

State Space Reduction of Linear Processes using Control Flow Reconstruction

Jaco van de Pol and Mark Timmer*

University of Twente, Department of Computer Science, The Netherlands
Formal Methods & Tools
{pol, timmer}@cs.utwente.nl

Abstract. We present a new method for fighting the state space explosion of process algebraic specifications, by performing static analysis on an intermediate format: linear process equations (LPEs). Our method consists of two steps: (1) we reconstruct the LPE’s control flow, detecting control flow parameters that were introduced by linearisation as well as those already encoded in the original specification; (2) we reset parameters found to be irrelevant based on data flow analysis techniques similar to traditional liveness analysis, modified to take into account the parallel nature of the specifications. Our transformation is correct with respect to strong bisimilarity, and never increases the state space. Case studies show that impressive reductions occur in practice, which could not be obtained automatically without reconstructing the control flow.

1 Introduction

Our society depends heavily on computer systems, asking increasingly for methods to verify their correctness. One successful approach is *model checking*; performing an exhaustive state space exploration. However, for concurrent systems this approach suffers from the infamous *state space explosion*, an exponential growth of the number of reachable states. Even a small system specification can give rise to a gigantic, or even infinite, state space. Therefore, much attention has been given to methods for reducing the state space.

It is often inefficient to first generate a state space and then reduce it, since most of the complexity is in the generation process. As a result, intermediate symbolic representations such as Petri nets and linear process equations (LPEs) have been developed, upon which reductions can be applied. We concentrate on LPEs, the intermediate format of the process algebraic language μCRL [13]. Although LPEs are a restricted part of μCRL , every specification can be transformed to an LPE by a procedure called *linearisation* [14, 20]. Our results could also easily be applied to other formalisms employing concurrency.

An LPE is a flat process description, consisting of a collection of summands that describe transitions symbolically. Each summand can perform an action and advance the system to some next state, given that a certain condition based

* This research has been partially funded by NWO under grant 612.063.817 (SYRUP).

on the current state is true. It has already been shown useful to reduce LPEs directly (e.g. [5, 15]), instead of first generating their entire (or partial) state spaces and reducing those, or performing reductions on-the-fly. The state space obtained from a reduced LPE is often much smaller than the equivalent state space obtained from an unreduced LPE; hence, both memory and time are saved.

The reductions we will introduce rely on the order in which summands can be executed. The problem when using LPEs, however, is that the explicit control flow of the original parallel processes has been lost, since they have been merged into one linear form. Moreover, some control flow could already have been encoded in the state parameters of the original specification. To solve this, we first present a technique to reconstruct the *control flow graphs* of an LPE. This technique is based on detecting which state parameters act as program counters for the underlying parallel processes; we call these *control flow parameters* (CFPs). We then reconstruct the control flow graph of each CFP based on the values it can take before and after each summand.

Using the reconstructed control flow, we define a parameter to be *relevant* if, before overwritten, it might be used by an enabling or action function, or by a next-state function to determine the value of another parameter that is relevant in the next state. Parameters that are not relevant are *irrelevant*, also called *dead*. Our syntactic reduction technique resets such irrelevant variables to their initial value. This is justified, because these variables will be overwritten before ever being read.

Finally, we describe several further insights about additional reductions, potential limitations, and potential adaptations to our theory.

Contributions. (1) We present a novel method to reconstruct the control flow of linear processes. Especially when specifications are translated between languages, their control flow may be hidden in the state parameters (as will also hold for our main case study). No such reconstruction method appeared in literature before.

(2) We use the reconstructed control flow to perform data flow analysis, resetting irrelevant state parameters. We prove that the transformed system is strongly bisimilar to the original, and that the state space never increases.

(3) We implemented our method in a tool called `stategraph` and provide several examples, showing that significant reductions can be obtained. The main case study clearly explains the use of control flow reconstruction. By finding useful variable resets automatically, the user can focus on modelling systems in an intuitive way, instead of formulating models such that the toolset can handle them best. This idea of automatic syntactic transformations for improving the efficiency of formal verification (not relying on users to make their models as efficient as possible) already proved to be a fruitful concept in earlier work [21].

Related work. Liveness analysis techniques are well-known in compiler theory [1]. However, their focus is often not on handling the multiple control flows arising from parallelism. Moreover, these techniques generally only work locally for each block of program code, and aim at reducing execution time instead of state space.

The concept of resetting dead variables for state space reduction was first formalised by Bozga et al. [7], but their analysis was based on a set of sequential processes with queues rather than parallel processes. Moreover, relevance of variables was only dealt with locally, such that a variable that is passed to a queue or written to another variable was considered relevant, even if it is never used afterwards. A similar technique was presented in [22], using analysis of control flow graphs. It suffers from the same locality restriction as [7]. Most recent is [11], which applies data flow analysis to value-passing process algebras. It uses Petri nets as its intermediate format, featuring concurrency and taking into account global liveness information. We improve on this work by providing a thorough formal foundation including bisimulation preservation proofs, and by showing that our transformation never increases the state space. Most importantly, none of the existing approaches attempts to reconstruct control flow information that is hidden in state variables, missing opportunities for reduction.

The μ CRL toolkit already contained a tool `parelm`, implementing a basic variant of our methods. Instead of resetting state parameters that are dead given some context, it simply removes parameters that are dead in all contexts [12]. That is, it marks all parameters that either occur in some condition or action argument, and also (recursively and iteratively) all parameters that are used in some summand to determine one of the marked parameters. Unmarked parameters are then deleted. As it does not take into account the control flow, parameters that are sometimes relevant and sometimes not will never be reset. We show by examples from the μ CRL toolset that `stategraph` indeed improves on `parelm`.

