable and its valuation cannot be observed externally.
We often write \([ v_1, \ldots, v_n \mid p ]\) and \([ v \mid p ]\) instead of \([ \{ v_1, \ldots, v_n \} \mid p ]\) and \([ \{ v \} \mid p ]\), respectively.

Note that it is possible to use variable names in \(V\) that are already used outside the scope of this abstraction. For example, suppose there is a global model variable named \(v\) and \([ v \mid p ]\) is used. In this case, all occurrences of the name \(v\) in \(p\) bind to the abstracted variable \(v\), not to the global variable \(v\). The valuation of the abstracted variable \(v\) is not visible externally, but the valuation of the global variable \(v\) is visible.

3.2 Semantics

The semantics of the abstraction operator is defined using deduction rules in the style of [15]. Please refer to [6,7] for a detailed description of the semantics of HyPA and the underlying hybrid transition system.

First, an auxiliary operator \([ [ V : \nu \mid p ] ]\) is defined. The extra variable \(\nu\) denotes the valuation of the variables in \(V\), i.e. the local state, and its domain is exactly \(V\). This auxiliary operator is needed to define the semantics of the normal abstraction operator \([ [ V \mid p ] ]\). It allows us to specify exactly what happens with the valuations of the abstracted variables and which valuations are visible in the underlying transition system. In principle it is possible to give axioms for this operator and use it in HyPA specifications, but so far this does not seem to be useful, especially for the purpose of linearization.

### Table 6
Operational semantics of the abstraction operator

\[
\begin{align*}
\langle p, m_V(\mu, \nu) \rangle & \mapsto \langle [ [ V : \nu \mid p ] ], \mu \rangle \quad (1) \\
\langle [ [ V : \nu \mid p ] ], \mu \rangle & \mapsto \langle [ [ V : \nu \mid p ] ], \mu \rangle \quad (4)
\end{align*}
\]

\[
\begin{align*}
\langle [ [ V : \nu \mid p ] ], \mu \rangle & \mapsto \langle p, m_V(\mu, \nu) \rangle \quad (2) \\
\langle p, m_V(\mu, \nu) \rangle & \mapsto \langle p', w' \rangle \\
\langle [ [ V : \nu \mid p ] ], \mu \rangle & \mapsto \langle [ [ V : w' \mid V \mid p' ] ], m_V(w', \mu) \rangle \\
\langle [ [ V : \nu \mid p ] ], \mu \rangle & \mapsto \langle [ [ V : w' \mid V \mid p' ] ], m_V(w', \sigma'([\top])) \rangle \\
\langle [ [ V : \nu \mid p ] ], \mu \rangle & \mapsto \langle p', w' \rangle \quad (3)
\end{align*}
\]

\[
\begin{align*}
\langle [ [ V : \nu \mid p ] ], \mu \rangle & \mapsto \langle p', w' \rangle \quad (5) \\
\langle [ [ V : \nu \mid p ] ], \mu \rangle & \mapsto \langle [ [ V : \nu \mid p ] ], \mu \rangle \quad (6)
\end{align*}
\]

In Table 6, the semantics of both types of abstraction operators is given. To
keep the rules concise, the auxiliary functions $m_V(\mu, \nu)$ and $m_V(\sigma, \sigma')$ are used, which merge two valuations and two flows, respectively. The first function takes the valuations of the variables in the set $V$ from $\nu$ and the valuations of all other variables from $\mu$. Similarly, the second one takes the flows of the variables in the set $V$ from $\sigma'$ and the flows of all other variables from $\sigma$. Intuitively, this works like opening a new variable scope in a (procedural) programming language, because the newly introduced variables ‘hide’ existing variables with the same name. For any $\mu, \nu \in \text{Val}$, $\sigma, \sigma' \in \Sigma$ and $V \subseteq \mathcal{V}_m$:

$$m_V(\mu, \nu)(n) = \begin{cases} 
\mu(n) & \text{if } n \notin V \\
\nu(n) & \text{if } n \in V 
\end{cases}$$

$$m_V(\sigma, \sigma')(n) = \begin{cases} 
\sigma(n) & \text{if } n \notin V \\
\sigma'(n) & \text{if } n \in V 
\end{cases}$$

Furthermore, $\nu|_V$ denotes the valuation where $\text{dom}(\nu|_V) = V$ and $\nu|_V(n) = \nu(n)$ for all $n \in V$. Finally, $\sigma'(|_1)$ denotes the valuation of the flows in $\sigma'$ in the last element of its domain.

Rule (1) states that the abstraction of a terminating process can also terminate. Rule (2) describes the case for a (discrete) action transition. This rule expresses an essential point of the abstraction operator, namely that the valuations of abstracted variables are not visible in the transition system. That is why the arrow in the conclusion is labeled with $m_V(w, \mu)$, instead of simply $w$. Furthermore, the semantics is chosen such that the valuation of ‘hidden’ model variables does not change during an action transition. The reason for this choice is that in the existing semantics of HyPA, the valuations of model variables also do not change during an action transition. Rule (3) describes the case for a (continuous) flow transition. This rule is similar to rule (2). Note however that the valuation in the resulting state is equal to the last valuation of the flow. Again, the reason for this choice is that in the existing semantics of HyPA, this is also the case. Rules (4) to (6) define the actual abstraction operator, in terms of the auxiliary abstraction operator. Note that the local state variable $\nu$ in the hypotheses of these rules is an arbitrary valuation whose domain is $V$.

**Theorem 7** Robust bisimilarity is a congruence for both the auxiliary abstraction operator $\llbracket V: \nu \mid p \rrbracket$ and the abstraction operator $\llbracket V \mid p \rrbracket$. Hence, if $p \sim_r q$, then $\llbracket V: \nu \mid p \rrbracket \sim_r \llbracket V: \nu \mid q \rrbracket$ and $\llbracket V \mid p \rrbracket \sim_r \llbracket V \mid q \rrbracket$ for all process terms $p$ and $q$, $V \subseteq \mathcal{V}_m$ and $\nu \in \text{Val}$.

**Proof.** This is straightforward to verify using the congruence formats in [17,18].
3.3 Axioms

A (partial) axiomatization is given in Table 7. In these axioms, \( \text{Var}(x) \) denotes the set of free variables in the term \( x \). Table 7 only gives some basic axioms and some axioms that are necessary for the linearization algorithm.

Table 7

<table>
<thead>
<tr>
<th>Axioms for the abstraction operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>([ V \mid x ] \oplus [ V \mid y ] \approx [ V \mid x \oplus y ])</td>
</tr>
<tr>
<td>([ V \mid x ] \odot [ V \mid y ] \approx [ V \mid x \odot [V\mid\text{true}] \gg y ])</td>
</tr>
<tr>
<td>([ V \mid x ] \gg [ V \mid y ] \approx [ V \mid x \gg [V\mid\text{true}] \gg y ])</td>
</tr>
<tr>
<td>([ V \mid \partial_H (x) ] \approx \partial_H ([ V \mid x ]))</td>
</tr>
<tr>
<td>([ V \mid [ W \mid x ] ] \approx [ V \cup W \mid x ])</td>
</tr>
<tr>
<td>([ V \mid x ] \approx x), if ( \text{Var}(x) \cap \emptyset = \emptyset )</td>
</tr>
<tr>
<td>([ V \mid x ] \parallel y \approx [ V \mid x \parallel y ]), if ( \text{Var}(y) \cap \emptyset = \emptyset )</td>
</tr>
<tr>
<td>(d \gg [ V \mid x ] \approx [ V \mid d \gg x ]), if ( \text{Var}(d) \cap \emptyset = \emptyset )</td>
</tr>
<tr>
<td>([ v \mid x ] \approx [ w \mid x[w/v] ]), if ( w \notin \text{Var}(x) )</td>
</tr>
</tbody>
</table>

Axioms (VA1), (VA2), (VA3) and (VA4) all express distribution of the abstraction operator over other operators. Together, they describe how abstraction can be distributed over (closed) linear terms. Axiom (VA5) describes how two abstractions can be merged. Axiom (VA6) states that abstracting from variables that do not occur freely in the abstracted term has no influence. Both of these axioms are useful for introducing abstractions or for eliminating them.

Axioms (VA7) and (VA8) expresses that a parallel term or re-initialization clause can be pulled into the abstraction, as long as the abstracted variables do not occur freely in that term or clause. These axioms appeal to the same intuition as axiom (VA6). However, they cannot simply be derived from (VA6), because abstraction does not distribute over parallel composition or re-initialization in general. Axiom (VA9) states that abstracted variables can be renamed (\(\alpha\)-conversion). Note that in this axiom \( v \) and \( w \) denote single variables, not sets of variables. The expression \( x[w/v] \) denotes the substitution of \( w \) for all free occurrences of \( v \) in the process term \( x \).

**Theorem 8** All axioms in Table 7 are sound.

**Proof.** The proof of this theorem can be found in Appendix A of [19].
4 Linearization

This section describes an algorithm for linearization of (a large subset of) HyPA. First, the specification of a general HyPA linearization algorithm is given. Second, the input restrictions of our particular algorithm are defined and the specification of a general linearization algorithm is adapted to these input restrictions. Third, the linearization algorithm is described informally and the formal translation functions are presented and the fact that they preserve equivalence is proven.

4.1 Specification of a General HyPA Linearization Algorithm

In this section, first the notions recursive specification and linear recursive specification are defined. Then, the specification of a general HyPA linearization algorithm is stated.

Definition 9 (Recursive Specification) A recursive specification consists of an open term $t$, called the initial term, and a finite set of recursive equations $E$ that define the variables in $t$. Such a specification is denoted $\langle t \mid E \rangle$. A recursive equation for a (recursion) variable $X$ is defined as $X \approx s(\text{Var}(E))$, meaning that the right-hand side of such an equation is an open term $s$ that only contains variables that are defined in the specification. There is exactly one recursive equation for every recursion variable.

Definition 10 (Solutions of a Recursive Specification) A solution of a recursive specification $\langle t \mid E \rangle$ is a collection of processes that can be substituted for the recursion variables, such that all recursive equations in $E$ become true statements. The variables of the initial term $t$ are then interpreted as their solutions in $E$. For example, a solution of the recursive equation $X \approx b \circ X$ is the process $b \circ b \circ b \circ \ldots$. A solution of the specification $\langle a \circ X \mid \{ X \approx b \circ X \} \rangle$ is then $a \circ (b \circ b \circ b \circ \ldots)$.

Definition 11 (Linear Recursive Specification) A linear recursive specification $\langle t \mid E \rangle$ is a recursive specification that satisfies the following requirements:

1. the initial term $t$ is either a recursion variable in $E$ (i.e. $X$) or the abstraction of a re-initialization on such a variable (i.e. $\parallel V \mid d \gg X \parallel$), and
2. the right-hand sides of all recursive equations in $E$ are linear terms.
A term is linear, if it has the form\(^1\)

\[
p ::= \delta \mid d \gg a \mid d \gg a \odot X \mid d \gg c \triangleright X \mid p \oplus p
\]

and the right-hand sides of all recursion variables occurring in the term are linear as well.

Now, a general HyPA linearization algorithm is specified as follows:

**Specification 1** A HyPA linearization algorithm transforms a recursive specification \(<t \mid E>\) into a linear recursive specification \(<t' \mid E'>\) such that \(<t \mid E>\) and \(<t' \mid E'>\) are robustly bisimilar.

### 4.2 Input Restrictions

The linearization algorithm presented in this paper has some restrictions on the input specifications. First some definitions are stated, which are then used to express these restrictions.

**Definition 12 (Guarded process term)** An open process term \(p\) is guarded if all occurrences of recursion variables in \(p\) are in the scope of an action prefix \(a \odot _-\) or a flow prefix \(c \triangleright _-\).

**Definition 13 (Guarded recursive specification)** A recursive specification \(<t \mid E>\) is guarded if the right-hand sides of all equations in \(E\) are guarded, or can be transformed into guarded terms by replacing variables by the right-hand side of their equation\(^2\).

Note that the Recursive Specification Principle (RSP) and Recursive Definition Principle (RDP) from [6,11] together state that guarded recursive specifications in HyPA have a unique solution. However, when extending HyPA with abstraction from variables, multiple solutions may be introduced due to so-called unbounded non-determinism (see [6,20]). In our specifications, this unbounded non-determinism does not occur, because the changes in the abstracted variables are always uniquely defined. Hence, the guarded recursive specifications we use still have a unique solution.

---

\(^1\) Forms of process terms are denoted in the familiar BNF notation. Recall that \(X\) denotes a recursion variable, \(a\) is a discrete action, \(c\) is a flow clause and \(d\) is a re-initialization clause.

\(^2\) The usual definition of guardedness states that a recursive specification is also guarded if it can be transformed into a guarded recursive specification by applying axioms. In this paper however, specifications that need to be transformed are not considered to be guarded, otherwise correctness of our algorithm cannot be proven.
Definition 14 (HyPA\textsubscript{par} Form) The HyPA\textsubscript{par} form is defined as the form $p$:

- $p ::= \left[ V \mid q \right] \mid q$
- $q ::= X \mid q \parallel q \mid \partial_H(q)$

Definition 15 (HyPA\textsubscript{lin} Form) The HyPA\textsubscript{lin} form is defined as follows:

- $p ::= a \mid X \mid \delta \mid c \mid p \oplus p \mid p \odot p \mid d \gg p \mid c \triangleright p \mid c \triangleleft p$

Definition 16 (HyPA\textsubscript{lin} Specification) A HyPA\textsubscript{lin} specification is a recursive specification $\langle t \mid E \rangle$ that satisfies the following three restrictions:

1. $\langle t \mid E \rangle$ is guarded;
2. $t$ is in HyPA\textsubscript{par} form, and
3. the right-hand sides of all recursive equations in $E$ are in HyPA\textsubscript{lin} form.

Note that a HyPA\textsubscript{lin} specification is expressive enough to model hybrid automata [1] and constitutive processes [21]. Although we do not prove this, it is straightforward to transform a hybrid automata into a HyPA\textsubscript{lin} specification. Furthermore, constitutive processes already satisfy the HyPA\textsubscript{lin} restrictions without transforming them.

Specification 2 The linearization algorithm that is presented in this paper transforms a HyPA\textsubscript{lin} specification $\langle t \mid E \rangle$ into a linear recursive specification $\langle t' \mid E' \rangle$, such that $\langle t \mid E \rangle$ and $\langle t' \mid E' \rangle$ are robustly bisimilar.

Most restrictions are made in order to avoid fundamental problems. First, the parallel composition is restricted in such a way that there is no recursion over the parallel composition, as in $X \approx X \parallel Y$ for instance. In such a case, there are in fact infinitely many parallel compositions, so trying to eliminate them one by one does not work.

Second, the abstraction operator is not allowed in the HyPA\textsubscript{lin} form, because it is not possible to eliminate abstraction of open terms from recursive equations. Third, the empty process ($\epsilon$) cannot be used, because it leads to some problems in the transformations.

Finally, only single flow clauses can be disrupted. On the one hand, this is a consequence of not allowing the use of the empty process, because the empty process is needed when there are actions in the left argument of a disrupt operator. For instance, eliminating the disrupt operator from $a \triangleright x$ yields $a \odot (\epsilon \oplus x) \oplus x$ and the empty process can not be eliminated from this term.

3 In the linearization algorithm of $\mu$CRL a similar restriction on the use of both the parallel composition and the encapsulation operator was made at first. The algorithm for $\mu$CRL was later extended to relax this restriction, but it became much more complex. We decided to take the same approach and keep the restriction for now.
in general. On the other hand, recursion in the left argument of the disrupt operator gives problems similar to the problems with recursion in the scope of parallel composition. Every time a disrupt operator is eliminated, another may be introduced. For instance, $X \approx (a \odot X) \triangleright y \approx (a \odot X) \triangleright y \ominus y \approx a \odot (X \triangleright y) \oplus y$. Clearly, trying to eliminate the disrupt operators one by one does not work in this case.

However, a few of the restrictions are only made to keep the description of the algorithm concise. First, as noted above, only a single flow clause is allowed in the left argument of a disrupt operator. It is straightforward to allow alternative and sequential composition of flow clauses, re-initializations, disrupt operators and deadlock ($\delta$) here, since the problem is the use of actions and recursion variables. Second, only one abstraction operator is allowed in the HyPA$_{par}$ form. Third, the encapsulation operator is not allowed in the HyPA$_{lin}$ form. Finally, the left-parallel composition and forced-synchronization are not allowed at all, mainly because they are not meant to be used directly in specifications. The first three restrictions are easily relaxed and could be implemented as some sort of pre-processing step to the algorithm described in this paper. The last one can partly be relaxed easily as well, but it gives the same problems with recursion as the parallel composition.