Organisation of the paper. After the preliminaries in Section 2, we discuss the reconstruction of control flow graphs in Section 3, the data flow analysis in Section 4, and the transformation in Section 5. The results of the case studies are given in Section 6, and conclusions and directions for future work in Section 7. The further insights are discussed in Appendix A.

2 Preliminaries

Notation. Variables for single values are written in lowercase, variables for sets or types in uppercase. We write variables for vectors and sets or types of vectors in boldface.

Labelled transition systems (LTSs). The semantics of an LPE is given in terms of an *LTS*: a tuple $\langle S, s_0, A, \Delta \rangle$, with S a set of states, $s_0 \in S$ the initial state, A a set of actions, and $\Delta \subseteq S \times A \times S$ a transition relation.

Linear process equations (LPEs). The LPE [4] is a common format for defining LTSs in a symbolic manner. It is a restricted process algebraic equation, similar to the Greibach normal form for formal grammars, specifications in the

language UNITY [8], and the precondition-effect style used for describing automata [17]. Usenko showed how to transform a general μ CRL specification into an LPE [14, 20].

Each LPE is of the form

$$X(\mathbf{d}: \mathbf{D}) = \sum_{i \in I} \sum_{\mathbf{e}_i: \mathbf{E}_i} c_i(\mathbf{d}, \mathbf{e}_i) \Rightarrow a_i(\mathbf{d}, \mathbf{e}_i) \cdot X(g_i(\mathbf{d}, \mathbf{e}_i)),$$

where \mathbf{D} is a type for *state vectors* (containing the global variables), I a set of *summand indices*, and \mathbf{E}_i a type for *local variables vectors* for summand i . The summations represent nondeterministic choices; the outer between different summands, the inner between different possibilities for the local variables. Furthermore, each summand i has an *enabling function* c_i , an *action function* a_i (yielding an atomic action, potentially with parameters), and a *next-state function* g_i , which may all depend on the state and the local variables. In this paper we assume the existence of an LPE with the above function and variable names, as well as an initial state vector *init*.

Given a vector of formal state parameters \mathbf{d} , we use d_j to refer to its j^{th} parameter. An actual state is a vector of values, denoted by \mathbf{v} ; we use v_j to refer to its j^{th} value. We use D_j to denote the type of d_j , and J for the set of all parameters d_j . Furthermore, $g_{i,j}(\mathbf{d}, \mathbf{e}_i)$ denotes the j^{th} element of $g_i(\mathbf{d}, \mathbf{e}_i)$, and $\text{pars}(t)$ the set of all parameters d_j that syntactically occur in the expression t .

The state space of the LTS underlying an LPE consists of all state vectors. It has a transition from \mathbf{v} to \mathbf{v}' by an atomic action $a(\mathbf{p})$ (parameterised by the possibly empty vector \mathbf{p}) if and only if there is a summand i for which a vector of local variables \mathbf{e}_i exists such that the enabling function is true, the action is $a(\mathbf{p})$ and the next-state function yields \mathbf{v}' . Formally, for all $\mathbf{v}, \mathbf{v}' \in \mathbf{D}$, there is a transition $\mathbf{v} \xrightarrow{a(\mathbf{p})} \mathbf{v}'$ if and only if there is a summand i such that

$$\exists \mathbf{e}_i \in \mathbf{E}_i \cdot c_i(\mathbf{v}, \mathbf{e}_i) \wedge a_i(\mathbf{v}, \mathbf{e}_i) = a(\mathbf{p}) \wedge g_i(\mathbf{v}, \mathbf{e}_i) = \mathbf{v}'.$$

Example 1. Consider a process consisting of two buffers, B_1 and B_2 . Buffer B_1 reads a datum of type D from the environment, and sends it synchronously to B_2 . Then, B_2 writes it back to the environment. The processes are given by

$$B_1 = \sum_{d: D} \text{read}(d) \cdot w(d) \cdot B_1, \quad B_2 = \sum_{d: D} r(d) \cdot \text{write}(d) \cdot B_2,$$

put in parallel and communicating on w and r . Linearised [20], they become

$$\begin{aligned} X(a: \{1, 2\}, b: \{1, 2\}, x: D, y: D) = & \\ & \sum_{d: D} \begin{array}{ll} a = 1 & \Rightarrow \text{read}(d) \cdot X(2, b, d, y) \quad (1) \\ + & b = 2 \quad \Rightarrow \text{write}(y) \cdot X(a, 1, x, y) \quad (2) \\ + & a = 2 \wedge b = 1 \Rightarrow c(x) \cdot X(1, 2, x, x) \quad (3) \end{array} \end{aligned}$$

where the first summand models behaviour of B_1 , the second models behaviour of B_2 , and the third models their communication. The global variables a and b are used as program counters for B_1 and B_2 , and x and y for their local memory.

Strong bisimulation. When transforming a specification S into S' , it is obviously important to verify that S and S' describe equivalent systems. For this we will use *strong bisimulation* [18], one of the most prominent notions of equivalence, which relates processes that have the same branching structure. It is well-known that strongly bisimilar processes satisfy the same properties, as for instance expressed in CTL* or μ -calculus. Formally, two processes with initial states p and q are strongly bisimilar if there exists a relation R such that $(p, q) \in R$, and

- if $(s, t) \in R$ and $s \xrightarrow{a} s'$, then there is a t' such that $t \xrightarrow{a} t'$ and $(s', t') \in R$;
- if $(s, t) \in R$ and $t \xrightarrow{a} t'$, then there is a s' such that $s \xrightarrow{a} s'$ and $(s', t') \in R$.