### 4.3 Linearization Algorithm

The algorithm consists of three consecutive stages. The first two stages deal with the recursive equations in the input specification and the third stage deals with the initial term. In Sections 4.3.1 to 4.3.3, each of these three stages is described in detail. Our aim is to present a basic algorithm, and possible optimizations are deferred to Section 5. In this section, correctness of our algorithm is proven as well, which is captured in the following theorem:

**Theorem 17** *The linearization algorithm satisfies Specification 2.*

**PROOF.** This theorem leads to three proof obligations. First, the algorithm has to be *sound*, which means that all transformation steps transform the specification into another specification whose initial terms are robustly bisimilar. Second, the *result* of the algorithm has to be a linear recursive specification. Finally, the algorithm has to be *well-defined*.

In the following, it is proved for each of the three stages that they are sound and that they are well-defined. Therefore, the algorithm as a whole is sound and well-defined.
In the following sections it is also proved that the result of the algorithm is a linear recursive specification. The input of the very first step is a HyPA_{lin} specification. Each step is proven to lead to a specification of a certain intermediate form, and this intermediate form is then the input for the next step. Finally, the very last step (T_1) is shown to lead to a recursive specification of linear form.

Therefore, the linearization algorithm presented in this paper satisfies Specification 2.

To make this paper more readable, the proofs for all following theorems and lemmas have been placed in Appendix A.

### 4.3.1 Stage 1: Transforming Equations into Semi-linear Form

In this first stage, several transformation functions are applied to the right-hand sides of all recursive equations in the specification. These transformation functions are applied in the order in which they are described below.

The form of the right-hand sides after this first stage is called *semi-linear*, which is defined as follows:

\[
\begin{align*}
p & ::= \delta \mid d \gg a \mid d \gg a \odot q \mid d \gg c \triangleright q \mid p \oplus p \\
q & ::= X \mid q \odot q
\end{align*}
\]

The only difference with the linear form is the fact that an action \(a\) or a flow clause \(c\) can be followed by the sequential composition of multiple recursion variables \((q)\), instead of a single variable \((X)\) only.

**Simple Rewriting**  This first step eliminates the disrupt operator \((\triangleright)\) and distributes the alternative composition \((\oplus)\) over sequential composition and re-initialization. This is done by rewriting the right-hand sides of all recursive equations in \(E\) using the rewrite system consisting of the following rewrite rules:

\[
\begin{align*}
x \triangleright y & \rightarrow x \triangleright y \oplus y \quad \text{(R1)} \\
(x \oplus y) \odot z & \rightarrow x \odot z \oplus y \odot z \quad \text{(R2)} \\
d \gg (x \oplus y) & \rightarrow d \gg x \oplus d \gg y \quad \text{(R3)}
\end{align*}
\]

Recall that the right-hand sides of all recursive equations in \(E\) are of the form HyPA_{lin}:

\[
p ::= a \mid X \mid \delta \mid c \mid p \oplus p \mid p \odot p \mid d \gg p \mid c \triangleright p \mid c \triangleright p
\]
Lemma 18 (Resulting form) After applying this rewrite system to the right-hand sides of all equations in \( E \), the right-hand sides of all equations in \( E \) are of the form \( r \):

\[
\begin{align*}
  r & := a \mid X \mid \delta \mid c \mid d \gg r_2 \mid r_2 \odot r \mid c \triangleright r \mid r \oplus r \\
  r_2 & := a \mid X \mid \delta \mid c \mid d \gg r_2 \mid r_2 \odot r \mid c \triangleright r
\end{align*}
\]

Lemma 19 (Soundness) This rewrite step is sound.

Lemma 20 (Termination) This rewrite system is terminating.

Adding New Recursive Equations This second step reduces the complexity of the terms further, by introducing new recursive equations. The right-hand sides become terms that are the alternative composition of sub terms (summands) that do not contain alternative composition themselves. Furthermore, they either have a single left disrupt operator or a number of sequential compositions, but not both.

The right-hand sides of all recursive equations in \( E \) are transformed with the function \( S_1 \). The function \( S_1 \) is not applied to the right-hand sides of newly introduced equations, except when this is stated explicitly. The introduction of a new recursive equation is denoted by adding it to the set of recursive equations \( E \). Note that it is always assumed that the left-hand side of this new equation is a fresh variable.

\[
\begin{align*}
  S_1(a) &= a \\
  S_1(X) &= X \\
  S_1(\delta) &= \delta \\
  S_1(c) &= X, \text{ and } E := E \cup \{X \approx c \triangleright Y, Y \approx \delta\} \\
  S_1(d \gg p) &= d \gg S_1(p) \\
  S_1(p \odot q) &= S_2(p \odot q) \\
  S_1(c \triangleright p) &= S_2(c \triangleright p) \\
  S_1(p \oplus q) &= S_1(p) \oplus S_1(q)
\end{align*}
\]
\[ S_2(a) = a \]
\[ S_2(X) = X \]
\[ S_2(\delta) = \delta \]
\[ S_2(c) = X, \text{ and } E := E \cup \{X \approx c \triangleright Y, Y \approx \delta\} \]
\[ S_2(d \gg p) = d \gg S_2(p) \]
\[ S_2(p \odot q) = X, \text{ and } E := E \cup \{X \approx S_2(p) \odot S_2(q)\} \]
\[ S_2(c \triangleright p) = X, \text{ and } E := E \cup \{X \approx c \triangleright Y, Y \approx S_1(p)\} \]
\[ S_2(p \oplus q) = X, \text{ and } E := E \cup \{X \approx S_1(p) \oplus S_1(q)\} \]

**Lemma 21 (Resulting form)** After applying \( S_1 \) to the right-hand sides of all equations in \( E \), the right-hand sides of all equations in \( E \) are of the form \( s \):

\[
\begin{align*}
s & := a \mid X \mid \delta \mid d \gg s_2 \mid s_2 \odot s_2 \mid c \triangleright X \mid s \oplus s \\
s_2 & := a \mid X \mid \delta \mid d \gg s_2
\end{align*}
\]

**Lemma 22 (Soundness)** \( S_1 \) is sound, i.e. \( S_1(p) \approx p \), for any term \( p \) of the form \( r \).

**Lemma 23 (Well-definedness)** \( S_1 \) is well-defined.

**Guarding** This third step transforms all right-hand sides of all recursive equations in \( E \) into guarded terms. Recall that the definition of guardedness states that a term is guarded if all recursion variables occur in the scope of an action prefix \( a \odot \_ \) or a flow prefix \( c \triangleright \_ \). However, a recursive specification is guarded if the right-hand sides of all equations are guarded, or can be transformed into guarded terms by replacing variables by the right-hand side of their equation. This means that, although the specification is guarded, the right-hand sides of the recursive equations may not be guarded.

The right-hand sides of all recursive equations in \( E \) are transformed with the
function \textit{guard}:

\[
\begin{align*}
guard(a) &= a \\
guard(X) &= guard(rhs(X)) \\
guard(\delta) &= \delta \\
guard(d \gg p) &= rew(rhs(d \gg \guard(p))) \\
guard(p \odot q) &= rew(\guard(p) \odot q) \\
guard(c \triangleright p) &= c \triangleright p \\
guard(p \oplus q) &= guard(p) \oplus guard(q)
\end{align*}
\]

where \(rhs(X)\) denotes the right-hand side of the recursive equation of \(X\) in \(E\), and \(rew\) is the rewrite system consisting of the following rules:

\[
\begin{align*}
(x \oplus y) \odot z &\rightarrow x \odot z \oplus y \odot z \quad (R10) \\
d \gg (x \oplus y) &\rightarrow d \gg x \oplus d \gg y \quad (R11) \\
(c \triangleright x) \odot y &\rightarrow c \triangleright (x \odot y) \quad (R12) \\
d \gg \delta &\rightarrow \delta \quad (R14) \\
\delta \odot x &\rightarrow \delta \quad (R15) \\
(d \gg (c \triangleright x)) \odot y &\rightarrow d \gg c \triangleright (x \odot y) \quad (R16) \\
d \gg (d' \gg x) &\rightarrow (d \sim d') \gg x \quad (R17) \\
(x \odot y) \odot z &\rightarrow x \odot (y \odot z) \quad (R18) \\
(d \gg (a \odot x)) \odot y &\rightarrow d \gg (a \odot (x \odot y)) \quad (R19) \\
d \gg ((d' \gg a) \odot x) &\rightarrow (d \sim d') \gg (a \odot x) \quad (R20)
\end{align*}
\]

\textbf{Lemma 24 (Resulting form)} After applying \textit{guard} to the right-hand sides of all equations in \(E\), the right-hand sides of all equations in \(E\) are of the form \(g\):

\[
\begin{align*}
g &::= g \oplus g \mid \delta \mid a \mid a \odot g_2 \mid c \triangleright g_2 \mid d \gg a \mid \\
g_2 &::= \delta \mid a \mid X \mid g_2 \odot g_2 \mid d \gg a \mid d \gg X
\end{align*}
\]

\textbf{Lemma 25 (Soundness)} \textit{guard} is sound.

\textbf{Lemma 26 (Well-definedness)} \textit{guard} is well-defined.
Post Processing  The right-hand sides of the recursive equations are almost semi-linear now. In this fourth and final step, the equations only need to be cleaned up somewhat. This step consists of two consecutive transformation functions, $P_1$ and $P_3$. In $P_1$, new equations are introduced. This is done in a similar fashion as in the $S_1$ function and $P_1$ is not applied to these new equations. After this post processing step, all recursive equations in $E$ are semi-linear. First, the function $P_1$ is applied to the right-hand sides of all recursive equations in $E$:

$$P_1(p \oplus q) = P_1(p) \oplus P_1(q)$$
$$P_1(\delta) = \delta$$
$$P_1(a) = \lbrack \text{true} \rbrack \gg a$$
$$P_1(a \odot p) = \lbrack \text{true} \rbrack \gg (a \odot P_2(p))$$
$$P_1(c \gg p) = \lbrack \text{true} \rbrack \gg (c \gg P_2(p))$$
$$P_1(d \gg a) = d \gg a$$
$$P_1(d \gg (a \odot p)) = d \gg (a \odot P_2(p))$$
$$P_1((d \gg a) \odot p) = d \gg (a \odot P_2(p))$$
$$P_1(d \gg (c \gg p)) = d \gg (c \gg P_2(p))$$

Second, the function $P_3$ is applied to the right-hand sides of all recursive equations in $E$:

$$P_2(\delta) = Y, \text{ and } E := E \cup \{Y \approx \delta\}$$
$$P_2(a) = Y, \text{ and } E := E \cup \{Y \approx \lbrack \text{true} \rbrack \gg a\}$$
$$P_2(X) = X$$
$$P_2(p \odot q) = P_2(p) \odot P_2(q)$$
$$P_2(d \gg a) = Y, \text{ and } E := E \cup \{Y \approx d \gg a\}$$
$$P_2(d \gg X) = Y, \text{ and } E := E \cup \{Y \approx d \gg X\}$$

Second, the function $P_3$ is applied to the right-hand sides of all recursive equations in $E$:

$$P_3(p) = \begin{cases} 
\text{rewr}(d \gg \text{rhs}(X)) & \text{if } p \text{ of the form } d \gg X \text{ for any re-init. clause } d \text{ and recursion variable } X \\
p & \text{otherwise}
\end{cases}$$

24
where \( \text{rewr} \) is the rewrite system consisting of the following rules:

\[
\begin{align*}
  d \gg (x \oplus y) & \rightarrow d \gg x \oplus d \gg y \quad (\text{R11}) \\
  d \gg \delta & \rightarrow \delta \quad (\text{R14}) \\
  d \gg (d' \gg x) & \rightarrow (d \sim d') \gg x \quad (\text{R17})
\end{align*}
\]

The goal of these transformations is to make the right-hand sides of all equations in \( E \) semi-linear. Recall that a term is semi-linear if it is of the following form, for brevity called \( l \) here:

\[
\begin{align*}
  l & ::= \delta \mid d \gg a \mid d \gg a \odot l_2 \mid d \gg c \triangleright l_2 \mid l \oplus l \\
  l_2 & ::= X \mid l_2 \odot l_2
\end{align*}
\]

**Lemma 27 (Resulting form)** After first applying \( P_1 \) and then applying \( P_3 \) to the right-hand sides of all equations in \( E \), the right-hand sides of all equations in \( E \) are semi-linear.

**Lemma 28 (Soundness)** \( P_1 \) and \( P_3 \) are sound.

**Lemma 29 (Well-definedness)** \( P_1 \) and \( P_3 \) are well-defined.

### 4.3.2 Stage 2: From Semi-linear to Abstracted-linear

In this stage, the semi-linear equations are transformed to a form where only a single recursion variable occurs after an action or a flow clause. This transformation introduces a stack of (a representation of) recursion variables. The idea is that this stack is a kind of to-do list. When a sequential composition of multiple recursion variables (i.e. \( X_0 \odot \ldots \odot X_n \)) is encountered, they are all pushed onto the stack and the process modeled by the recursion variable on top of the stack starts executing (i.e. \( X_0 \)). As soon as this process terminates, execution is resumed with the next process on the stack (if any). This stack is represented by a fresh model variable. This variable is abstracted from, because it should not be visible externally.

Currently, the right-hand sides of all recursive equations are in semi-linear form. Another way to denote this form is the following:

\[
\begin{align*}
  X_i \approx \bigoplus_{j \in J(i)} d_j \gg a_j \oplus \\
  \bigoplus_{k \in K(i)} d_k \gg (a_k \odot (X_{f(k,1)} \odot \ldots \odot X_{f(k,n)})) \oplus \\
  \bigoplus_{l \in L(i)} d_l \gg (c_l \triangleright (X_{f(l,1)} \odot \ldots \odot X_{f(l,n')}))
\end{align*}
\]
The $\oplus$-notation is used as a shorthand for the alternative composition of a finite number of terms. If the domain of this quantifier is empty, it is deadlock ($\delta$). The index $i$ ranges from 1 to the number of recursive equations in the specification. The sets $J(i)$, $K(i)$ and $L(i)$ are disjoint for all such $i$.

The transformation goes as follows. First, a single new recursive equation $A$ is defined, which captures the contents of all equations in $E$. $A$ is defined as follows (note that $A$ is linear):

$$A \approx \bigoplus_{X_i \in \text{Var}(E)} \left( \bigoplus_{j \in J(i)} \left[ s^- \neq \emptyset \land \text{get}(s^-) = X_i \land \text{pop}(s^-) = \emptyset \right] \sim d_j \gg a_j \right)$$

$$\bigoplus_{j \in J(i)} \left[ s \neq \emptyset \land \text{get}(s^-) = X_i \land \text{pop}(s^-) \neq \emptyset \land s^+ = \text{pop}(s^-) \right] \sim d_j \gg a_j \odot A$$

$$\bigoplus_{k \in K(i)} \left[ s^- \neq \emptyset \land \text{get}(s^-) = X_i \land s^+ = \text{push}(X_{f(k,1)}, \ldots, \text{push}(X_{f(k,n_k)}, \text{pop}(s^-)) \ldots) \right] \sim d_k \gg a_k \odot A$$

$$\bigoplus_{l \in L(i)} \left[ s^- \neq \emptyset \land \text{get}(s^-) = X_i \land s^+ = \text{push}(X_{f(l,1)}, \ldots, \text{push}(X_{f(l,n_l)}, \text{pop}(s^-)) \ldots) \right] \sim d_l \gg \left( c_l \land \left( s \mid s = 0 \right) \right) \triangleright A$$

where the operations on a stack variable $s$ are defined as follows:

- $\emptyset$ Represents the empty stack.
- $\text{push}(i, s)$ Returns the stack $s$ with element $i$ pushed onto it.
- $\text{pop}(s)$ Returns the stack $s$ without the top element.
- $\text{get}(s)$ Returns the top element of the stack $s$ (but doesn’t pop it).

Second, all recursive equations in $E$ are transformed, except for the newly introduced equation $A$, as follows:

For all $X_i \in E \setminus \{A\}$, $X_i \approx \left[ s \mid s^+ = \text{push}(X_i, \emptyset) \right] \gg A$
Because of the abstraction, the right-hand sides of the equations are not transformed into linear form, but into a form called abstracted-linear. A term is in abstracted-linear form if it has the form \([V \mid d \gg X]\), and the right-hand side of the variable \(X\) is linear. The result of the transformation in this stage is that the right-hand sides of all recursive equations in \(E\) are abstracted-linear, except for the single linear recursive equation \(A\).

**Lemma 30 (Resulting form)** After applying the transformation in this stage, the right-hand sides of all equations in \(E\) are abstracted-linear, except one equation, whose right-hand side is linear.

**Lemma 31 (Soundness)** The transformation in this stage is sound.

**Lemma 32 (Well-definedness)** The transformation in this stage is well-defined.

### 4.3.3 Stage 3: Transforming the Initial Term

In this final stage, the initial term \(t\) is transformed to the form \([V \mid d \gg X]\), where \(X\) is linear. The transformation is based on two operations. The first operation computes the parallel composition of two recursion variables whose right-hand side is abstracted-linear. The result is a single recursion variable whose right-hand side is also abstracted-linear. The second operation computes the encapsulation of a variable whose right-hand side is abstracted-linear and the result is another single recursion variable whose right-hand side is abstracted-linear as well. The transformation function \(T_1\) below uses these two operations to eliminate all parallel composition and encapsulation operators from \(t\), so the initial term becomes an abstraction of a single variable whose right-hand side is abstracted-linear.