3 Reconstructing the Control Flow Graphs

First, we define a parameter to be *changed* in a summand i if its value after taking i might be different from its current value. A parameter is *directly used* in i if it occurs in its enabling function or action function, and *used* if it is either directly used or needed to calculate the next state.

Definition 1 (Changed, used). *Let i be a summand, then a parameter d_j is changed in i if $g_{i,j}(\mathbf{d}, \mathbf{e}_i) \neq d_j$, otherwise it is unchanged in i . It is directly used in i if $d_j \in \text{pars}(a_i(\mathbf{d}, \mathbf{e}_i)) \cup \text{pars}(c_i(\mathbf{d}, \mathbf{e}_i))$, and used in i if it is directly used in i or $d_j \in \text{pars}(g_{i,k}(\mathbf{d}, \mathbf{e}_i))$ for some k such that d_k is changed in i .*

We will often need to deduce the value s that a parameter d_j must have for a summand i to be taken; the *source* of d_j for i . More precisely, this value is defined such that the enabling function of i can only evaluate to true if $d_j = s$.

Definition 2 (Source). *A function $f: I \times (d_j : J) \rightarrow D_j \cup \{\perp\}$ is a source function if, for every $i \in I$, $d_j \in J$, and $s \in D_j$, $f(i, d_j) = s$ implies that*

$$\forall \mathbf{v} \in \mathbf{D}, \mathbf{e}_i \in \mathbf{E}_i \cdot c_i(\mathbf{v}, \mathbf{e}_i) \implies v_j = s.$$

Furthermore, $f(i, d_j) = \perp$ is always allowed; it indicates that no unique value s complying to the above could be found.

In the following we assume the existence of a source function `source`.

Note that `source(i, d_j)` is allowed to be \perp even though there might be some source s . The reason for this is that computing the source is in general undecidable, so in practice heuristics are used that sometimes yield \perp when in fact a source is present. However, we will see that this does not result in any errors. The same holds for the destination functions defined below.

Basically, the heuristics we apply to find a source can handle equations, disjunctions and conjunctions. For an equational condition $x = c$ the source is obviously c , for a disjunction of such terms we apply set union, and for conjunction intersection. If for some summand i a *set* of sources is obtained, it can be split into multiple summands, such that each again has a unique source.

Example 2. Let $c_i(\mathbf{d}, \mathbf{e}_i)$ be given by $(d_j = 3 \vee d_j = 5) \wedge d_j = 3 \wedge d_k = 10$, then obviously $\text{source}(i, d_j) = 3$ is valid (because $(\{3\} \cup \{5\}) \cap \{3\} = \{3\}$), but also (as always) $\text{source}(i, d_j) = \perp$.

We define the destination of a parameter d_j for a summand i to be the unique value d_j has after taking summand i . Again, we only specify a minimal requirement.

Definition 3 (Destination). *A function $f: I \times (d_j:J) \rightarrow D_j \cup \{\perp\}$ is a destination function if, for every $i \in I$, $d_j \in J$, and $s \in D_j$, $f(i, d_j) = s$ implies*

$$\forall \mathbf{v} \in \mathbf{D}, \mathbf{e}_i \in \mathbf{E}_i \cdot c_i(\mathbf{v}, \mathbf{e}_i) \implies g_{i,j}(\mathbf{v}, \mathbf{e}_i) = s.$$

Furthermore, $f(i, d_j) = \perp$ is always allowed, indicating that no unique destination value could be derived.

In the following we assume the existence of a destination function dest .

Our heuristics for computing $\text{dest}(i, d_j)$ just substitute $\text{source}(i, d_j)$ for d_j in the next-state function of summand i , and try to rewrite it to a closed term.

Example 3. Let $c_i(\mathbf{d}, \mathbf{e}_i)$ be given by $d_j = 8$ and $g_{i,j}(\mathbf{d}, \mathbf{e}_i)$ by $d_j + 5$, then $\text{dest}(i, d_j) = 13$ is valid, but also (as always) $\text{dest}(i, d_j) = \perp$. If for instance $c_i(\mathbf{d}, \mathbf{e}_i) = d_j = 5$ and $g_{i,j}(\mathbf{d}, \mathbf{e}_i) = e_3$, then $\text{dest}(i, d_j)$ can only yield \perp , since the value of d_j after taking i is not fixed.

We say that a parameter *rules* a summand if both its source and its destination for that summand can be computed.

Definition 4 (Rules). *A parameter d_j rules a summand i if $\text{source}(i, d_j) \neq \perp$ and $\text{dest}(i, d_j) \neq \perp$.*

The set of all summands that d_j rules is denoted by $R_{d_j} = \{i \in I \mid d_j \text{ rules } i\}$. Furthermore, V_{d_j} denotes the set of all possible values that d_j can take before and after taking one of the summands which it rules, plus its initial value. Formally,

$$V_{d_j} = \{ \text{source}(i, d_j) \mid i \in R_{d_j} \} \cup \{ \text{dest}(i, d_j) \mid i \in R_{d_j} \} \cup \{ \text{init}_j \}.$$

Examples will show that summands can be ruled by several parameters.

We now define a parameter to be a *control flow parameter* if it rules all summands in which it is changed. Stated differently, in every summand a *control flow parameter* is either left alone or we know what happens to it. Such a parameter can be seen as a *program counter* for the summands it rules, and therefore its values can be seen as *locations*. All other parameters are called *data parameters*.

Definition 5 (Control flow parameters). *A parameter d_j is a control flow parameter (CFP) if for all $i \in I$, either d_j rules i , or d_j is unchanged in i . A parameter that is not a CFP is called a data parameter (DP).*

The set of all summands $i \in I$ such that d_j rules i is called the cluster of d_j . The set of all CFPs is denoted by \mathcal{C} , the set of all DPs by \mathcal{D} .