The function \(T_1\) is applied to the initial term \(t\) and is defined as follows:

\[
T_1(p) = \begin{cases} 
[V \mid T_2(q)] & \text{if } p \text{ is of the form } [V \mid q] \\
[\emptyset \mid T_2(p)] & \text{otherwise}
\end{cases}
\]

\[
T_2(X) = X \\
T_2(p \parallel q) = \text{elim\_par\_comp}(T_2(p), T_2(q)) \\
T_2(\partial_H(p)) = \text{elim\_encap}(H, T_2(p))
\]

The result of \(T_2(p)\) is always a single recursion variable, say \(X\), whose right-hand side is abstracted-linear. Therefore, the result of \(T_1(p)\) is always an abstraction of \(X\), i.e. \([V \mid X]\).
To obtain the final linear recursive specification, this final transformation is performed on the initial term. Suppose $X \approx [ W \mid d \gg A ]$, where $A$ is linear, then:

$$t = [ V \mid X ] \approx [ V \mid [ W \mid d \gg A ] ] \approx [ V \cup W \mid d \gg A ]$$

The specification $(t \mid \{A\})$ is taken as the resulting specification. This resulting specification is linear, so the linearization algorithm is finished.

First, the operations for the elimination of parallel composition and encapsulation are introduced and the corresponding correctness proofs are given. Then, the correctness of $T_1$ is considered.

**Eliminating Parallel Composition** The $\text{elim\_par\_comp}$ function takes two recursion variables whose right-hand sides are abstracted-linear, and it returns a single recursion variable whose right-hand side is abstracted-linear as well. The returned recursion variable is equivalent to the parallel composition of the two arguments.

Suppose $X$ and $Y$ are abstracted-linear, and the parallel composition in $X \parallel Y$ needs to be eliminated. From the previous stage it is known that all recursive equations in $E$ are abstracted-linear, except for the single linear recursive equation $A$. The functions $T_1$, $\text{elim\_par\_comp}$ and $\text{elim\_encap}$ maintain this property, so $X$ and $Y$ both refer to the same linear equation $A$. Therefore, $X$ and $Y$ have the following form:

$$X \approx [ S \mid dX \gg A ]$$
$$Y \approx [ T \mid dY \gg A ]$$
$$A \approx \bigoplus_{j \in J(A)} d_j \gg a_j \oplus \bigoplus_{k \in K(A)} d_k \gg a_k \ominus A \oplus \bigoplus_{l \in L(A)} d_l \gg c_l \uplus A$$

All equations in the set of equations $E$ of the specification at this point are abstracted-linear, except for the linear equation for $A$. We denote a linear term that contains only the variable $A$ as $\text{lin}(A)$ and we denote an abstracted-linear term that contains only the variable $A$ as $\text{abslin}(A)$. Suppose $P_0, \ldots, P_n$ are all the recursion variables defined in $E$ except $A$, $X$ and $Y$. Then, before eliminating parallel composition:

$$E = \begin{cases} A \approx \text{lin}(A), X \approx \text{abslin}(A), Y \approx \text{abslin}(A), \\ P_0 \approx \text{abslin}(A), \ldots, P_n \approx \text{abslin}(A) \end{cases}$$

The following steps are performed in the order they are listed:
(1) First, by applying the renaming axiom for abstraction, all variables in that are in both S and T are renamed, in order to make S and T disjoint. This is done by applying axioms (VA5) and (VA9) multiple times and it leads to a fresh equation, say B, in which these variables are renamed. Y then refers to B instead of A. B is added to E.

\[
\begin{align*}
X & \approx \left[ S \mid d_X \gg A \right] & Y & \approx \left[ T \mid d_Y \gg B \right] \\
A & \approx \bigoplus_{j \in J(A)} d_j \gg a_j \oplus & B & \approx \bigoplus_{j \in J(B)} d_j \gg a_j \oplus \\
& \quad \bigoplus_{k \in K(A)} d_k \gg a_k \ominus A \ominus & & \quad \bigoplus_{k \in K(B)} d_k \gg a_k \ominus B \ominus \\
& \quad \bigoplus_{l \in L(A)} d_l \gg c_l \triangleright A & & \quad \bigoplus_{l \in L(B)} d_l \gg c_l \triangleright B
\end{align*}
\]

Now

\[
E = \left\{ A \approx \text{lin}(A), B \approx \text{lin}(B), X \approx \text{abslin}(A), Y \approx \text{abslin}(B), \right. \\
\left. P_0 \approx \text{abslin}(A), \ldots, P_n \approx \text{abslin}(A) \right\}
\]

(2) Now, the parallel composition of X and Y is computed, which gives the abstracted-linear recursive equation Z and the linear equations Z_1 to Z_{11,l}. Please see Appendix B for the resulting equations. These new equations are not added to E.

(3) Now, there are several linear recursive equations, namely A, B and Z_1 to Z_{11,l}. Since a linear recursive equation is also a semi-linear recursive equation, these linear equations can be transformed to abstracted-linear equations using the transformation of the previous stage. The result of this transformation is that A, B and Z_1 to Z_{11,l} are now abstracted-linear. They are of the form \([ v \mid d \gg C ]\), where v is a fresh variable and C is a linear recursive equation.

\[
E = \left\{ A \approx \text{abslin}(C), B \approx \text{abslin}(C), \\
X \approx \left[ S_X \mid d_X \gg A \right], Y \approx \left[ S_Y \mid d_Y \gg B \right], \\
P_0 \approx \left[ S_{P_0} \mid d_{P_0} \gg A \right], \ldots, P_n \approx \left[ S_{P_n} \mid d_{P_n} \gg A \right] \right\}
\]

and C \approx \text{lin}(C), Z \approx \left[ S_Z \mid d_Z \gg Z_1 \right], Z_1 \approx \text{abslin}(C), \ldots, Z_{11,l} \approx \text{abslin}(C).

(4) The recursive equations in E are not abstracted-linear anymore, because A and B are now abstracted-linear. Moreover, Z is not abstracted-linear anymore, because Z_1 is now abstracted-linear as well. This is solved by applying the following transformation once to all recursive equations in E (except A and B) and to Z. Suppose X \approx \left[ S_X \mid d_X \gg A \right] and
\( A \approx [ V \mid d \gg C ] : \)

\[
X \approx [ S_X \mid d_X \gg A ] \\
\approx [ S_X \mid [ V \mid d \gg C ] ] \\
\approx [ S_X \mid [ V \mid d_X \gg \gg C ] ] \\
\approx [ S_X \cup V \mid (d_X \sim d) \gg C ]
\]

Now

\[
E = \left\{ \begin{array}{l}
A \approx \text{abslin}(C), B \approx \text{abslin}(C), X \approx \text{abslin}(C), Y \approx \text{abslin}(C), \\
P_0 \approx \text{abslin}(C), \ldots, P_n \approx \text{abslin}(C)
\end{array} \right\}
\]

and \( C \approx \text{lin}(C), Z \approx \text{abslin}(C), Z_1 \approx \text{abslin}(C), \ldots, Z_{11,l} \approx \text{abslin}(C) \).

(5) Finally, \( Z \) and \( C \) are added to \( E \), and \( A \) and \( B \) are deleted from \( E \) because they are not used anymore. Now, all recursive equations in \( E \) are abstracted-linear, except for the single equation \( C \) which is linear. Now

\[
E = \left\{ \begin{array}{l}
C \approx \text{lin}(C), X \approx \text{abslin}(C), Y \approx \text{abslin}(C), Z \approx \text{abslin}(C), \\
P_0 \approx \text{abslin}(C), \ldots, P_n \approx \text{abslin}(C)
\end{array} \right\}
\]

The result of the \( \text{elim}_{\text{par}}\_\text{comp} \) function is the single recursion variable \( Z \).

**Lemma 33 (Resulting form)** The result of transformation \( \text{elim}_{\text{par}}\_\text{comp} \) is a single abstracted-linear recursion variable. Furthermore, after transformation, all recursive equations in \( E \) are abstracted-linear except for the single equation \( C \) which is linear.

**Lemma 34 (Soundness)** The transformation \( \text{elim}_{\text{par}}\_\text{comp} \) is sound and \( \text{elim}_{\text{par}}\_\text{comp}(X, Y) \approx X \parallel Y \) for abstracted-linear recursion variables \( X \) and \( Y \).

**Lemma 35 (Well-definedness)** \( \text{elim}_{\text{par}}\_\text{comp} \) is well-defined.

**Eliminating Encapsulation** The \( \text{elim}_{\text{encap}} \) transformation takes a set of discrete actions \( H \) and an abstracted-linear recursion variable \( X \) as its parameters. The result is a single abstracted-linear recursion variable \( Z \) that is equivalent to \( \partial_H(X) \).

Suppose \( X \) is abstracted-linear, and the encapsulation in \( \partial_H(X) \) needs to be eliminated. Then, \( X \) has the following form, where \( A \) is a linear recursive
As in the elimination of parallel composition, all equations in the set of equations $E$ at this point are abstracted-linear, except for the linear equation for $A$. Therefore, before eliminating encapsulation:

$$E = \begin{cases} A \approx \text{lin}(A), X \approx \text{abslin}(A), \\
P_0 \approx \text{abslin}(A), \ldots, P_n \approx \text{abslin}(A) \end{cases}$$

The following steps are performed in the order they are listed:

(1) Two new recursive equations $Z$ and $B$ are introduced and added to $E$:

$$E := E \cup \{ Z \approx \lbrack S \mid d_X \gg A \rbrack, B \approx \text{elim\_encap}_2(H, \text{rhs}(A)) \}$$

where the $\text{elim\_encap}_2$ function is defined as follows:

$$\begin{align*}
\text{elim\_encap}_2(H, p \oplus q) &= \text{elim\_encap}_2(p) \oplus \text{elim\_encap}_2(q) \\
\text{elim\_encap}_2(H, d \gg a) &= \begin{cases} \delta & \text{if } a \in H \\
d \gg a & \text{if } a \notin H \end{cases} \\
\text{elim\_encap}_2(H, d \gg a \odot A) &= \begin{cases} \delta & \text{if } a \in H \\
d \gg a \odot B & \text{if } a \notin H \end{cases} \\
\text{elim\_encap}_2(H, d \gg c \triangleright A) &= d \gg c \triangleright B \\
\text{elim\_encap}_2(H, \delta) &= \delta
\end{align*}$$

Then

$$E = \begin{cases} Z \approx \text{abslin}(B), A \approx \text{lin}(A), B \approx \text{lin}(B), X \approx \text{abslin}(A), \\
P_0 \approx \text{abslin}(A), \ldots, P_n \approx \text{abslin}(A) \end{cases}$$

(2) Now, there are two linear recursive equations in $E$, namely $A$ and $B$. As in the $\text{elim\_par\_comp}$ transformation, the transformation of the previous stage is applied to these two equations to make them abstracted-linear. They become of the form $\lbrack v \mid d \gg C \rbrack$, where $v$ is a fresh variable and
$C$ is a linear equation. Now

$$E = \begin{cases} 
Z \approx [S_Z | d_Z \gg B], A \approx \text{abslin}(C), \\
B \approx \text{abslin}(C), X \approx [S_X | d_X \gg A], \\
P_0 \approx [S_{P_0} | d_{P_0} \gg A], \ldots, P_n \approx [S_{P_n} | d_{P_n} \gg A]
\end{cases}$$

and $C \approx \text{lin}(C)$.

(3) As in the $\text{elim}_{\text{par}_{\text{comp}}}$ transformation, the recursive equations in $E$ are not abstracted-linear anymore, because $A$ and $B$ are now abstracted-linear. The same transformation as in the $\text{elim}_{\text{par}_{\text{comp}}}$ transformation is applied to all recursive equations in $E$, except the equations of $A$ and $B$, to make them abstracted-linear again:

$$X \approx [S | d_X \gg A]$$

$$\approx [S | d_X \gg [v | d \gg C]]$$

$$\approx [S | [v | d_X \gg d \gg C]]$$

$$\approx [S \cup \{v\} | (d_X \sim d) \gg C]$$

Now

$$E = \begin{cases} 
Z \approx \text{abslin}(C), A \approx \text{abslin}(C), B \approx \text{abslin}(C), X \approx \text{abslin}(C), \\
P_0 \approx \text{abslin}(C), \ldots, P_n \approx \text{abslin}(C)
\end{cases}$$

and $C \approx \text{lin}(C)$.

(4) Finally, $C$ is added to $E$, and $A$ and $B$ are deleted from $E$. All recursive equations in $E$ are now abstracted-linear, except for the single equation $C$ which is linear. Now

$$E = \begin{cases} 
C \approx \text{lin}(C), Z \approx \text{abslin}(C), X \approx \text{abslin}(C), \\
P_0 \approx \text{abslin}(C), \ldots, P_n \approx \text{abslin}(C)
\end{cases}$$

The result of the $\text{elim}_{\text{encap}}$ function is the single recursion variable $Z$.

**Lemma 36 (Resulting form)** The result of the transformation $\text{elim}_{\text{encap}}$ is a single abstracted-linear recursion variable. Furthermore, after transformation, all recursive equations in $E$ are abstracted-linear except for the single equation $C$ which is linear.

**Lemma 37 (Soundness)** The transformation $\text{elim}_{\text{encap}}$ is sound, that is $\text{elim}_{\text{encap}}(H, X) \approx \partial_H(X)$ for any abstracted-linear recursion variable $X$.

**Lemma 38 (Well-definedness)** $\text{elim}_{\text{encap}}$ is well-defined.
Now that the lemmas for the \textit{elim\_par\_comp} and \textit{elim\_encap} functions have been presented, the lemmas for the function $T_1$ can be presented.

**Lemma 39 (Resulting form)** After the transformations in this stage, the specification $\langle t \mid E \rangle$ is linear.

**Lemma 40 (Soundness)** $T_1$ and the final transformation are sound.

**Lemma 41 (Well-definedness)** $T_1$ is well-defined.

5 Optimization

The linearization algorithm was proven to be correct in the previous section. However, the number of summands in the resulting linear specification is enormous. An experiment showed that the linearization of the simple Thermostat example in Section 2.4 leads to a linear specification that has 1678 summands. This number has to be reduced drastically to make our algorithm suitable for linearization of real-world models.

The experiment also showed that the problem lies in the step that combines multiple linear equations into a single linear equation (i.e. the transformation in stage 2). The problem is magnified at least quadratically when the parallel composition of two such linear equations is calculated. Therefore, the optimization efforts are focused on reducing the number and size of the summands in this linear equation.

These optimization steps could be incorporated into the main linearization algorithm. However, this would make the main algorithm more complicated and harder to understand, so the optimizations are presented separately in this section.

5.1 Creating a Single Stack Clause per Summand

All summands in the linear specification that are the result of the transformation in stage 2 have the form $\langle s \sim d \rangle \gg x$ where $s$ denotes a re-initialization clause that only contains predicates on stack variables and $d$ denotes a re-initialization clause that does \textit{not} refer to stack variables. However, after eliminating parallel composition and merging all linear equations into a single new linear equation, the summands of the new linear equation do not have the form $\langle s \sim d \rangle \gg x$ anymore. The goal of this step is to make these summands of the form $\langle s \sim d \rangle \gg x$ again.
Conjecture 42  For all terms \( x \), all re-initialization clauses \( s \) and \( s' \) which only contain predicates on stack variables as used in the linearization algorithm and all re-initialization clauses \( d \) and \( d' \) which do not refer to stack variables, where \( \equiv \) denotes equivalence of re-initialization clauses:

\[
(s \sim d) \land (s' \sim d') \equiv (s \land s') \sim (d \land d') \quad (1)
\]

\[
(s \sim d) \land d' \equiv s \sim (d \land d') \quad (2)
\]

Theorem 43  After elimination of a parallel composition, the specification contains a single linear equation. All summands in this equation can be transformed to the form \( (s \sim d) \gg x \) where \( s \) denotes a re-initialization clause that only contains predicates on stack variables and \( d \) denotes a re-initialization clause that does not refer to stack variables.

5.2 Merging Summands

Creating a single stack clause per summand did not make the size of the single linear equation smaller, but that step prepared the linear equation for this step, which merges summands. In this step, the rewrite system consisting of the following rule is applied to the right-hand side of the linear equation:

\[
s \sim d \gg x \oplus s' \sim d' \gg x \rightarrow (s \lor s') \sim d \gg x
\]

Soundness of this rule is straightforward to prove using the axioms \( d \gg x \oplus d' \gg x \approx (d \lor d') \gg x \) and \( d \sim d' \gg x \approx d \gg d' \gg x \) and termination is trivial.

Experiments show that this step drastically reduces the number of summands in the linear equation. Using this optimization, the number of summands in the resulting linear specification of the Thermostat example was reduced from 1678 to only 5 (see the next section for the linearization of this example). However, after this optimization, the re-initialization clauses on stack variables are very large. We strongly feel that these clauses can be optimized further, by trying to simplify their contents. This optimization is not a core part of the algorithm though, so this is considered to be future work.

5.3 Eliminating Superfluous Clauses

A final simple step is to eliminate all superfluous \([true] \), \([false] \) and \( c_{jmp} \) re-initialization clauses. This is achieved by applying the rewrite system consisting of the following rules to the right-hand side of the linear equation:
\[
x \oplus \delta \quad \rightarrow \quad x \\
\delta \oplus x \quad \rightarrow \quad x \\
[\text{false}] \gg x \quad \rightarrow \quad \delta \\
[\text{true}] \gg x \quad \rightarrow \quad x \\
[\text{false}] \land d \quad \rightarrow \quad [\text{false}] \\
[\text{true}] \land d \quad \rightarrow \quad d \\
d \land [\text{false}] \quad \rightarrow \quad [\text{false}] \\
d \land [\text{true}] \quad \rightarrow \quad d \\
(c_{\text{jmp}} \land c'_{\text{jmp}}) \gg (c \land c') \quad \rightarrow \quad c \land c'
\]

It is straightforward to verify that these rewrite rules are sound, except for the rule for \(c_{\text{jmp}}\) clauses. Soundness of this rule can be proven by calculation on re-initialization clauses and the axiom \(c_{\text{jmp}} \gg c \approx c\). Furthermore, the rewrite system is terminating, because the right-hand side of every rule is strictly smaller than its left-hand side.

### 6 Example

In this section, the example in Section 2.4 is linearized to illustrate the linearization algorithm. The model of this example is the following:

\[
\begin{align*}
\text{HeaterOn} & \approx (x | \dot{x} = -x + 4) \gg r\text{Off} \odot \text{HeaterOff} \\
\text{HeaterOff} & \approx (x | \dot{x} = -x) \gg r\text{On} \odot \text{HeaterOn} \\
\text{Thermostat} & \approx (x_{\text{min}} \leq x \leq x_{\text{max}}) \gg \\
& \quad \left( [x = x_{\text{min}}] \gg s\text{On} \odot \text{Thermostat} \right) \oplus \\
& \quad \left( [x = x_{\text{max}}] \gg s\text{Off} \odot \text{Thermostat} \right) \\
\text{Controller} & \approx \partial_H (\text{HeaterOn} \parallel \text{Thermostat})
\end{align*}
\]

\[
\gamma(s\text{On}, r\text{On}) = \text{On} \\
\gamma(s\text{Off}, r\text{Off}) = \text{Off} \\
H = \{s\text{Off}, s\text{On}, r\text{Off}, r\text{On}\}
\]
The HyPA\textsubscript{lin} specification of this model is therefore

\[ \langle \partial H (\text{HeaterOn} \parallel \text{Thermostat}) \mid E \rangle \]

where \( E \) consists of the previously given equations.

### 6.1 Stage 1: Transforming Equations into Semi-linear Form

The right-hand sides of all equations in the specification are transformed to semi-linear form. This results in the following specification:

\[ \langle \partial H (\text{HeaterOn} \parallel \text{Thermostat}) \mid E_1 \rangle \]

where \( E_1 \) consists of the following equations:

- **Thermostat**
  \[ \approx [\text{true}] \gg (x_{\text{min}} \leq x \leq x_{\text{max}}) \triangleright X2 \]
  \[ \triangleright [x = x_{\text{min}}] \gg \text{sOn} \odot \text{Thermostat} \]
  \[ \triangleright [x = x_{\text{max}}] \gg \text{sOff} \odot \text{Thermostat} \]

- **X2**
  \[ \approx [x = x_{\text{min}}] \gg \text{sOn} \odot \text{Thermostat} \]
  \[ \triangleright [x = x_{\text{max}}] \gg \text{sOff} \odot \text{Thermostat} \]

- **HeaterOff**
  \[ \approx [\text{true}] \gg (x \mid \dot{x} = -x) \triangleright X4 \]
  \[ \triangleright [\text{true}] \gg \text{rOn} \odot \text{HeaterOn} \]

- **X4**
  \[ \approx [\text{true}] \gg \text{rOn} \odot \text{HeaterOn} \]

- **HeaterOn**
  \[ \approx [\text{true}] \gg (x \mid \dot{x} = -x + 4) \triangleright X6 \]
  \[ \triangleright [\text{true}] \gg \text{rOff} \odot \text{HeaterOff} \]

- **X6**
  \[ \approx [\text{true}] \gg \text{rOff} \odot \text{HeaterOff} \]

### 6.2 Stage 2: From Semi-linear to Abstracted-linear

Now, the equations of the previous stage are combined into one linear equation \( A_0 \) and all equations in the specification are transformed into abstracted-linear form. Note that \( A_0 \) is optimized using the optimizations of the previous section. This stage results in the following specification:

\[ \langle \partial H (\text{HeaterOn} \parallel \text{Thermostat}) \mid E_2 \rangle \]
where $E_2$ consists of the following equations

\[
\begin{align*}
\text{HeaterOff} & \approx \left[ s_0 \mid s_0^+ = \text{push}(\text{HeaterOff}, \emptyset) \right] \Rightarrow A_0 \\
\text{X4} & \approx \left[ s_0 \mid s_0^+ = \text{push}(\text{X4}, \emptyset) \right] \Rightarrow A_0 \\
\text{HeaterOn} & \approx \left[ s_0 \mid s_0^+ = \text{push}(\text{HeaterOn}, \emptyset) \right] \Rightarrow A_0 \\
\text{X6} & \approx \left[ s_0 \mid s_0^+ = \text{push}(\text{X6}, \emptyset) \right] \Rightarrow A_0 \\
\text{Thermostat} & \approx \left[ s_0 \mid s_0^+ = \text{push}(\text{Thermostat}, \emptyset) \right] \Rightarrow A_0 \\
\text{X2} & \approx \left[ s_0 \mid s_0^+ = \text{push}(\text{X2}, \emptyset) \right] \Rightarrow A_0
\end{align*}
\]

and

\[
A_0 \approx \left( \begin{array}{c}
s_0 \neq \emptyset \land \text{get}(s_0^-) = \text{HeaterOff} \\
\land s_0^+ = \text{push}(\text{HeaterOn}, \text{pop}(s_0^-)) \\
\end{array} \right) \lor
\]

\[
\Rightarrow r\text{On} \odot A_0
\]

\[
\oplus \left( \begin{array}{c}
s_0 \neq \emptyset \land \text{get}(s_0^-) = \text{HeaterOn} \\
\land s_0^+ = \text{push}(\text{HeaterOn}, \text{pop}(s_0^-)) \\
\end{array} \right) \lor
\]

\[
\Rightarrow r\text{Off} \odot A_0
\]

\[
\oplus \left( \begin{array}{c}
s_0 \neq \emptyset \land \text{get}(s_0^-) = \text{Thermostat} \\
\land s_0^+ = \text{push}(\text{Thermostat}, \text{pop}(s_0^-)) \\
\end{array} \right) \lor
\]

\[
\Rightarrow s\text{On} \odot A_0
\]

\[
\oplus \left( \begin{array}{c}
s_0 \neq \emptyset \land \text{get}(s_0^-) = \text{Thermostat} \\
\land s_0^+ = \text{push}(\text{Thermostat}, \text{pop}(s_0^-)) \\
\end{array} \right) \lor
\]

\[
\Rightarrow s\text{Off} \odot A_0
\]

\[
\oplus \left[ s_0 \neq \emptyset \land \text{get}(s_0^-) = \text{HeaterOff} \land s_0^+ = \text{push}(\text{X4}, \text{pop}(s_0^-)) \right] \Rightarrow (\{x \mid \dot{x} = -x \land (s_0 \mid s_0^- = 0)\} \Rightarrow A_0
\]

\[
\oplus \left[ s_0 \neq \emptyset \land \text{get}(s_0^-) = \text{HeaterOn} \land s_0^+ = \text{push}(\text{X6}, \text{pop}(s_0^-)) \right]
\]

37
\[ ((x|\dot{x} = -x + 4) \land (s_0|\dot{s}_0 = 0)) \Rightarrow A_0 \]
\[ \oplus [s_0|s_0 \neq 0 \land \text{get}(s_0^-) = \text{Thermostat} \land s_0^+ = \text{push}(X2, \text{pop}(s_0^-))] \]
\[ ((x_{\text{min}} \leq x \leq x_{\text{max}}) \land (s_0|\dot{s}_0 = 0)) \Rightarrow A_0 \]

6.3 Stage 3: Transforming the Initial Term

First, the parallel composition of HeaterOn and Thermostat is computed. Two new stack variables are introduced, namely one in the step where the stack variable of the Thermostat is renamed and one in the step where all equations are combined into one linear equation again. Note that a clause that contains only conditions on stack variables is denoted as a function \([sc_i(s_0, \ldots, s_n)]\), which represents a boolean predicate on the stack variables \(s_0, \ldots, s_n\). The actual contents of these clauses are omitted, because these clauses are quite large at this point (one clause would fill a whole page). After this step, the optimized specification is the following:

\[ \langle \partial_H(Z_0) \mid \{Z_0, A_2\} \rangle \]

where

\[ Z_0 \approx \left[ \begin{array}{c} s_0, s_1, s_2 \\ \hline \end{array} \right] \left[ \begin{array}{c} [s_0|s_0^+ = \text{push}(\text{HeaterOn}, \emptyset)] \sim \\ [s_1|s_1^+ = \text{push}(\text{Thermostat}, \emptyset)] \sim \\ [s_2|s_2^+ = \text{push}(Z_0, \emptyset)] \end{array} \right] \Rightarrow A_2 \]

\[ A_2 \approx [sc_0(s_0, s_1, s_2)] \sim [x = x_{\text{max}}] \Rightarrow \text{Off} \odot A_2 \]
\[ \oplus [sc_1(s_0, s_1, s_2)] \sim [x = x_{\text{min}}] \Rightarrow \text{On} \odot A_2 \]
\[ \oplus [sc_2(s_0, s_1, s_2)] \sim [x = x_{\text{min}}] \Rightarrow s\text{On} \odot A_2 \]
\[ \oplus [sc_3(s_0, s_1, s_2)] \sim [x = x_{\text{max}}] \Rightarrow s\text{Off} \odot A_2 \]
\[ \oplus [sc_4(s_0, s_1, s_2)] \Rightarrow r\text{Off} \odot A_2 \]
\[ \oplus [sc_5(s_0, s_1, s_2)] \Rightarrow r\text{On} \odot A_2 \]

\[ \oplus [sc_6(s_0, s_1, s_2)] \Rightarrow \begin{cases} (x|\dot{x} = -x + 4) \land (x_{\text{min}} \leq x \leq x_{\text{max}}) \\ \land (s_0|\dot{s}_0 = 0) \land (s_1|\dot{s}_1 = 0) \\ \land (s_2|\dot{s}_2 = 0) \end{cases} \Rightarrow A_2 \]

\[ \oplus [sc_7(s_0, s_1, s_2)] \Rightarrow \begin{cases} (x|\dot{x} = -x) \land (x_{\text{min}} \leq x \leq x_{\text{max}}) \\ \land (s_0|\dot{s}_0 = 0) \land (s_1|\dot{s}_1 = 0) \\ \land (s_2|\dot{s}_2 = 0) \end{cases} \Rightarrow A_2 \]
Then, the encapsulation of $Z_0$ is calculated (that is the encapsulation of HeaterOn $|$ Thermostat). After this step, the specification is $\langle X8 \mid \{X8, A3\}\rangle$ where $\text{encap}_Z$ is the fresh recursion variable that results from the elimination of encapsulation step.

$$\oplus [sc_8(s_0, s_1, s_2)] \gg \begin{cases} (x| \dot{x} = -x) \land (x| \dot{x} = -x + 4) \\ \land (s_0| \dot{s}_0 = 0) \land (s_1| \dot{s}_1 = 0) \\ \land (s_2| \dot{s}_2 = 0) \end{cases} \triangleright A_2$$

$$\oplus [sc_9(s_0, s_1, s_2)] \gg \begin{cases} (x| \dot{x} = -x) \land (s_0| \dot{s}_0 = 0) \\ \land (s_1| \dot{s}_1 = 0) \land (s_2| \dot{s}_2 = 0) \end{cases} \triangleright A_2$$

$$X8 \approx \begin{bmatrix} s_0, s_1, s_2, s_3 \end{bmatrix} \begin{bmatrix} [s_0| s_0^+ = \text{push(HeaterOn, 0)}] \sim \\ [s_1| s_1^+ = \text{push(Thermostat, 0)}] \sim \\ [s_2| s_2^+ = \text{push(Z0, 0)}] \sim \\ [s_3| s_3^+ = \text{push(\text{encap}_Z, 0)}] \end{bmatrix} \triangleright A_3$$

$$A_3 \approx [sc_0(s_0, s_1, s_2, s_3)] \sim [x = x_{\text{min}}] \gg \text{On} \odot A_3$$
$$\oplus [sc_1(s_0, s_1, s_2, s_3)] \sim [x = x_{\text{max}}] \gg \text{Off} \odot A_3$$

$$\oplus [sc_2(s_0, s_1, s_2, s_3)] \gg \begin{cases} (x| \dot{x} = -x) \land (x| \dot{x} = -x + 4) \\ \land (s_0| \dot{s}_0 = 0) \land (s_1| \dot{s}_1 = 0) \\ \land (s_2| \dot{s}_2 = 0) \land (s_3| \dot{s}_3 = 0) \end{cases} \triangleright A_3$$

$$\oplus [sc_3(s_0, s_1, s_2, s_3)] \gg \begin{cases} (x| \dot{x} = -x) \land (x_{\text{min}} \leq x \leq x_{\text{max}}) \\ \land (s_0| \dot{s}_0 = 0) \land (s_1| \dot{s}_1 = 0) \\ \land (s_2| \dot{s}_2 = 0) \land (s_3| \dot{s}_3 = 0) \end{cases} \triangleright A_3$$

$$\oplus [sc_4(s_0, s_1, s_2, s_3)] \gg \begin{cases} (x| \dot{x} = -x + 4) \\ \land (x_{\text{min}} \leq x \leq x_{\text{max}}) \\ \land (s_0| \dot{s}_0 = 0) \land (s_1| \dot{s}_1 = 0) \\ \land (s_2| \dot{s}_2 = 0) \land (s_3| \dot{s}_3 = 0) \end{cases} \triangleright A_3$$

Finally, the abstraction in $X8$ is pulled into the initial term, which gives the linear specification:

$$\langle \llbracket s_0, s_1, s_2, s_3 \mid i \gg A_3 \rrbracket \mid \{A_3\} \rangle$$

39
where

\[
i = \begin{align*}
& [s_0 \mid s_0^+ = \text{push}(\text{HeaterOn}, \emptyset)] \sim [s_1 \mid s_1^+ = \text{push}(\text{Thermostat}, \emptyset)] \\
& [s_2 \mid s_2^+ = \text{push}(Z_0, \emptyset)] \sim [s_3 \mid s_3^+ = \text{push}(\text{encap}Z_0, \emptyset)]
\end{align*}
\]

7 Conclusions and Future Work

We presented a linearization algorithm for the hybrid process algebra HyPA, proved its correctness and presented several optimizations. Our linearization algorithm transforms a HyPA\textsubscript{lin} specification into an equivalent linear recursive specification. Furthermore, we introduced an abstraction operator for HyPA and gave several useful axioms.

The main advantage of linear recursive specifications is that it becomes fairly straightforward to generate the state space from them. Furthermore, a weaker notion of equivalence can be used on linear specifications, which enables the use of certain analysis techniques that cannot be used on normal HyPA specifications. Finally, linear recursive specifications are a convenient form for exchange and manipulation by tools.

We have implemented our algorithm in an experimental tool. This tool showed that application of the algorithm to real-world specifications is still problematic, because the re-initialization clauses on stack variables are very large in the resulting linear specification. Therefore, the most pressing issue currently is optimization of the size of these re-initialization clauses. This optimization would be a very interesting and useful subject of future research, because we feel that this is the last step to enable the linearization of real-world specifications.

Moreover, there are still a number of restrictions on the input specifications because of fundamental difficulties. First, the parallel composition is restricted in such a way that there is no recursion over the parallel composition, as in \( X \approx X \parallel Y \) for instance. Second, the abstraction operator is not allowed in the HyPA\textsubscript{lin} form, because it is not possible to eliminate abstraction of open terms from recursive equations. Third, the empty process (\( \epsilon \)) cannot be used, because it leads to some problems in the transformations. Finally, only single flow clauses can be disrupted. Relaxing these restrictions, especially on the use of recursion, is an interesting topic for future work as well.
References


A Proofs

A.1 Linearization Stage 1: Transforming Equations into Semi-linear Form

A.1.1 Simple Rewriting

PROOF. [Lemma 18] We prove that normal forms of this rewrite system, when applied to terms of the form $\text{HyPA}_{lin}$, are of the form $r$. The atoms of the $\text{HyPA}_{lin}$ form $(a, X, \delta, c)$ are in $r$. Therefore, every $\text{HyPA}_{lin}$ term $p \notin r$ has a smallest sub term $s \notin r$ of the form $R$, where $r$ denotes the form $r$:

$$R ::= r \oplus r \mid r \odot r \mid d \gg r \mid c \rhd r \mid c \triangleright r$$

We give at least one applicable rewrite rule for every of these possible sub terms, unless the specific sub term is in normal form already. In that case, we
do not need to give a rule, because we have a contradiction with the initial assumption that $s \not\in r$.