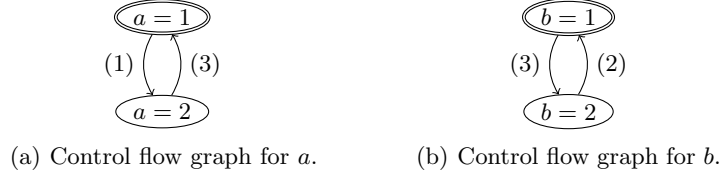


Fig. 1. Control flow graphs for the LPE of Example 1

Example 4. Consider the LPE of Example 1 again. For the first summand we may define $\text{source}(1, a) = 1$ and $\text{dest}(1, a) = 2$. Therefore, parameter a rules the first summand. Similarly, it rules the third summand. As a is unchanged in the second summand, it is a CFP (with summands 1 and 3 in its cluster). In the same way, we can show that parameter b is a CFP ruling summands 2 and 3. Parameter x is a DP, as it is changed in summand 1 while both its source and its destination are not unique. From summand 3 it follows that y is a DP.

Based on CFPs, we can define *control flow graphs*. The nodes of the control flow graph of a CFP d_j are the values d_j can take, and the edges denote possible transitions. Specifically, an edge labelled i from value s to t denotes that summand i might be taken if $d_j = s$, resulting in $d_j = t$.

Definition 6 (Control flow graphs). Let d_j be a CFP, then the control flow graph for d_j is the tuple (V_{d_j}, E_{d_j}) , where V_{d_j} was given in Definition 4, and

$$E_{d_j} = \{ (s, i, t) \mid i \in R_{d_j} \wedge s = \text{source}(i, d_j) \wedge t = \text{dest}(i, d_j) \}.$$

Figure 1 shows the control flow graphs for the LPE of Example 1.

The next proposition states that if a CFP d_j rules a summand i , and i is enabled for some state vector $\mathbf{v} = (v_1, \dots, v_j, \dots, v_n)$ and local variable vector \mathbf{e}_i , then the control flow graph of d_j contains an edge from v_j to $g_{i,j}(\mathbf{v}, \mathbf{e}_i)$.

Proposition 1. Let d_j be a CFP, \mathbf{v} a state vector, and \mathbf{e}_i a local variable vector. Then, if d_j rules i and $c_i(\mathbf{v}, \mathbf{e}_i)$ holds, it follows that $(v_j, i, g_{i,j}(\mathbf{v}, \mathbf{e}_i)) \in E_{d_j}$.

Proof. If d_j rules i , then by definition $\text{source}(i, d_j) \neq \perp$. By the definition of source, $c_i(\mathbf{v}, \mathbf{e}_i)$ implies that $v_j = \text{source}(i, d_j)$. Using the definition of rules again, $\text{dest}(i, d_j) \neq \perp$, and by the definition of dest we know that $c_i(\mathbf{v}, \mathbf{e}_i)$ implies that $\text{dest}(i, d_j) = g_{i,j}(\mathbf{v}, \mathbf{e}_i)$. Thus, using the definition of control flow graph, indeed $(v_j, i, g_{i,j}(\mathbf{v}, \mathbf{e}_i)) \in E_{d_j}$. \square

Note that we reconstruct a local control flow graph per CFP, rather than a global control flow graph. Although global control flow might be useful, its graph can grow larger than the complete state space, completely defeating its purpose.

4 Simultaneous Data Flow Analysis

Using the notion of CFPs, we analyse to which clusters DPs belong.

Definition 7 (The belongs-to relation). *Let d_k be a DP and d_j a CFP, then d_k belongs to d_j if all summands $i \in I$ that use or change d_k are ruled by d_j . We assume that each DP belongs to at least one CFP, and define CFPs to not belong to anything.*

Note that the assumption above can always be satisfied by adding a dummy parameter b of type Bool to every summand, initialising it to `true`, adding $b = \text{true}$ to every c_i , and leaving b unchanged in all g_i .

Also note that the fact that a DP d_k belongs to a CFP d_j implies that the complete data flow of d_k is contained in the summands of the cluster of d_j . Therefore, all decisions on resetting d_k can be made based on the summands within this cluster.

Example 5. For the LPE of the previous example, x belongs to a , and y to b .

If a DP d_k belongs to a CFP d_j , it follows that all analyses on d_k can be made by the cluster of d_j . We begin these analyses by defining for which values of d_j (so during which part of the cluster’s control flow) the value of d_k is relevant.

Basically, d_k is *relevant* if it might be directly used before it will be changed, otherwise it is *irrelevant*. More precisely, the relevance of d_k is divided into three conditions. They state that d_k is relevant given that $d_j = s$, if there is a summand i that can be taken when $d_j = s$, such that either (1) d_k is directly used in i ; or (2,3) d_k is indirectly used in i to determine the value of a DP that is relevant after taking i . Basically, clause (2) deals with temporal dependencies within one cluster, whereas (3) deals with dependencies through concurrency between different clusters. The next definition formalises this.

Definition 8 (Relevance). *Let $d_k \in \mathcal{D}$ and $d_j \in \mathcal{C}$, such that d_k belongs to d_j . Given some $s \in D_j$, we use $(d_k, d_j, s) \in R$ (or $R(d_k, d_j, s)$) to denote that the value of d_k is relevant when $d_j = s$. Formally, R is the smallest relation such that*

1. *If d_k is directly used in some $i \in I$, d_k belongs to some $d_j \in \mathcal{C}$, and $s = \text{source}(i, d_j)$, then $R(d_k, d_j, s)$;*
2. *If $R(d_l, d_j, t)$, and there exists an $i \in I$ such that $(s, i, t) \in E_{d_j}$, and d_k belongs to d_j , and $d_k \in \text{pars}(g_{i,l}(\mathbf{d}, \mathbf{e}_i))$, then $R(d_k, d_j, s)$;*
3. *If $R(d_l, d_p, t)$, and there exists an $i \in I$ and an r such that $(r, i, t) \in E_{d_p}$, and $d_k \in \text{pars}(g_{i,l}(\mathbf{d}, \mathbf{e}_i))$, and d_k belongs to some cluster d_j to which d_l does not belong, and $s = \text{source}(i, d_j)$, then $R(d_k, d_j, s)$.*

If $(d_k, d_j, s) \notin R$, we write $\neg R(d_k, d_j, s)$ and say that d_k is irrelevant when $d_j = s$.

Although it might seem that the second and third clause could be merged, we provide an example in Appendix A.7 where this would decrease the number of reductions.

Example 6. Applying the first clause of the definition of relevance to the LPE of Example 1, we see that $R(x, a, 2)$ and $R(y, b, 2)$. Then, no clauses apply anymore, so $\neg R(x, a, 1)$ and $\neg R(y, b, 1)$. Now, we hide the action c , obtaining

$$\begin{aligned} X(a: \{1, 2\}, b: \{1, 2\}, x: D, y: D) = \\ \sum_{d: D} a = 1 &\Rightarrow \text{read}(d) \cdot X(2, b, d, y) & (1) \\ + \quad b = 2 &\Rightarrow \text{write}(y) \cdot X(a, 1, x, y) & (2) \\ + \quad a = 2 \wedge b = 1 &\Rightarrow \tau \cdot X(1, 2, x, x) & (3) \end{aligned}$$

In this case, the first clause of relevance only yields $R(y, b, 2)$. Moreover, since x is used in summand 3 to determine the value that y will have when b becomes 2, also $R(x, a, 2)$. Formally, this can be found using the third clause, substituting $l = y$, $p = b$, $t = 2$, $i = 3$, $r = 1$, $k = x$, $j = a$, and $s = 2$.

Since clusters have only limited information, they do not always detect a DP's irrelevance. However, they always have sufficient information to never erroneously find a DP irrelevant. Therefore, we define a DP d_k to be relevant given a state vector \mathbf{v} , if it is relevant for the valuations of *all* CFPs d_j it belongs to.

Definition 9 (Relevance in state vectors). *The relevance of a parameter d_k given a state vector \mathbf{v} , denoted $\text{Relevant}(d_k, \mathbf{v})$, is defined by*

$$\text{Relevant}(d_k, \mathbf{v}) = \bigwedge_{\substack{d_j \in \mathcal{C} \\ d_k \text{ belongs to } d_j}} R(d_k, d_j, v_j).$$

Note that, since a CFP belongs to no other parameters, it is always relevant.

Example 7. For the LPE of the previous example we derived that x belongs to a , and that it is irrelevant when $a = 1$. Therefore, the valuation $x = d_5$ is not relevant in the state vector $\mathbf{v} = (1, 2, d_5, d_2)$, so we write $\neg \text{Relevant}(x, \mathbf{v})$.

Obviously, the value of a DP that is irrelevant in a state vector does not matter. For instance, the two state vectors $\mathbf{v} = (w, x, y)$ and $\mathbf{v}' = (w, x', y)$ are equivalent if $\neg \text{Relevant}(d_2, \mathbf{v})$. To formalise this, we introduce a relation $\tilde{=}$ on state vectors, given by

$$\mathbf{v} \tilde{=} \mathbf{v}' \iff \forall d_k \in J: (\text{Relevant}(d_k, \mathbf{v}) \implies v_k = v'_k),$$

and prove that it is a strong bisimulation; one of the main results of this paper.

First, we show that $\tilde{=}$ is an equivalence relation.

Lemma 1. *Let \mathbf{v} and \mathbf{v}' be state vectors such that $\mathbf{v} \tilde{=} \mathbf{v}'$, and $\text{Relevant}(d_k, \mathbf{v}')$ for some d_k . Then it follows that $\text{Relevant}(d_k, \mathbf{v})$.*

Proof. If d_k is a CFP, then $\text{Relevant}(d_k, \mathbf{v})$ by definition. From now on we therefore assume that it is a DP.

Assume that $\mathbf{v} \tilde{=} \mathbf{v}'$ and $\text{Relevant}(d_k, \mathbf{v}')$. Let d_j be one of the CFPs d_k belongs to. Then, by definition of Relevant , we have $R(d_k, d_j, v'_j)$. Because d_j is

a CFP we know that $Relevant(d_j, \mathbf{v})$, so by the definition of \cong we have $v_j = v'_j$. Since $R(d_k, d_j, v'_j)$, this immediately implies $R(d_k, d_j, v_j)$. Since this argument holds for all d_j that d_k belongs to, we obtain $Relevant(d_k, \mathbf{v})$. \square

Lemma 2. *The relation \cong is an equivalence relation.*

Proof. Reflexivity is trivial. For symmetry, assume that $\mathbf{v} \cong \mathbf{v}'$. For all d_k , if $Relevant(d_k, \mathbf{v}')$, then by Lemma 1 also $Relevant(d_k, \mathbf{v})$. Therefore, by definition of \cong and the assumption that $\mathbf{v} \cong \mathbf{v}'$, we obtain $v_k = v'_k$, hence $\mathbf{v}' \cong \mathbf{v}$.