- $s$ of the form $r \oplus r$: implies $s \in r$.
- $s$ of the form $r \odot r$:
  - $s$ of the form $a \odot r, X \odot r, \delta \odot r, c \odot r, (d \gg r_2) \odot r, (r_2 \odot r) \odot r$
    or $(c \triangleright r) \odot r$: implies $s \in r$.
  - $s$ of the form $(r \oplus r) \odot r$: rewrites using (R2).
- $s$ of the form $d \gg r$:
  - $s$ of the form $d \gg a, d \gg X, d \gg \delta, d \gg c, d \gg (d \gg r_2), d \gg (r_2 \odot r)$
    or $d \gg (c \triangleright r)$: implies $s \in r$.
  - $s$ of the form $d \gg (r \oplus r)$: rewrites using (R3).
- $s$ of the form $c \triangleright r$: rewrites using (R1).
- $s$ of the form $c \triangleright r$: implies $s \in r$.

All existing equations in $E$ are rewritten into the form $r$ and no new equations are added to $E$. Therefore, after applying this rewrite system to the right-hand sides of all equations in $E$, the right-hand sides of all equations in $E$ are of the form $r$.

**PROOF. [Lemma 19]** All rewrite rules are directed versions of HyPA axioms. Therefore, this rewrite step is sound.

**PROOF. [Lemma 20]** In [7], a rewrite system is presented that is proven to rewrite closed HyPA terms into basic terms. That rewrite system was proved to be terminating. The rewrite system used in this step is a subset of that rewrite system, so the rewrite system in this step is terminating as well.

### A.1.2 Adding New Recursive Equations

**PROOF. [Lemma 21]** Before applying $S_1$, the right-hand sides of all equations in $E$ have the form $r$. We have two proof obligations.

First, $S_1(p)$ is of the form $s$, for any term $p$ of the form $r$: applying the function $S_1$ to the form $r$ gives the following form. The sub form $s_1^1$ results from $S_1$ applied to $r$, $s_2^1$ results from $S_1$ applied to $r_2$, $s_1^2$ results from $S_2$ applied to $r$, $s_2^2$ results from $S_2$ applied to $r_2$. 


and \( s_2^2 \) results from \( S_2 \) applied to \( r_2 \).

\[
\begin{align*}
  s_1^1 & := a \mid X \mid \delta \mid c \triangleright X \mid d \gg s_2^1 \mid s_2^1 \circ s_2^2 \mid s_1^1 \oplus s_1^1 \\
  s_1^2 & := a \mid X \mid \delta \mid d \gg s_1^2 \\
  s_2^1 & := a \mid X \mid \delta \mid d \gg s_2^1 \\
  s_2^2 & := a \mid X \mid \delta \mid d \gg s_2^2
\end{align*}
\]

We see that \( s_1^1, s_1^2 \) and \( s_2^2 \) are identical. Therefore, we can delete two of these rules and replace all references to the two deleted rules by the name of the third remaining rule. Renaming the two rules that are left gives the form \( s \).

Second, all newly introduced equations are of the form \( s \): equations with right-hand sides of the following form might be introduced:

- \( \delta \), which is in \( s \).
- \( c \triangleright Y \), which is in \( s \).
- \( S_1(p) \) for \( p \) of the form \( r \): \( S_1(p) \) is of the form \( s \).
- \( S_1(p) \oplus S_1(q) \) for \( p \) and \( q \) of the form \( r \): \( S_1(p) \) and \( S_1(q) \) are both of the form \( s \) and \( s \oplus s \) is in the form \( s \).
- \( S_2(p) \circ S_2(q) \) for \( p \) of the form \( r_2 \) and \( q \) of the form \( r \): First, we prove by induction that \( S_2(x) \) is of the form \( s_2 \) for any \( x \) of the form \( r \):
  - \( a \): \( S_2(a) = a \), which is in \( s_2 \).
  - \( \delta \): \( S_2(\delta) = \delta \), which is in \( s_2 \).
  - \( X \), \( c \), \( y \odot z \), \( c \triangleright z \), \( z \oplus z \), where \( y \) is of the form \( r_2 \) and \( z \) is of the form \( r \): applying \( S_2 \) to these terms gives a single recursion variable \( X \), which is in \( s_2 \).
  - \( d \gg y \), where \( y \) is of the form \( r_2 \): \( S_2(d \gg y) = d \gg S_2(y) \). Since \( r_2 \) is a subset of \( r \), applying the induction hypothesis gives \( d \gg S_2(y) = d \gg z \) where \( z \) is of the form \( s_2 \). \( d \gg y \) is of the form \( s_2 \) as well, so \( S_2(d \gg y) \) is of the form \( s_2 \).

Now, we see that \( S_2(p) \) and \( S_2(q) \) are both of the form \( s_2 \). \( s_2 \circ s_2 \) is in \( s \), so \( S_2(p) \circ S_2(q) \) is of the form \( s \).

We conclude that after applying \( S_1 \) to the right-hand sides of all equations in \( E \), the right-hand sides of all equations in \( E \) are of the form \( s \).

**PROOF.** [Lemma 22] First, we prove by induction on the form \( r \) that \( S_1(p) \approx S_2(p) \) for any \( p \) of the form \( r \).

Base cases:

---

4 Although \( d \gg s_1^1 \) and \( d \gg s_2^2 \) appear to be different, they describe the same form in these three BNF rules.
• a, X, δ: trivial.
• c: $S_1(c) = X \approx c \triangleright Y \approx c \triangleright \delta \approx c$ and $S_2(c) = X \approx c \triangleright Y \approx c \triangleright \delta \approx c$, so $S_1(c) \approx S_2(c)$.
• p ⊗ q, where p is of the form $r_2$ and q is of the form r: from the definition of $S_1$ follows that $S_1(p \otimes q) = S_2(p \otimes q)$.
• c ⊳ p, where p is of the form r: from the definition of $S_1$ follows that $S_1(c \triangleright p) = S_2(c \triangleright p)$.
• p ⊕ q, where p and q are of the form r: $S_2(p \oplus q) = X \approx S_1(p) \oplus S_1(q) = S_1(p \oplus q)$.

Induction case:

• d ≫ p, where p is of the form $r_2$: $r_2$ is a subset of r, so using the induction hypothesis $S_1(p) \approx S_2(p)$ we see that $S_1(d \triangleright p) = d \triangleright S_1(p) \approx d \triangleright S_2(p) = S_2(d \triangleright p)$.

Now, we prove by induction that $S_2(p) \approx p$ for any p of the form r.

Base cases:

• a, X, δ: trivial.
• c: $S_2(c) = X \approx c \triangleright Y \approx c \triangleright \delta \approx c$.

Induction cases:

• d ≫ p, where p is of the form $r_2$: using the induction hypothesis $S_2(p) \approx p$, we see that $S_2(d \triangleright p) = d \triangleright S_2(p) \approx d \triangleright p$.
• p ⊗ q, where p is of the form $r_2$ and q is of the form r: using the induction hypothesis twice gives us $S_2(p \otimes q) = X \approx S_2(p) \otimes S_2(q) \approx p \otimes q$.
• c ⊳ p, where p is of the form r: using the fact that $S_1(p) \approx S_2(p)$ and the induction hypothesis we see that $S_2(c \triangleright p) = X \approx c \triangleright Y \approx c \triangleright S_1(p) \approx S_2(p) \triangleright c \triangleright p$.
• p ⊕ q, where p and q are of the form r: using the fact that $S_1(p) \approx S_2(p)$ and the induction hypothesis we see that $S_2(p \oplus q) = X \approx S_1(p) \oplus S_1(q) \approx S_2(p) \oplus S_2(q) \approx p \oplus q$.

From $S_1(p) \approx S_2(p)$ and $S_2(p) \approx p$ we conclude that $S_1(p) \approx p$. Therefore, $S_1$ is sound.

**PROOF.** [Lemma 23] When either of the functions $S_1$ or $S_2$ is used in the right-hand side, it is usually applied to a (strict) sub term of the argument of the left-hand side. One exception is the case $p \otimes q$ in $S_1$, but here we see that $S_1(p \otimes q) = S_2(p \otimes q) = X$ and $E := E \cup \{X \approx S_2(p) \otimes S_2(q)\}$, so immediately after the next step there is recursion on strictly smaller sub terms of $p \otimes q$. The only other exception is $c \triangleright p$ in $S_1$, which is analogous to the previous exception. Therefore, $S_1$ is well-defined.
A.1.3 Guarding

Lemma 44 \( \text{rewr}(p) \) is of the form \( g \), for any \( p \) of the form \( g \).

PROOF. We prove this by showing that each rewrite rule has this property. If a rule is not applicable to any term of the form \( g \), then it follows trivially that this term is still of the form \( g \).

- (R10), (R11), (R12), (R14), (R16), (R17), (R19) and (R20): cannot be applied to any term of the form \( g \).
- (R15): this rule is only applicable to sub terms of the form \( g^2 \). Application of this rule gives \( \delta \), which is in \( g^2 \) as well. We replaced a \( g^2 \) sub term by another \( g^2 \) term, so the resulting term is still in \( g \).
- (R18): this rule is only applicable to sub terms of the form \( g^2 \), where \( x \), \( y \) and \( z \) are of the form \( g^2 \). Application of this rule gives \( x \odot (y \odot z) \), which is in \( g^2 \) as well. We replaced a \( g^2 \) sub term by another \( g^2 \) term, so the resulting term is still in \( g \).

Lemma 45 \( \text{rewr}(d \gg p) \) has the form \( g \), for any \( p \) of the form \( g \).

PROOF. We prove this by induction on the structure of the form \( g \). We show for any term of the form \( d \gg p \), where \( p \) is of the form \( g \), that it is already in \( g \) or can be rewritten into a term that is in \( g \). Note that according to Lemma 44, such a term will remain in \( g \) after applying other rewrite rules. Also note that rewriting on \( p \) itself has no influence, because \( p \) will remain in \( g \).

- \( d \gg (q \oplus r) \), for \( q \), \( r \) in \( g \): only rewrites with (R11) to \( d \gg q \oplus d \gg r \).
  Applying the induction hypothesis twice gives that \( \text{rewr}(d \gg q) \) is in \( g \) and \( \text{rewr}(d \gg r) \) is in \( g \). \( g \oplus g \) is in \( g \), so \( \text{rewr}(d \gg (q \oplus r)) \) is in \( g \).
- \( d \gg \delta \): only rewrites with (R14) to \( \delta \), which is in \( g \).
- \( d \gg a \): is in \( g \).
- \( d \gg (a \odot q) \), for \( q \) in \( g^2 \): is in \( g \).
- \( d \gg (c \triangleright q) \), for \( q \) in \( g^2 \): is in \( g \).
- \( d \gg (d' \gg a) \): only rewrites with (R17) to \( (d \sim d') \gg a \), which is in \( g \).
- \( d \gg (d' \gg (a \odot q)) \), for \( q \) in \( g^2 \): only rewrites with (R17) to \( (d \sim d') \gg (a \odot q) \), which is in \( g \).
- \( d \gg ((d' \gg a) \odot q) \), for \( q \) in \( g^2 \): only rewrites with (R20) to \( (d \sim d') \gg (a \odot q) \), which is in \( g \).
- \( d \gg (d' \gg (c \triangleright q)) \), for \( q \) in \( g^2 \): only rewrites with (R17) to \( (d \sim d') \gg (c \triangleright q) \), which is in \( g \).

Lemma 46 \( \text{rewr}(p \odot q) \) has the form \( g \), for \( p \) of the form \( g \) and \( q \) of the form \( s^2 \).
PROOF. We prove this by induction on the structure of the form $g$. We show for any term of the form $p \odot q$, where $p$ is of the form $g$ and $q$ is of the form $s_2$, that it is already in $g$ or can be rewritten into a term that is in $g$. Note that according to Lemma 44, such a term will remain in $g$ after applying other rewrite rules. Also note that rewriting on $p$ itself has no influence, because $p$ will remain in $g$ and it can be verified that the same holds for $q$.

- $(r \ominus s) \odot q$, for $r, s$ in $g$: rewrites with (R10) to $r \odot q \ominus s \odot q$. Applying the induction hypothesis twice gives that $\text{rewr}(r \odot q)$ is in $g$ and $\text{rewr}(s \odot q)$ is in $g$. $g \ominus g$ is in $g$, so $r \odot q \ominus s \odot q$ is in $g$.
- $\delta \odot q$: rewrites with (R15) to $\delta$, which is in $g$.
- $a \odot q$: $s_2$ is a subset of $g_2$ and $a \odot g_2$ is in $g$, so $a \odot q$ is in $g$.
- $(a \odot r) \odot q$, for $r$ in $g_2$: rewrites with (R18) to $a \odot (r \odot q)$. $s_2$ is a subset of $g_2$ and $a \odot (g_2 \odot g_2)$ is in $g$, so $a \odot (r \odot q)$ is in $g$.
- $(c \triangleright r) \odot q$, for $r$ in $g_2$: rewrites with (R12) to $c \triangleright (r \odot q)$. $s_2$ is a subset of $g_2$ and $c \triangleright (g_2 \odot g_2)$ is in $g$, so $c \triangleright (r \odot q)$ is in $g$.
- $(d \triangleright a) \odot q$: $s_2$ is a subset of $g_2$ and $(d \triangleright a) \odot g_2$ is in $g$, so $(d \triangleright a) \odot q$ is in $g$.
- $(d \triangleright (a \odot r)) \odot q$, for $r$ in $g_2$: rewrites with (R19) to $d \triangleright ((a \odot (r \odot q))$. $s_2$ is a subset of $g_2$ and $d \triangleright ((a \odot (g_2 \odot g_2))$ is in $g$, so $d \triangleright ((a \odot (r \odot q))$ is in $g$.
- $((d \triangleright a) \odot r) \odot q$, for $r$ in $g_2$: rewrites with (R18) to $(d \triangleright a) \odot (r \odot q)$. $s_2$ is a subset of $g_2$ and $(d \triangleright a) \odot (g_2 \odot g_2)$ is in $g$, so $(d \triangleright a) \odot (r \odot q)$ is in $g$.
- $(d \triangleright (c \triangleright r)) \odot q$, for $r$ in $g_2$: rewrites with (R16) to $d \triangleright (c \triangleright (r \odot q))$. $s_2$ is a subset of $g_2$ and $d \triangleright (c \triangleright (g_2 \odot g_2))$ is in $g$, so $d \triangleright (c \triangleright (r \odot q))$ is in $g$.

PROOF. [Lemma 24] We prove that guard($p$) has the form $g$, for any $p$ of the form $s$. Lemma 26 states that the guard function is well-defined, so we can apply induction on the form $s$:

- $a, \delta, c \triangleright X$: trivial.
- $X$: guard($X$) = guard(rhs($X$)). The form of rhs($X$) is $s$, so our induction hypothesis applies and we conclude that guard($X$) has the form $g$.
- $p \ominus q$, for any $p, q$ of the form $s$: guard($p \ominus q$) = guard($p$) $\ominus$ guard($q$). The induction hypothesis applies to both operands. Therefore, guard($p$) and guard($q$) are both of the form $g$ and $g \ominus g$ is in $g$.
- $d \triangleright p$, for any $p$ of the form $s_2$: guard($d \triangleright p$) = $\text{rewr}(d \triangleright guard(p))$. Since $s_2$ is a subset of $s$, $p$ is also of the form $s$. Hence, the induction hypothesis applies, so guard($p$) is of the form $g$. By Lemma 45, $\text{rewr}(d \triangleright guard(p))$ has the form $g$.
- $p \odot q$, for any $p, q$ of the form $s_2$: guard($p \odot q$) = $\text{rewr}(guard(p) \odot q)$. Since $s_2$ is a subset of $s$, $p$ is also of the form $s$. Hence, the induction hypothesis
applies, so $guard(p)$ is of the form $g$. By Lemma 46, $rewr(guard(p) \odot q)$ has the form $g$.

All existing equations in $E$ are rewritten into the form $g$ and no new equations are added to $E$. Therefore, after applying $guard$ to the right-hand sides of all equations in $E$, the right-hand sides of all equations in $E$ are of the form $g$.

**PROOF.** [Lemma 25] Most rewrite rules are directed versions of HyPA axioms, except for rules (R12), (R15), (R19) and (R20). These rules are derivable from the axioms as follows:

(R12) :  $$(c \triangleright x) \odot y \approx ([true] \triangleright (c \triangleright x)) \odot y$$

(R15) :  $$\delta \odot x \approx ([false] \triangleright a) \odot x$$

(R19) :  $$(d \triangleright (a \odot x)) \odot y \approx ((d \triangleright a) \odot x) \odot y$$

(R20) :  $$d \triangleright ((d' \triangleright a) \odot x) \approx d' \triangleright ((a \odot x))$$

Therefore, the rewrite system $rewr$ is sound. It is trivial to see that the $guard$ function is sound, since it only substitutes recursion variables by their right-hand sides and the rewrite system is sound.