For transitivity, assume that $\mathbf{v} \cong \mathbf{v}'$ and $\mathbf{v}' \cong \mathbf{v}''$. If $Relevant(d_k, \mathbf{v})$, then by definition $v_k = v'_k$. Using symmetry and Lemma 1 it follows that $Relevant(d_k, \mathbf{v}')$, and hence $v'_k = v''_k$. Therefore, $\mathbf{v} \cong \mathbf{v}''$. \square

Now, we show that if a summand i is enabled given some state vector \mathbf{v} , then it is also enabled given a state vector \mathbf{v}' such that $\mathbf{v} \cong \mathbf{v}'$.

Lemma 3. *Let \mathbf{v} and \mathbf{v}' be state vectors such that $\mathbf{v} \cong \mathbf{v}'$. Let $i \in I$ be a summand and \mathbf{e}_i a local variable vector for i . Then, $c_i(\mathbf{v}, \mathbf{e}_i)$ implies $c_i(\mathbf{v}', \mathbf{e}_i)$.*

Proof. It has to be shown that for all $d_k \in \text{pars}(c_i(\mathbf{d}, \mathbf{e}_i))$ it holds that $v_k = v'_k$. Since this is trivially true for CFPs, we from now on assume that d_k is a DP. Assume that $c_i(\mathbf{v}, \mathbf{e}_i)$ holds. Let an arbitrary $d_k \in \text{pars}(c_i(\mathbf{d}, \mathbf{e}_i))$ be given, and let d_j be a CFP that d_k belongs to. Then, since d_k is directly used in i , by definition of belongs-to d_j rules i . Therefore, by Proposition 1 and Definition 6, $v_j = \text{source}(i, d_j)$, and because d_k is used directly in i by definition $R(d_k, d_j, v_j)$. Since this holds for all d_j to which d_k belongs, we obtain $Relevant(d_k, \mathbf{v})$, and, using the definition of \cong , $v_k = v'_k$. Since d_k was chosen arbitrary, this holds for all $d_k \in \text{pars}(c_i(\mathbf{d}, \mathbf{e}_i))$, so $c_i(\mathbf{v}', \mathbf{e}_i)$ also holds. \square

We can also show that if a summand i is taken given some state vector \mathbf{v} , the resulting action is identical to when i is taken given a state vector \mathbf{v}' such that $\mathbf{v} \cong \mathbf{v}'$.

Lemma 4. *Let \mathbf{v} and \mathbf{v}' be state vectors such that $\mathbf{v} \cong \mathbf{v}'$. Let $i \in I$ be a summand and \mathbf{e}_i a local variable vector for i . Then, $a_i(\mathbf{v}, \mathbf{e}_i) = a$ implies that $a_i(\mathbf{v}', \mathbf{e}_i) = a$.*

Proof. Identical to the proof of Lemma 3, when substituting c_i by a_i . \square

Finally, we show that taking a summand i given some state vector \mathbf{v} and taking it given a state vector \mathbf{v}' such that $\mathbf{v} \cong \mathbf{v}'$ yield next-state vectors that are equivalent with respect to \cong .

Lemma 5. *Let \mathbf{v} and \mathbf{v}' be state vectors such that $\mathbf{v} \cong \mathbf{v}'$. Let $i \in I$ be a summand and \mathbf{e}_i a local variable vector for i . Then $c_i(\mathbf{v}, \mathbf{e}_i)$ implies that $g_i(\mathbf{v}, \mathbf{e}_i) \cong g_i(\mathbf{v}', \mathbf{e}_i)$.*

Proof. By definition of \cong , it has to be shown that for all parameters d_k such that $\text{Relevant}(d_k, g_i(\mathbf{v}, \mathbf{e}_i))$, it holds that $g_{i,k}(\mathbf{v}, \mathbf{e}_i) = g_{i,k}(\mathbf{v}', \mathbf{e}_i)$. For this we have to show that $v_m = v'_m$ for all parameters $d_m \in \text{pars}(g_{i,k}(\mathbf{d}, \mathbf{e}_i))$. Since CFPs are always relevant they cannot differ between \mathbf{v} and \mathbf{v}' , therefore we assume that d_m is a DP from now on. Furthermore, since the next-state function of a CFP can only depend on CFPs, we can also assume that d_k is a DP.

Let d_k and d_m be such that $\text{Relevant}(d_k, g_i(\mathbf{v}, \mathbf{e}_i))$ and $d_m \in \text{pars}(g_{i,k}(\mathbf{d}, \mathbf{e}_i))$. Furthermore, let d_l be a CFP that d_m belongs to. Now, we distinguish between whether d_k belongs to d_l or not.

- Suppose that d_k belongs to d_l . Because $\text{Relevant}(d_k, g_i(\mathbf{v}, \mathbf{e}_i))$ and d_k belongs to d_l , by definition of Relevant it holds that $R(d_k, d_l, g_{i,l}(\mathbf{v}, \mathbf{e}_i))$. Now, if d_l rules d_i , then by Proposition 1 (and the initial assumption that $c_i(\mathbf{v}, \mathbf{e}_i)$ holds), we have $(v_l, i, g_{i,l}(\mathbf{v}, \mathbf{e}_i)) \in E_{d_l}$. Now it immediately follows from the second clause of the definition of relevance that $R(d_m, d_l, v_l)$.
On the other hand, if d_l does not rule i , then by definition of belongs-to d_k is unchanged in i , so $g_{i,k}(\mathbf{d}, \mathbf{e}_i) = d_k$. This implies that $d_m = d_k$. Since d_l does not rule i , but it is a CFP, it follows that $g_{i,l}(\mathbf{d}, \mathbf{e}_i) = d_l$. So, $R(d_k, d_l, g_{i,l}(\mathbf{v}, \mathbf{e}_i)) = R(d_m, d_l, v_l)$, and therefore $R(d_m, d_l, v_l)$ follows trivially.
- Suppose that d_k does not belong to d_l . Since d_m does belong to d_l , $d_m \neq d_k$. So, i uses d_m (because d_m occurs in $\text{pars}_{i,k}(\mathbf{d}, \mathbf{e}_i)$ and $d_m \neq d_k$), hence d_l rules i . Using Proposition 1, we obtain $(v_l, i, g_{i,l}(\mathbf{v}, \mathbf{e}_i)) \in E_{d_l}$, which implies that $v_l = \text{source}(i, d_l)$. Next, let d_p be some CFP that d_k belongs to. Then, by definition of Relevant and the fact that $\text{Relevant}(d_k, g_i(\mathbf{v}, \mathbf{e}_i))$, we know that $R(d_k, d_p, g_{i,p}(\mathbf{v}, \mathbf{e}_i))$. Because $d_m \neq d_k$ and $d_m \in \text{pars}(g_{i,k}(\mathbf{d}, \mathbf{e}_i))$ we know that d_k is changed by i , and because d_k belongs to d_p it follows that d_p rules i . Applying Proposition 1 again, $(v_p, i, g_{i,p}(\mathbf{v}, \mathbf{e}_i)) \in E_{d_p}$. Then, by the third clause of the definition of relevance, $R(d_m, d_l, v_l)$.

Since in all cases we have shown that $R(d_m, d_l, v_l)$ for an arbitrary d_l that d_m belongs to, by definition $\text{Relevant}(d_m, \mathbf{v})$. Therefore, since $\mathbf{v} \cong \mathbf{v}'$, by definition of \cong it follows that $v_m = v'_m$. \square

Using the lemmas above, the following theorem easily follows.

Theorem 1. *The relation \cong is a strong bisimulation.*

Proof. Let \mathbf{v}_0 and \mathbf{v}'_0 be state vectors such that $\mathbf{v}_0 \cong \mathbf{v}'_0$. Furthermore, assume that $\mathbf{v}_0 \xrightarrow{a} \mathbf{v}_1$. Because \cong is symmetric (Lemma 2), we only need to prove that there exists a transition $\mathbf{v}'_0 \xrightarrow{a} \mathbf{v}'_1$ such that $\mathbf{v}_1 \cong \mathbf{v}'_1$.

By the operational semantics there is a summand i and a local variable vector \mathbf{e}_i such that $c_i(\mathbf{v}_0, \mathbf{e}_i)$ holds, and $a = a_i(\mathbf{v}_0, \mathbf{e}_i)$, and $\mathbf{v}_1 = g_i(\mathbf{v}_0, \mathbf{e}_i)$. Now, by Lemma 3 we know that $c_i(\mathbf{v}'_0, \mathbf{e}_i)$ holds, and by Lemma 4 that $a = a_i(\mathbf{v}'_0, \mathbf{e}_i)$ holds. Therefore, $\mathbf{v}'_0 \xrightarrow{a} g_i(\mathbf{v}'_0, \mathbf{e}_i)$. By Lemma 5, $g_i(\mathbf{v}_0, \mathbf{e}_i) \cong g_i(\mathbf{v}'_0, \mathbf{e}_i)$, proving the theorem. \square

5 Transformations on LPEs

The most important application of the data flow analysis described in the previous section is to reduce the number of reachable states of the LTS underlying an LPE. Note that by modifying irrelevant parameters in an arbitrary way, this number could even increase. We present a syntactic transformation of LPEs, and prove that it yields a strongly bisimilar system and can never increase the number of reachable states. In several practical examples, it yields a decrease.

Our transformation uses the idea that a data parameter d_k that is irrelevant in all possible states after taking a summand i , can just as well be reset by i to its initial value.

Definition 10 (Transforms). *Given an LPE X of the familiar form*

$$X(\mathbf{d}: \mathbf{D}) = \sum_{i \in I} \sum_{\mathbf{e}_i: \mathbf{E}_i} c_i(\mathbf{d}, \mathbf{e}_i) \Rightarrow a_i(\mathbf{d}, \mathbf{e}_i) \cdot X(g_i(\mathbf{d}, \mathbf{e}_i)),$$

we define its transform to be the LPE X' given by

$$X'(\mathbf{d}: \mathbf{D}) = \sum_{i \in I} \sum_{\mathbf{e}_i: \mathbf{E}_i} c_i(\mathbf{d}, \mathbf{e}_i) \Rightarrow a_i(\mathbf{d}, \mathbf{e}_i) \cdot X'(g'_i(\mathbf{d}, \mathbf{e}_i)),$$

with

$$g'_{i,k}(\mathbf{d}, \mathbf{e}_i) = \begin{cases} g_{i,k}(\mathbf{d}, \mathbf{e}_i) & \text{if } \bigwedge_{\substack{d_j \in \mathcal{C} \\ d_j \text{ rules } i \\ d_k \text{ belongs to } d_j}} R(d_k, d_j, \text{dest}(i, d_j)), \\ \text{init}_k & \text{otherwise.} \end{cases}$$

We will use the notation $X(\mathbf{v})$ to denote state \mathbf{v} in the underlying LTS of X , and $X'(\mathbf{v})$ to denote state \mathbf{v} in the underlying LTS of X' .

Note that $g'_i(\mathbf{d}, \mathbf{e}_i)$ only deviates from $g_i(\mathbf{d}, \mathbf{e}_i)$ for parameters d_k that are irrelevant after taking i , as stated by the following lemma.