**Definition 47 (PNUDG)**  The Process Name Unguarded Dependency Graph (PNUDG) \(^5\) of a specification $\langle t \mid E \rangle$ is constructed as follows. Every recursion variable in the specification is a node and there is a directed edge from a node $X$ to a node $Y$ if $Y$ occurs unguarded in the right-hand side of the equation for $X$ in $E$.

**Lemma 48** If the specification $\langle t \mid E \rangle$ is guarded, then its PNUDG has no cycles.

**PROOF.** Recall that the input of the linearization algorithm is a guarded recursive specification. A recursive specification $\langle t \mid E \rangle$ is guarded if the right-

\(^5\) This definition is adapted from [14], but our notion of guardedness is quite different.
hand sides of all equations in $E$ are guarded, or can be transformed into guarded terms by replacing variables by the right-hand side of their equation.

Suppose $\langle t \mid E \rangle$ is guarded. Then, we have the following two cases:

- the right-hand sides of all equations in $E$ are guarded: the PNUDG contains no edges, so the PNUDG has no cycles.
- the right-hand sides of all equations in $E$ can be transformed into guarded terms by replacing variables by the right-hand side of their equation: to make a certain equation $X$ in $E$ guarded, we have to repeatedly substitute unguarded occurrences of recursion variables by the right-hand side of their defining equations in the right-hand side of the equation for $X$. This means that, to make $X$ guarded, we at least have to follow every path in the PNUDG starting from $X$. The fact that it is possible to make $X$ guarded, implies that it takes only finitely many substitutions to do that. Therefore all paths in the PNUDG starting in $X$ are finite. This holds for every equation in $E$, so all paths in the PNUDG are finite. Therefore, the PNUDG has no cycles.

**Proof.** [Lemma 26] In [7], a rewrite system is presented that is proven to rewrite closed HyPA terms into basic terms. That rewrite system was also proved to be terminating. The rewrite system $\text{rewr}$ is a subset of that rewrite system, so $\text{rewr}$ is terminating as well.

The only case that makes the argument of $\text{guard}$ larger is the case of $X$. Note that $\text{guard}(X)$ is only applied to unguarded occurrences of $X$, because guarded occurrences of recursion variables only occur in the right argument of the sequential composition ($\odot$) and the $\text{guard}$ function is not applied to these arguments ($\text{guard}(p \odot q) = \text{rewr}(\text{guard}(p) \odot q)$). Now, let $n$ be the number of equations in $E$. Due to the fact that the PNUDG is acyclic, this clause cannot be applied more than $n$ times (otherwise there would have to be a cycle in the PNUDG).

**4.1.4 Post Processing**

**Lemma 49** $P_2(p)$ is of the form $l_2$, for any $p$ of the form $g_2$.

**Proof.** We prove this lemma by induction on the structure of the form $g_2$, with the lemma as the induction hypothesis. For the base cases $\delta, a, X, d \gg a$ and $d \gg X$ this is trivial, since all of them give a single recursion variable $X$, which is in the form $l_2$. 

49
Inductive case $p \odot q$, where $p$ and $q$ are of the form $g_2$: applying the induction hypothesis twice we see that $P_2(p)$ and $P_2(q)$ are both of the form $l_2$. Since $l_2 \odot l_2$ is in $l_2$, we conclude that $P_2(p \odot q)$ is in $l_2$.

**Lemma 50** After applying $P_1$ to the right-hand sides of all equations in $E$, these right-hand sides are either of the form $l$ or of the form $d \gg X$ where $\text{rhs}(X)$ is of the form $l$.

**PROOF.** Before applying $P_1$, the right-hand sides of all equations in $E$ have the form $g$. We have two proof obligations:

- $P_1(p)$ is of the form $l$ for any $p$ of the form $g$: we prove this by induction on the structure of the form $g$. For the base cases $\delta$, $a$ and $d \gg a$ this is trivial. Inductive cases:
  - $p \oplus q$, where $p$ and $q$ are of the form $g$: applying the induction hypothesis twice, we see that both $P_1(p)$ and $P_1(q)$ are of the form $l$. Since $l \odot l$ is in $l$, $P_1(p \oplus q)$ is also in $l$.
  - $a \odot p$, where $p$ is of the form $g_2$: applying Lemma 49 gives us that $P_2(p)$ is of the form $l_2$. Since $[true] \gg (a \odot l_2)$ is in $l$, $P_1(a \odot p)$ is also in $l$.
  - $c \triangleright p$, where $p$ is of the form $g_2$: applying Lemma 49 gives us that $P_2(p)$ is of the form $l_2$. Since $[true] \gg (c \triangleright l_2)$ is in $l$, $P_1(c \triangleright p)$ is also in $l$.
  - $d \gg (a \odot p)$, where $p$ is of the form $g_2$: analogous to $P_1(a \odot p)$.
  - $(d \gg a) \odot p)$, where $p$ is of the form $g_2$: analogous to $P_1(a \odot p)$.
  - $d \gg (c \triangleright p)$, where $p$ is of the form $g_2$: analogous to $P_1(c \triangleright p)$.

- all newly introduced equations are of the form $l$ or of the form $d \gg X$ where $\text{rhs}(X)$ is of the form $l$: equations with right-hand sides of the following form might be introduced:
  - $\delta$, which is in $l$.
  - $[true] \gg a$, which is in $l$.
  - $d \gg a$, which is in $l$.
  - $d \gg X$: we know that $X$ was already present in $E$ before the transformation, so its right-hand side after transformation is of the form $l$ as shown above.

We conclude that after applying $P_1$ to the right-hand sides of all equations in $E$, these right-hand sides are either of the form $l$ or of the form $d \gg X$ where $\text{rhs}(X)$ is of the form $l$.

**Lemma 51** $\text{rewr}(d \gg p)$ is of the form $l$, for any $p$ of the form $l$.

**PROOF.** We prove this by induction on the structure of $d \gg p$, for $p$ of the form $l$.

- $d \gg \delta$: rewrites using (R14) to $\delta$, which is in $l$. 

50
• \(d \gg (d \gg a)\): rewrites using \((R17)\) to \((d \sim d) \gg a\), which is in \(l\).

• \(d \gg (d \gg (a \odot l_2))\): rewrites using \((R17)\) to \((d \sim d) \gg a \odot l_2\), which is in \(l\).

• \(d \gg (d \gg (c \triangleright l_2))\): rewrites using \((R17)\) to \((d \sim d) \gg c \triangleright l_2\), which is in \(l\).

• \(d \gg (p \oplus q)\): rewrites using \((R11)\) to \(\text{rewr}(d \gg p) \oplus \text{rewr}(d \gg q)\). Using the induction hypothesis, we see that both \(\text{rewr}(d \gg p)\) and \(\text{rewr}(d \gg q)\) are of the form \(l\). Since \(l \oplus l\) is in \(l\), we conclude that \(\text{rewr}(d \gg (p \oplus q))\) is in \(l\).

**PROOF. [Lemma 27]** Lemma 50 states that after applying \(P_1\) to the right-hand sides of all equations in \(E\), these right-hand sides are either of the form \(l\) or of the form \(d \gg X\) where \(\text{rhs}(X)\) is of the form \(l\).

Now, we prove that \(P_3(p)\) is of the form \(l\), for any \(p\) either of the form \(l\) or of the form \(d \gg X\) where \(\text{rhs}(X)\) is of the form \(l\):

• \(p\) of the form \(d \gg X\), where \(\text{rhs}(X)\) is of the form \(l\): \(P_3(p) = P_3(d \gg X) = \text{rewr}(d \gg \text{rhs}(X))\). Lemma 51 states that \(\text{rewr}(d \gg \text{rhs}(X))\) is of the form \(l\), because \(\text{rhs}(X)\) is of the form \(l\). Therefore, \(P_3(p)\) is of the form \(l\).

• \(p\) of the form \(l\): Since \(d \gg X\) is not in the form \(l\), \(P_3(p) = p\). Therefore, \(P_3(p)\) is of the form \(l\).

We conclude that after applying \(P_3\) to the right-hand sides of all equations in \(E\), the right-hand sides of all equations in \(E\) are semi-linear.

**Lemma 52** \(P_2(p) \approx p\) for any \(p\) of the form \(g_2\).

**PROOF.** We prove this by induction on the form \(g_2\):

• \(\delta\): \(P_2(\delta) = Y \approx \delta\).

• \(a\): \(P_2(a) = Y \approx [\text{true}] \gg a \approx a\).

• \(X\): \(P_2(X) = X\).

• \(p \odot q\), where \(p\) and \(q\) are of the form \(g_2\): applying the induction hypothesis twice, we see that \(P_2(p \odot q) = P_2(p) \odot P_2(q) \approx p \odot q\).

• \(d \gg a\): \(P_2(d \gg a) = Y \approx d \gg a\).

• \(d \gg X\): \(P_2(d \gg X) = Y \approx d \gg X\).

**PROOF. [Lemma 28]** First, we prove that \(P_1\) is sound, i.e. \(P_1(p) \approx p\) for any \(p\) of the form \(g\). We prove this by induction on the form \(g\):

• \(p \oplus q\), where \(p\) and \(q\) are of the form \(g\): applying the induction hypothesis twice, we see that \(P_1(p \oplus q) = P_1(p) \oplus P_1(q) \approx p \oplus q\).
• $\delta$: $P_1(\delta) = \delta$.
• $a$: $P_1(a) = [\text{true}] \gg a \approx a$.
• $d \gg a$: $P_1(d \gg a) = d \gg a$.
• $a \odot p$, where $p$ is of the form $g_2$: using Lemma 52 we see that $P_1(a \odot p) = [\text{true}] \gg (a \odot P_2(p)) \approx [\text{true}] \gg (a \odot p) \approx a \odot p$.
• $c \triangleright p$, where $p$ is of the form $g_2$: analogous to $a \odot p$.
• $d \gg (a \odot p)$, where $p$ is of the form $g_2$: analogous to $a \odot p$.
• $(d \gg a) \odot p$, where $p$ is of the form $g_2$: analogous to $a \odot p$.
• $d \gg (c \triangleright p)$, where $p$ is of the form $g_2$: analogous to $a \odot p$.

The rewrite system used in $P_3$ is sound, because all rewrite rules are directed versions of HyPA axioms.

Second, we prove that $P_3$ is sound, i.e. $P_3(p) \approx p$ for any $p$. We have two cases:

• $p$ of the form $d \gg X$: Since $\text{rewr}$ is sound, $P_3(p) = P_3(d \gg X) = \text{rewr}(d \gg \text{rhs}(X)) \approx d \gg \text{rhs}(X) \approx d \gg X$.
• $p$ not of the form $d \gg X$: $P_3(p) = p$.

We conclude that $P_1$ and $P_3$ are both sound.

**PROOF.** [Lemma 29] The rewrite system used in this step is a subset of the rewrite system used in the guard function. Since the rewrite system of the guard function is terminating, the rewrite system used in this step is terminating as well.

Well-definedness of $P_1$ and $P_3$ is trivial, since $P_1$ and $P_3$ do not occur in the right-hand sides of their definitions.

### A.2 Linearization Stage 2: From Semi-linear to Abstracted-linear

**Lemma 53** For any $s \in V_m$, re-initialization clause $d$ and constant expressions $C$, $C'$, where $s \not\in \text{Var}(d)$ and $\equiv$ denotes equivalence of re-initialization clauses:

\[
\begin{align*}
[s | s^+ = C] & \sim [s | s^+ = C'] & \equiv [s | s^+ = C'] & \quad (1) \\
\quad \quad d & \sim [s | s^+ = C] & \equiv [s | s^+ = C] & \sim d & \quad (2) \\
\quad \quad [s | s^+ = C] & \equiv [s | \text{true}] & \sim [s | s^+ = C] & \quad (3) \\
\quad \quad [s | \text{pop}(s^-) \neq \emptyset \land s^+ = C] & \equiv [\text{pop}(s^-) \neq \emptyset] & \sim [s | s^+ = C] & \quad (4)
\end{align*}
\]

**PROOF.** See Appendix B.1 from [19] for the proofs.
Lemma 54 For any $s \in \mathcal{V}_m$, $a \in \mathcal{A}$, $x \in \mathcal{T}$, re-initialization clauses $d$, $d_s$ and flow clause $c$, where $s \not\in \text{Var}(c) \cup \text{Var}(x)$ and $d_s = [s \mid s^+ = C]$ with $C$ a constant expression:

$$[[ s \mid d_s \gg x ]] \approx x \quad (1)$$

$$[[ s \mid (d \sim d_s) \gg a \odot x ]] \approx [[ s \mid d \gg a \odot (d_s \gg x) ]] \quad (2)$$

$$[[ s \mid (d \sim d_s) \gg (c \land (s \mid \hat{s} = 0)) \gg x ]] \approx [[ s \mid d \gg c \gg (d_s \gg x) ]] \quad (3)$$

PROOF. See Appendix B.2 from [19] for the proofs.

PROOF. [Lemma 30] It is straightforward to verify that the recursive equation $A$ is linear. Furthermore, it is straightforward to see that all other recursive equations in $E$ are transformed into abstracted-linear equations.

PROOF. [Lemma 31] Using the principle RSP, we prove that every $X_i \in E \setminus \{A\}$ before transformation is equivalent to $[[ s \mid [s \mid s^+ = \text{push}(X_i, \emptyset)] \gg A ]]$.

We first present some derivations, which are used later on in the proof.

For any $w_0, \ldots, w_n$, for $n \geq 0$ and where for all $w_i$, $X_{w_i} \in \text{Var}(E) \setminus \{A\}$:

$$[[ s \mid [s \mid s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \cdots)] \gg A ]]$$

$$\approx \{ \text{Substitute } A \text{ by its right-hand side, distribute } \gg \}$$

$$[[ s \mid \bigoplus_{X_i \in \text{Var}(E)} \bigoplus_{j \in J(i)} \begin{cases} [s \mid s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \cdots)] \sim \\ [s^+ \neq \emptyset \land \text{get}(s^-) = X_i \land \text{pop}(s^-) = \emptyset] \sim d_j \\
\end{cases} \gg a_j \bigoplus]$$

$$\approx a_j \bigoplus$$

$$[[ s \mid \bigoplus_{j \in J(i)} \begin{cases} [s \mid s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \cdots)] \sim \\ s^+ \neq \emptyset \land \text{get}(s^-) = X_i \land \text{pop}(s^-) \neq \emptyset \land s^+ = \text{pop}(s^-) \sim d_j \\
\end{cases} \gg a_j \bigoplus A \bigoplus]$$

$$\approx a_j \bigoplus A \bigoplus$$

$$[[ s \mid \bigoplus_{k \in K(i)} \begin{cases} [s \mid s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \cdots)] \sim \\
\end{cases} \gg a_k \bigoplus A \bigoplus]$$

$$\approx a_k \bigoplus A \bigoplus$$

53
$$\bigoplus_{l \in L(t)} \left( \begin{array}{l} [s \mid s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_i}, \emptyset) \ldots)] \sim \\
 s^- \neq \emptyset \land \text{get}(s^-) = X_l \land \\
 s^+ = \text{push}(X_{f(l,1)}, \ldots \text{push}(X_{f(l,n)}) \text{pop}(s^-) \ldots) \sim d_l \\
 \end{array} \right) \gg (c_l \land (s \| s = 0)) \triangleright A$$

$$\approx \{ \text{Eliminate sum: all summands are } \delta \text{ except for summands of } X_{w_0} \}$$

$$[[ s ] \bigoplus_{j \in J(w_0)} \left( \begin{array}{l} [s \mid s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_i}, \emptyset) \ldots)] \sim \\
 s^- \neq \emptyset \land \text{get}(s^-) = X_{w_0} \land \text{pop}(s^-) = \emptyset] \sim d_j \\
 s^+ = \text{push}(X_{f(l,1)}, \ldots \text{push}(X_{f(l,n)}) \text{pop}(s^-) \ldots) \sim d_j \end{array} \right) \gg a_j \odot A \odot$$

$$\bigoplus_{k \in K(w_0)} \left( \begin{array}{l} s^- \neq \emptyset \land \text{get}(s^-) = X_{w_0} \land \\
 s^+ = \text{push}(X_{f(k,1)}, \ldots \text{push}(X_{f(k,n)}) \text{pop}(s^-) \ldots) \sim d_k \\
 \end{array} \right) \gg a_k \odot A \odot$$

$$\bigoplus_{l \in L(w_0)} \left( \begin{array}{l} s^- \neq \emptyset \land \text{get}(s^-) = X_{w_0} \land \\
 s^+ = \text{push}(X_{f(l,1)}, \ldots \text{push}(X_{f(l,n)}) \text{pop}(s^-) \ldots) \sim d_l \\
 \end{array} \right) \gg (c_l \land (s \| s = 0)) \triangleright A$$

$$\approx \{ \text{Calculation on re-initialization clauses } \}$$
\[
\begin{align*}
\oplus_{k \in K(w_0)} & \left( \left[ s \mid s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots) \right] \sim s \mid s^+ = \text{push}(X_{f(k,1)}, \ldots \text{push}(X_{f(k,n_k)}, \ldots) \text{push}(X_{w_1}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots) \ldots) \sim d_k \right) \Rightarrow a_k \odot A \oplus \\
\oplus_{l \in L(w_0)} & \left( \left[ s \mid s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots) \right] \sim s \mid s^+ = \text{push}(X_{f(l,1)}, \ldots \text{push}(X_{f(l,n_l)}, \ldots) \text{push}(X_{w_1}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots) \ldots) \sim d_l \right) \Rightarrow (c_l \land (s \mid \hat{s} = 0)) \triangleright A
\end{align*}
\]

\[
\approx \{ \text{Lemma 53 (1) and (4)} \}
\]

\[
\begin{align*}
\oplus_{j \in J(w_0)} & \left( \left[ s \mid s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots) \right] \sim \left[ \text{pop}(s^-) = \emptyset \right] \sim d_j \right) \Rightarrow a_j \oplus \\
\oplus_{j \in J(w_0)} & \left( \left[ s \mid s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots) \right] \sim \left[ \text{pop}(s^-) \neq \emptyset \right] \sim d_j \right) \Rightarrow a_j \odot A \oplus \\
\oplus_{k \in K(w_0)} & \left( s \mid s^+ = \text{push}(X_{f(k,1)}, \ldots \text{push}(X_{f(k,n_k)}, \ldots) \text{push}(X_{w_1}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots) \ldots) \sim d_k \Rightarrow a_k \odot A \oplus \\
\oplus_{l \in L(w_0)} & \left( s \mid \text{push}(X_{f(l,1)}, \ldots \text{push}(X_{f(l,n_l)}, \ldots) \text{push}(X_{w_1}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots) \ldots) \sim d_l \Rightarrow (c_l \land (s \mid \hat{s} = 0)) \triangleright A
\end{align*}
\]

\[
\approx \{ \text{Distribute abstraction} \}
\]
\[ \oplus_{k \in K(w_0)} \left[ \begin{array}{c|c} s & \begin{array}{c} s^+ = \text{push}(X_{f(k,1)}, \ldots \text{push}(X_{f(k,n_k)}, \text{push}(X_{w_1}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots)) \ldots) \sim d_k \Rightarrow a_k \odot A \end{array} \\
\end{array} \right] \]

\[ \oplus_{l \in L(w_0)} \left[ \begin{array}{c|c} s & \begin{array}{c} s^+ = \text{push}(X_{f(l,1)}, \ldots \text{push}(X_{f(l,n_l)}, \text{push}(X_{w_1}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots)) \ldots) \sim d_l \Rightarrow (c_l \land (s \hat{s} = 0)) \triangleright A \\
\end{array} \\
\end{array} \right] \]

\[ \approx \{ \text{Lemma 53 (2)} \} \]

\[ \oplus_{j \in J(w_0)} \left[ \begin{array}{c|c} s & \begin{array}{c} [s | s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots)] \sim \Rightarrow a_j \\
\end{array} \\
[\text{pop}(s^-) = \emptyset] \sim d_j \end{array} \right] \]

\[ \oplus_{j \in J(w_0)} \left[ \begin{array}{c|c} s & \begin{array}{c} [s | s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots)] \\
\end{array} \\
\sim [\text{pop}(s^-) \neq \emptyset] \sim d_j \sim \Rightarrow a_j \odot A \\
\end{array} \right] \]

\[ \oplus_{k \in K(w_0)} \left[ \begin{array}{c|c} s & \begin{array}{c} d_k \sim [s | s^+ = \text{push}(X_{f(k,1)}, \ldots \text{push}(X_{f(k,n_k)}, \text{push}(X_{w_1}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots)) \ldots)] \Rightarrow a_k \odot A \\
\end{array} \\
\end{array} \right] \]

\[ \oplus_{l \in L(w_0)} \left[ \begin{array}{c|c} s & \begin{array}{c} d_l \sim [s | s^+ = \text{push}(X_{f(l,1)}, \ldots \text{push}(X_{f(l,n_l)}, \text{push}(X_{w_1}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots)) \ldots)] \\
\end{array} \\
\Rightarrow (c_l \land (s \hat{s} = 0)) \triangleright A \\
\end{array} \right] \]

\[ \approx \{ \text{Lemma 54 (2) and (3)} \} \]

\[ \oplus_{j \in J(w_0)} \left[ \begin{array}{c|c} s & \begin{array}{c} [s | s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots)] \sim \Rightarrow a_j \\
\end{array} \\
[\text{pop}(s^-) = \emptyset] \sim d_j \end{array} \right] \]

\[ \oplus_{j \in J(w_0)} \left[ \begin{array}{c|c} s & \begin{array}{c} [s | s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots)] \\
\end{array} \\
\sim [\text{pop}(s^-) \neq \emptyset] \sim d_j \sim \Rightarrow a_j \odot A \\
\end{array} \right] \]

\[ \oplus_{j \in J(w_0)} \left[ \begin{array}{c|c} s & \begin{array}{c} \odot [s | s^+ = \text{push}(X_{w_1}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots)] \Rightarrow A \\
\end{array} \\
\end{array} \right] \]

\[ \oplus_{k \in K(w_0)} \left[ \begin{array}{c|c} s & \begin{array}{c} d_k \Rightarrow a_k \odot [s | s^+ = \text{push}(X_{f(k,1)}, \ldots \text{push}(X_{f(k,n_k)}, \text{push}(X_{w_1}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots)) \ldots)] \Rightarrow A \\
\end{array} \\
\end{array} \right] \]

\[ \oplus_{l \in L(w_0)} \left[ \begin{array}{c|c} s & \begin{array}{c} d_l \Rightarrow c_l \triangleright [s | s^+ = \text{push}(X_{f(l,1)}, \ldots \text{push}(X_{f(l,n_l)}, \text{push}(X_{w_1}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots)) \ldots)] \Rightarrow A \\
\end{array} \\
\end{array} \right] \]
\[ \approx \{ \text{Lemma 53 (3)} \} \]

\[ \bigoplus_{j \in J(w_0)} \left[ s \mid \left( [s]^{s^+ = \text{push}(X_{w_0}, \ldots, \text{push}(X_{w_n}, \emptyset) \ldots)] \sim a_j \right) \right] \bigoplus \]

\[ \left[ s \mid [s]^{s^+ = \text{push}(X_{w_0}, \ldots, \text{push}(X_{w_n}, \emptyset) \ldots]} \sim \bigoplus_{j \in J(w_0)} [\text{pop}(s^-) = \emptyset] \sim d_j \right] \bigoplus \]

\[ \left[ s \mid [s]^{s^+ = \text{push}(X_{w_0}, \ldots, \text{push}(X_{w_n}, \emptyset) \ldots]} \sim \bigoplus_{j \in J(w_0)} [\text{pop}(s^-) \neq \emptyset] \sim d_j \right] \bigoplus \]

\[ \left[ s \mid [s]^{s^+ = \text{push}(X_{w_0}, \ldots, \text{push}(X_{w_n}, \emptyset) \ldots]} \Rightarrow A \right] \bigoplus \]

\[ \left[ s \mid [s]^{s^+ = \text{push}(X_{w_0}, \ldots, \text{push}(X_{w_n}, \emptyset) \ldots]} \approx A \right] \bigoplus \]

\[ \approx \{ \text{Distribution of abstraction (VA2, VA3)} \} \]

\[ \bigoplus_{j \in J(w_0)} \left[ s \mid [s]^{s^+ = \text{push}(X_{w_0}, \ldots, \text{push}(X_{w_n}, \emptyset) \ldots]} \sim [\text{pop}(s^-) = \emptyset] \sim d_j \right] \bigoplus \]

\[ \left[ s \mid [s]^{s^+ = \text{push}(X_{w_0}, \ldots, \text{push}(X_{w_n}, \emptyset) \ldots]} \sim \bigoplus_{j \in J(w_0)} [\text{pop}(s^-) \neq \emptyset] \sim d_j \right] \bigoplus \]

\[ \left[ s \mid [s]^{s^+ = \text{push}(X_{w_0}, \ldots, \text{push}(X_{w_n}, \emptyset) \ldots]} \Rightarrow A \right] \bigoplus \]

\[ \left[ s \mid [s]^{s^+ = \text{push}(X_{w_0}, \ldots, \text{push}(X_{w_n}, \emptyset) \ldots]} \approx A \right] \bigoplus \]

\[ \approx \{ \text{Case distinction on } n \} \]

If \( n = 0 \):

\[ \bigoplus_{j \in J(w_0)} \left[ s \mid [s]^{s^+ = \text{push}(X_{w_0}, \emptyset) \sim d_j \Rightarrow a_j } \right] \bigoplus \]
$$\bigodot_{k \in K(w_0)} \left[ s \mid d_k \gg a_k \right] \bigodot$$
$$\bigodot_{l \in L(w_0)} \left[ s \mid d_l \gg c_l \right] \triangleright$$

If \( n > 0 \):
$$\bigodot_{j \in J(w_0)} \left[ s \mid [s \, s^+ = \text{push}(X_{w_0}, \ldots \text{push}(X_{w_n}, \emptyset) \ldots)] \sim d_j \gg a_j \right] \bigodot$$
$$\bigodot_{k \in K(w_0)} \left[ s \mid [s \, s^+ = \text{push}(X_{f(k,1)}, \ldots \text{push}(X_{f(k,n_k)}, \emptyset) \ldots)] \gg A \right] \bigodot$$
$$\bigodot_{l \in L(w_0)} \left[ s \mid [s \, s^+ = \text{push}(X_{f(l,1)}, \ldots \text{push}(X_{f(l,n_l)}, \emptyset) \ldots)] \gg A \right] \bigodot$$

$$\approx \{ \text{Lemma 54 (1) and the axiom } [V \mid x] \approx x, \text{ if } Var(x) \cap V = \emptyset \}$$

If \( n = 0 \):
$$\bigodot_{j \in J(w_0)} d_j \gg a_j \bigodot$$
$$\bigodot_{k \in K(w_0)} d_k \gg a_k \bigodot$$
$$\bigodot_{l \in L(w_0)} [s \mid [s \, s^+ = \text{push}(X_{f(k,1)}, \ldots \text{push}(X_{f(k,n_k)}, \emptyset) \ldots)] \gg A \right] \bigodot$$
$$\bigodot_{l \in L(w_0)} [s \mid [s \, s^+ = \text{push}(X_{f(l,1)}, \ldots \text{push}(X_{f(l,n_l)}, \emptyset) \ldots)] \gg A \right] \bigodot$$

If \( n > 0 \):
$$\bigodot_{j \in J(w_0)} d_j \gg a_j \bigodot$$
$$\bigodot_{k \in K(w_0)} d_k \gg a_k \bigodot$$
$$\bigodot_{l \in L(w_0)} [s \mid [s \, s^+ = \text{push}(X_{f(k,1)}, \ldots \text{push}(X_{f(k,n_k)}, \emptyset) \ldots)] \gg A \right] \bigodot$$
$$\bigodot_{l \in L(w_0)} [s \mid [s \, s^+ = \text{push}(X_{f(l,1)}, \ldots \text{push}(X_{f(l,n_l)}, \emptyset) \ldots)] \gg A \right] \bigodot$$

58
\[ X_{w_0} \circ \ldots \circ X_{w_n} \]
\[
\begin{cases}
\bigoplus_{j \in J(w_0)} d_j \gg a_j \oplus \\
\bigoplus_{k \in K(w_0)} d_k \gg a_k \circ X_{f(k,1)} \circ \ldots \circ X_{f(k,n)} \oplus \\
\bigoplus_{l \in L(w_0)} d_l \gg c_l \triangleright X_{f(l,1)} \circ \ldots \circ X_{f(l,n)}
\end{cases}
\]
\[ \approx \]
\[
\begin{cases}
\bigoplus_{j \in J(w_0)} d_j \gg a_j \circ X_{w_1} \circ \ldots \circ X_{w_n} \oplus \\
\bigoplus_{k \in K(w_0)} d_k \gg a_k \circ X_{f(k,1)} \circ \ldots \circ X_{f(k,n)} \circ \\
\bigoplus_{l \in L(w_0)} d_l \gg c_l \triangleright X_{f(l,1)} \circ \ldots \circ X_{f(l,n)}
\end{cases}
\]

We see that \( \left[ s \mid [s]^{+} = \text{push}(X_{w_0}, \ldots, \text{push}(X_{w_n}, \emptyset) \cdots) \gg A \right] \) as well as \( X_{w_0} \circ \ldots \circ X_{w_n} \) are solutions of \( Y_{w_0 \ldots w_n} \). Since all these equations are guarded, we can apply RSP, so we have that

\[ X_{w_0} \circ \ldots \circ X_{w_n} \approx \left[ s \mid [s]^{+} = \text{push}(X_{w_0}, \ldots, \text{push}(X_{w_n}, \emptyset) \cdots) \gg A \right]. \]

From this we conclude that it is sound to transform the right-hand sides of all equations \( X_i \in E \setminus \{A\} \) into \( \left[ s \mid [s]^{+} = \text{push}(X_i, \emptyset) \gg A \right] \), so the transformation in this stage is sound.

A.3 Linearization Stage 3: Transforming the Initial Term

A.3.1 Eliminating Parallel Composition

PROOF. [Lemma 33] The result of \textit{elim\_par\_comp} is the single recursion variable \(Z\). \(Z\) is introduced in step 2 above and it is straightforward to verify that \(Z\) is abstracted-linear at that point. In step 3, \(Z\) becomes the abstraction of the abstracted-linear variable \(Z_1\). Then in step 4 this abstraction is merged with the abstraction in \(Z_1\), which makes \(Z\) abstracted-linear again. Finally, step 5 does not affect \(Z\), so the resulting recursion variable \(Z\) is abstracted-linear.

By following the steps 1 to 5 above it is straightforward to see that, after transformation, indeed all recursive equations in \(E\) are abstracted-linear except for the single equation \(C\) which is linear.

PROOF. [Lemma 34] We prove soundness of \textit{elim\_par\_comp} by proving that each step is sound:

(1) Only axioms of abstraction are used.
(2) It is tedious but straightforward to verify that \(Z\) is indeed equal to the parallel composition of \(X\) and \(Y\).
(3) Soundness of this transformation was proven to be sound in the previous section.
(4) Only axioms of abstraction are used.
(5) As can be seen in the contents of \(E\) after step 4, \(A\) and \(B\) are not used anymore, so they can be deleted without affecting any other equations.

We conclude that \textit{elim\_par\_comp} is sound.

The result of the \textit{elim\_par\_comp} function is the single recursion variable \(Z\). In step 2, \(Z\) is defined such that it is robustly bisimilar to \(X \parallel Y\). Every subsequent step either does not affect \(Z\) at all or it transforms \(Z\) into an equivalent process. Therefore, \textit{elim\_par\_comp}(\(X, Y\)) = \(Z \approx X \parallel Y\).

PROOF. [Lemma 35] The result of the computation of \(Z\) in step 2 is finite. The transformation from linear equations to abstracted-linear equations in step 3 was proven to be well-defined in the previous section. Well-definedness of the other steps is trivial. Therefore, \textit{elim\_par\_comp} is well-defined.
A.3.2 Eliminating Encapsulation

Lemma 55 \( B \approx \partial_H (A) \).

**PROOF.** We prove this using RSP, by showing that \( \partial_H (A) \) and \( B \) are both solutions of another guarded recursive equation.

\[
\partial_H (A) \approx \partial_H (\text{rhs}(A))
\]

\[
\approx \partial_H \left( \bigoplus_{j \in J(A)} d_j \gg a_j \oplus \bigoplus_{k \in K(A)} d_k \gg a_k \odot A \oplus \bigoplus_{l \in L(A)} d_l \gg c_l \triangleright A \right)
\]

\[
\approx \bigoplus_{j \in J(A)} \partial_H (d_j \gg a_j) \oplus \bigoplus_{k \in K(A)} \partial_H (d_k \gg a_k \odot A) \oplus \bigoplus_{l \in L(A)} \partial_H (d_l \gg c_l \triangleright A)
\]

\[
B \approx \text{elim_encap}_2(H, \text{rhs}(A))
\]

\[
\approx \text{elim_encap}_2 \left( H, \bigoplus_{k \in K(A)} d_k \gg a_k \odot A \oplus \bigoplus_{l \in L(A)} d_l \gg c_l \triangleright A \right)
\]

\[
\approx \bigoplus_{j \in J(A)} \text{elim_encap}_2(H, d_j \gg a_j) \oplus \bigoplus_{k \in K(A)} \text{elim_encap}_2(H, d_k \gg a_k \odot A) \oplus \bigoplus_{l \in L(A)} \text{elim_encap}_2(H, d_l \gg c_l \triangleright A)
\]
\[
Y \approx \bigoplus_{j \in J(A) \land a_j \notin H} d_j \gg a_j \oplus \\
\bigoplus_{k \in K(A) \land a_k \notin H} d_k \gg a_k \odot B \oplus \\
\bigoplus_{l \in L(A)} d_l \gg c_l \triangleright B
\]

We define the equation \( Y \) as follows:

\[
Y \approx \bigoplus_{j \in J(A) \land a_j \notin H} d_j \gg a_j \oplus \\
\bigoplus_{k \in K(A) \land a_k \notin H} d_k \gg a_k \odot Y \oplus \\
\bigoplus_{l \in L(A)} d_l \gg c_l \triangleright Y
\]

We see that both \( \partial_H(A) \) and \( B \) are a solution of the guarded equation \( Y \), so by RSP we conclude that \( B \approx \partial_H(A) \).

**PROOF.** [Lemma 36] The result of \( \text{elim\_par\_comp} \) is the single recursion variable \( Z \). In the proof of Lemma 55 we see that

\[
B \approx \bigoplus_{j \in J(A) \land a_j \notin H} d_j \gg a_j \oplus \\
\bigoplus_{k \in K(A) \land a_k \notin H} d_k \gg a_k \odot B \oplus \\
\bigoplus_{l \in L(A)} d_l \gg c_l \triangleright B
\]

Clearly, \( B \) is linear after step 1. Since \( Z \approx [S \mid d_{X} \gg B] \), we conclude that \( Z \) is abstracted-linear after step 1. In step 2, \( Z \) becomes the abstraction of the abstracted-linear variable \( B \). Then in step 3 this abstraction is merged with the abstraction in \( B \), which makes \( Z \) abstracted-linear again. Finally, step 4 does not affect \( Z \), so the resulting recursion variable \( Z \) is abstracted-linear.

By following the steps 1 to 4 above it is straightforward to see that, after transformation, indeed all recursive equations in \( E \) are abstracted-linear except for the single equation \( C \) which is linear.

**PROOF.** [Lemma 37] We prove soundness of \( \text{elim\_encap} \) by proving that each step is sound:
(1) We prove that $Z \approx \partial_H (X)$, using Lemma 55:

\[
Z \approx \left[ S \mid dX \gg B \right] \\
\approx \left[ S \mid dX \gg \partial_H (A) \right] \\
\approx \left[ S \mid \partial_H (dX \gg A) \right] \\
\approx \partial_H \left( \left[ S \mid dX \gg A \right] \right) \\
\approx \partial_H (X)
\]

decap(H, X) = Z and $Z \approx \partial_H (X)$, so decap(H, X) $\approx \partial_H (X)$ for any abstracted-linear recursion variable $X$.

(2) Soundness of this transformation was proven in the previous section.

(3) Only axioms of abstraction are used.

(4) As can be seen in the contents of $E$ after step 3, $A$ and $B$ are not used anymore, so they can be deleted without affecting any other equations.

We conclude that decap is sound.

PROOF. [Lemma 38] The transformation from linear equations to abstracted-linear equations was proven to be well-defined in the previous section. Well-definedness of the other steps is trivial. Therefore, decap is well-defined.

A.3.3 T₁ and the Final Transformation

PROOF. [Lemma 39] The result of $T₂(p)$ is always a single recursion variable, say $X$, whose right-hand side is abstracted-linear. Therefore, the result of $T₁(p)$ is always an abstraction of $X$, i.e. $\left[ V \mid X \right]$. Then we have,

\[
t = \left[ V \mid X \right] \approx \left[ V \mid \left[ W \mid d \gg A \right] \right] \approx \left[ V \cup W \mid d \gg A \right],
\]

where $A$ is linear. Clearly, the resulting specification $⟨ t \mid \{ A \} ⟩$ is linear.

PROOF. [Lemma 40] First, we prove that $T₂(p) \approx p$ for any $p$ is of the form $HyPA_{par}$, by induction on the structure of the form $HyPA_{par}$:

- $X$: trivial.
- $p \parallel q$, with $p$ and $q$ of the form $HyPA_{par}$: Applying the induction hypothesis twice gives $T₂(p) \approx p$ and $T₂(q) \approx q$. Furthermore, it is straightforward to see that $T₂(r)$ always returns a single abstracted-linear recursion variable $X$ for any term $r$ of the form $HyPA_{par}$. Therefore, using Lemma 34, $T₂(p \parallel q) = decap_{comp}(p, q) \approx p \parallel q$. 

63
• $\partial_H(p)$: Applying the induction hypothesis gives $T_2(p) \approx p$. Furthermore, it is straightforward to see that $T_2(r)$ always returns a single abstracted-linear recursion variable $X$ for any term $r$ of the form $\text{HyPA}_{\text{par}}$. Therefore, using Lemma 37, $T_2(\partial_H(p)) = \text{elim_encap}(H, p) \approx \partial_H(p)$.

Now, we prove that $T_1(p) \approx p$ for any $p$ is of the form $\text{HyPA}_{\text{par}}$:

• $p$ is of the form $[V \mid q]$, where $q$ is of the form $\text{HyPA}_{\text{par}}$:

$$T_1(p) = T_1([V \mid q]) = [V \mid T_2(q)] \approx [V \mid q] \approx p.$$

• $p$ is of not the form $[V \mid q]$: $T_1(p) = [\emptyset \mid T_2(p)] \approx [\emptyset \mid p] \approx p$.

Therefore, $T_1$ is sound. It is trivial to see that the final transformation $t = [V \mid X] \approx [V \mid [W \mid d \gg A]] \approx [V \cup W \mid d \gg A]$ is sound.

**PROOF.** [Lemma 41] It is straightforward to see that $T_2$ is well-defined, because recursion in the right-hand side only occurs on strictly smaller sub terms. Since $T_2$ is well-defined, it is trivial to see that $T_1$ is well-defined.

### A.4 Optimization: Creating a Single Stack Clause per Summand

**PROOF.** [Theorem 43] Lemma 33 says that, after elimination of a parallel composition, the specification contains a single linear equation. First we show which forms the summands in this linear equation may have and then we show that each of these forms can be transformed into the form $(s \sim d) \gg x$.

As explained in Section 4.3.3, this linear equation is created by merging the processes $A$, $B$ and $Z_1$ to $Z_{11,l}$. The re-initialization clauses in the processes $A$, $B$ and $Z_1$ to $Z_{11,l}$ are of one of the following forms:

$$s \sim d \gg x \quad (s \sim s' \sim d) \land (s'' \sim s''' \sim d') \gg x$$

$$s \sim (s' \sim d) \gg x \quad (s \sim d \sim d') \land (s' \sim d'' \sim d''') \gg x$$

$$(s \sim d \sim d') \land d'' \gg x \quad (s \sim s' \sim d \sim d') \land (s'' \sim s''' \sim d'' \sim d''') \gg x$$

$$(s \sim d) \land (s' \sim d') \gg x$$

Then, when these equations are merged into one single linear equation, all summands are prefixed with a re-initialization clause that only contains a predicate on the newly introduced stack variable. Therefore, the summands in the new linear equation are of one of the following forms:

$$(1) \quad s' \sim (s \sim d) \gg x$$
Now, we show that each of these forms can be transformed into the form 
\((s \sim d) \gg x\), using only Conjecture 42, associativity of concatenation and 
conjunction, and commutativity of conjunction:

(1) \(s' \sim (s \sim d) \gg x \approx (s' \sim s) \sim d \gg x\)

(2) \(s''' \sim ((s \sim s') \sim d) \sim (s'' \sim s'' \sim d') \gg x\)
\[
\approx s''' \sim ((s \sim s') \sim d) \sim (s'' \sim s'' \sim d') \gg x
\]
\[
\approx s''' \sim ((s \sim s') \sim s'' \sim s'' \sim d') \sim (d \sim d') \gg x
\]
\[
\approx (s''' \sim ((s \sim s') \sim s'' \sim s'' \sim d')) \sim (d \sim d') \gg x
\]
(3) \(s'' \sim (s \sim (s' \sim d)) \gg x \approx (s'' \sim s \sim s') \sim d \gg x\)

(4) \(s'' \sim ((s \sim d \sim d') \sim (s' \sim d' \sim d'' \sim d''')) \gg x\)
\[
\approx s'' \sim ((s \sim d \sim d') \sim (s' \sim d' \sim d'' \sim d''')) \gg x
\]
\[
\approx s'' \sim ((s \sim s') \sim (d \sim d' \sim d'' \sim d''')) \gg x
\]
\[
\approx (s'' \sim (s \sim s')) \sim (d \sim d' \sim d'' \sim d''')) \gg x
\]
(5) \(s' \sim ((s \sim d \sim d') \sim d'' \sim d''') \gg x\)
\[
\approx s' \sim ((s \sim (d \sim d')) \sim d'' \sim d''') \gg x
\]
\[
\approx s' \sim (s \sim (d \sim d') \sim d'' \sim d''') \gg x
\]
\[
\approx (s' \sim s) \sim (d \sim d' \sim d'' \sim d''') \gg x
\]
(6) \(s''' \sim ((s \sim s' \sim d \sim d') \sim (s'' \sim s'' \sim d'' \sim d''')) \gg x\)
\[
\approx s''' \sim ((s \sim s') \sim (d \sim d') \sim (s'' \sim s'' \sim d'' \sim d''')) \gg x
\]
\[
\approx s''' \sim ((s \sim s') \sim s'' \sim s'' \sim d' \sim d'' \sim d''')) \sim (d \sim d' \sim d'' \sim d''') \gg x
\]
\[
\approx (s''' \sim ((s \sim s') \sim s'' \sim s'' \sim d' \sim d'' \sim d''')) \sim (d \sim d' \sim d'' \sim d''') \gg x
\]
(7) \(s'' \sim ((s \sim d) \sim (s' \sim d')) \gg x\)
\[
\approx s'' \sim ((s \sim s') \sim (d \sim d') \gg x
\]
\[
\approx (s'' \sim (s \sim s')) \sim (d \sim d') \gg x
\]
B Parallel Composition of $X$ and $Y$

\[
Z \approx X \parallel Y \\
\approx \left[ S \mid d_X \gg A \right] \parallel \left[ T \mid d_Y \gg B \right] \\
\approx \left[ S \mid d_X \gg A \right] \parallel \left[ T \mid d_Y \gg B \right] \\
\approx \left[ S \cup T \mid d_X \gg A \parallel d_Y \gg B \right] \\
\approx \left[ S \cup T \mid \{true\} \gg Z_1 \right]
\]

\[
Z_1 \approx d_X \gg A \parallel d_Y \gg B \\
\approx \bigoplus_{j \in J(A)} (d_X \sim d_j) \gg a_j \odot Z_6 \oplus \bigoplus_{k \in K(A)} (d_X \sim d_k) \gg a_k \odot Z_2 \oplus \bigoplus_{j' \in J(B)} (d_Y \sim d_{j'}) \gg a_{j'} \odot Z_5 \oplus \bigoplus_{k' \in K(B)} (d_Y \sim d_{k'}) \gg a_{k'} \odot Z_3 \oplus \bigoplus_{j \in J(A)} \bigoplus_{j' \in J(B)} ((d_X \sim d_j) \land (d_Y \sim d_{j'})) \gg \gamma(a_j, a_{j'}) \oplus \bigoplus_{k \in K(A)} \bigoplus_{j' \in J(B)} ((d_X \sim d_k) \land (d_Y \sim d_{j'})) \gg \gamma(a_k, a_{j'}) \cap A \oplus \bigoplus_{j \in J(A)} \bigoplus_{k' \in K(B)} ((d_X \sim d_j) \land (d_Y \sim d_{k'})) \gg \gamma(a_j, a_{k'}) \cap B \oplus \bigoplus_{k \in K(A)} \bigoplus_{k' \in K(B)} ((d_X \sim d_k) \land (d_Y \sim d_{k'})) \gg \gamma(a_k, a_{k'}) \cap Z_4 \oplus \bigoplus_{l \in L(A)} \bigoplus_{l' \in L(B)} (((d_X \sim d_l \sim c_{l,jm_p}) \land (d_Y \sim d_{l'} \sim c_{l'jmp}))) \gg (c_l \land c_{l'}) \triangleright Z_{7,l,l'}
\]

\[
Z_2 \approx A \parallel (d_Y \gg B), \text{ which is similar to } Z_1
\]

\[
Z_3 \approx (d_X \gg A) \parallel B, \text{ which is similar to } Z_1
\]

\[
Z_4 \approx A \parallel B, \text{ which is similar to } Z_1
\]

\[
Z_5 \approx d_X \gg A \\
\approx \bigoplus_{j \in J(A)} (d_X \sim d_j) \gg a_j \oplus \bigoplus_{k \in K(A)} (d_X \sim d_k) \gg a_k \odot A \oplus
\]

66
\[ \bigoplus_{l \in L(A)} (d_X \sim d_l) \gg c_l \gg A \]

\[ Z_6 \approx d_Y \gg B, \text{ which is similar to } Z_5 \]

\[ Z_{7,l,l'} \approx A \parallel (c_{l'} \gg B) \oplus B \parallel (c_l \gg A) \oplus A \parallel (c_{l'} \gg B) \oplus B \parallel (c_l \gg A) \]
\[ \approx \bigoplus_{j \in J(A)} d_j \gg a_j \circ Z_{8,l'} \oplus \bigoplus_{k \in K(A)} d_k \gg a_k \circ Z_{9,l'} \oplus \bigoplus_{j' \in J(B), k' \in K(B)} d_{j'} \gg a_{j'} \circ Z_{10,l} \oplus \bigoplus_{k' \in K(B)} (d_{j''} \sim c_{l''} \wedge c_{l_{j''}}) \gg (c_{l''} \wedge c_{l'}) \gg Z_{7,l',l''} \oplus \bigoplus_{l'' \in L(A)} (d_{j''} \sim c_{l_{j''}}) \gg (c_{l''} \wedge c_l) \gg Z_{7,l',l''} \oplus \bigoplus_{l'' \in L(B)} (d_{j''} \sim c_{l_{j''}}) \gg (c_{l''} \wedge c_l) \gg Z_{7,l',l''} \oplus \bigoplus_{j \in J(A), j' \in J(B)} (d_j \land d_{j'}) \gg \gamma(a_j, a_{j'}) \oplus \bigoplus_{k \in K(A), j' \in J(B)} (d_k \land d_{j'}) \gg \gamma(a_k, a_{j'}) \circ A \oplus \bigoplus_{j \in J(A), k \in K(B), k' \in K(B)} (d_k \land d_{k'}) \gg \gamma(a_k, a_{k'}) \circ Z_4 \oplus \bigoplus_{l'' \in L(A), l''' \in L(B)} ((d_{j''} \sim c_{l_{j''}}) \wedge (d_{j'''} \sim c_{l_{j'''}})) \gg (c_{l''} \wedge c_{l''}) \gg Z_{7,l',l''} \oplus \]

\[ Z_{8,l'} \approx c_{l'} \gg B \]

\[ \approx \text{[true]} \gg c_{l'} \gg B \oplus \text{rhs}(B) \]

\[ Z_{9,l'} \approx A \parallel (c_{l'} \gg B) \]
\[ \approx \bigoplus_{j \in J(A)} d_j \gg a_j \circ Z_{8,l'} \oplus \bigoplus_{k \in K(A)} d_k \gg a_k \circ Z_{9,l'} \oplus \bigoplus_{j' \in J(B)} d_{j'} \gg a_{j'} \circ A \oplus \bigoplus_{k' \in K(B)} d_{k'} \gg a_{k'} \circ Z_4 \oplus \]
\( \bigoplus \bigoplus (d_j \land d_{j'}) \gg \gamma(a_j, a_{j'}) \bigoplus \)
\( j \in J(A) \land j' \in J(B) \)

\( \bigoplus \bigoplus (d_k \land d_{k'}) \gg \gamma(a_k, a_{k'}) \odot A \bigoplus \)
\( k \in K(A) \land k' \in K(B) \)

\( \bigoplus \bigoplus (d_j \land d_{k'}) \gg \gamma(a_j, a_{k'}) \odot B \bigoplus \)
\( j \in J(A) \land k' \in K(B) \)

\( \bigoplus \bigoplus (d_k \land d_{k'}) \gg \gamma(a_k, a_{k'}) \odot Z_4 \bigoplus \)
\( k \in K(A) \land k' \in K(B) \)

\( \bigoplus \bigoplus ((d_{l''} \sim c_{l''}^{jmp}) \land (d_{l'''} \sim c_{l'''}^{jmp})) \gg (c_{l''} \land c_{l'''}^{jmp}) \bigoplus \)
\( l'' \in L(A) \land l''' \in L(B) \)

\( \bigoplus ((d_{l''} \sim c_{l''}^{jmp}) \land c_{l''}^{jmp}) \gg (c_{l''} \land c_{l''}^{jmp}) \bigoplus \)
\( l'' \in L(A) \)

\( Z_{10,l} \approx c_l \bigoplus A, \) which is similar to \( Z_{8,l'} \)

\( Z_{11,l} \approx B \parallel (c_l \bigoplus A), \) which is similar to \( Z_{9,l'} \)