Lemma 6. *For every $i \in I$, state vector \mathbf{v} , and local variable vector \mathbf{e}_i , given that $c_i(\mathbf{v}, \mathbf{e}_i) = \text{true}$ it holds that $g_i(\mathbf{v}, \mathbf{e}_i) \simeq g'_i(\mathbf{v}, \mathbf{e}_i)$.*

Proof. To show that $g_i(\mathbf{v}, \mathbf{e}_i) \simeq g'_i(\mathbf{v}, \mathbf{e}_i)$, we need to show that for all parameters d_k such that $\text{Relevant}(d_k, g_i(\mathbf{v}, \mathbf{e}_i))$ we have $g_{i,k}(\mathbf{v}, \mathbf{e}_i) = g'_{i,k}(\mathbf{v}, \mathbf{e}_i)$. Assume such a $d_k \in J$. Then, by definition of *Relevant* we have

$$\bigwedge_{\substack{d_j \in \mathcal{C} \\ d_k \text{ belongs to } d_j}} R(d_k, d_j, g_{i,j}(\mathbf{v}, \mathbf{e}_i)) \quad \text{so also} \quad \bigwedge_{\substack{d_j \in \mathcal{C} \\ d_j \text{ rules } i \\ d_k \text{ belongs to } d_j}} R(d_k, d_j, g_{i,j}(\mathbf{v}, \mathbf{e}_i)).$$

By definition of g' and the fact that $\text{dest}(i, d_j) = g_{i,j}(\mathbf{v}, \mathbf{e}_i)$ when d_j rules i and $c_i(\mathbf{v}, \mathbf{e}_i)$ holds, we obtain $g'_{i,k}(\mathbf{v}, \mathbf{e}_i) = g_{i,k}(\mathbf{v}, \mathbf{e}_i)$. \square

Using this lemma we show that $X(\mathbf{v})$ and $X'(\mathbf{v})$ are bisimilar, by first proving an even stronger statement.

Theorem 2. *Let \cong be defined by*

$$X(\mathbf{v}) \cong X'(\mathbf{v}') \iff \mathbf{v} \simeq \mathbf{v}',$$

then \cong is a strong bisimulation. The relation \simeq is used as it was defined for X .

Proof. Let \mathbf{v}_0 and \mathbf{v}'_0 be state vectors such that $X(\mathbf{v}_0) \cong X'(\mathbf{v}'_0)$, so $\mathbf{v}_0 \simeq \mathbf{v}'_0$.

Assume that $X(\mathbf{v}_0) \xrightarrow{a} X(\mathbf{v}_1)$. We need to prove that there exists a transition $X'(\mathbf{v}'_0) \xrightarrow{a} X'(\mathbf{v}'_1)$ such that $X(\mathbf{v}_1) \cong X'(\mathbf{v}'_1)$. By Theorem 1 there exists a state vector \mathbf{v}''_1 such that $X(\mathbf{v}'_0) \xrightarrow{a} X(\mathbf{v}''_1)$ and $\mathbf{v}_1 \simeq \mathbf{v}''_1$. By the operational semantics, for some i and \mathbf{e}_i we thus have $c_i(\mathbf{v}'_0, \mathbf{e}_i)$, $a_i(\mathbf{v}'_0, \mathbf{e}_i) = a$, and $g_i(\mathbf{v}'_0, \mathbf{e}_i) = \mathbf{v}''_1$. By Definition 10, we have $X'(\mathbf{v}'_0) \xrightarrow{a} X'(g'_i(\mathbf{v}'_0, \mathbf{e}_i))$, and by Lemma 6 $g_i(\mathbf{v}'_0, \mathbf{e}_i) \simeq g'_i(\mathbf{v}'_0, \mathbf{e}_i)$. Now, by transitivity and reflexivity of \simeq (Lemma 2), $\mathbf{v}_1 \simeq \mathbf{v}''_1 = g_i(\mathbf{v}'_0, \mathbf{e}_i) \simeq g'_i(\mathbf{v}'_0, \mathbf{e}_i)$, hence $X(\mathbf{v}_1) \cong X'(g'_i(\mathbf{v}'_0, \mathbf{e}_i))$. By symmetry of \simeq , this completes the proof. \square

The following corollary, stating the desired bisimilarity, immediately follows.

Corollary 1. *Let X be an LPE, X' its transform, and \mathbf{v} a state vector. Then, $X(\mathbf{v})$ is strongly bisimilar to $X'(\mathbf{v})$.*

We now show that our choice of $g'(\mathbf{d}, \mathbf{e}_i)$ ensures that the state space of X' is at most as large as the state space of X . We first prove the invariant that if a parameter is irrelevant for a state vector, it is equal to its initial value.

Proposition 2. *For the process $X'(\mathbf{init})$ invariably $\neg \text{Relevant}(d_k, \mathbf{v})$ implies that $v_k = \text{init}_k$.*

Proof. For the initial state the invariant is trivially true. Now assume that the invariant holds for an arbitrary d_k for some reachable state vector \mathbf{v} . We will prove by induction that it still holds for all states reachable by an arbitrary summand i given a local state vector \mathbf{e}_i such that $c_i(\mathbf{v}, \mathbf{e}_i)$ holds. If

$$\bigwedge_{\substack{d_j \in \mathcal{C} \\ d_j \text{ rules } i \\ d_k \text{ belongs to } d_j}} R(d_k, d_j, \text{dest}(i, d_j)) \quad (1)$$

is false, then by definition we have $g'_{i,k} = \text{init}_k$ and the invariant holds. From now on we therefore assume that this conjunction is true, so by definition $g'_{i,k}(\mathbf{v}, \mathbf{e}_i) = g_{i,k}(\mathbf{v}, \mathbf{e}_i)$. We make a case distinction between $\text{Relevant}(d_k, \mathbf{v})$ and $\neg \text{Relevant}(d_k, \mathbf{v})$.

- Assume that $Relevant(d_k, \mathbf{v})$, so by definition

$$\bigwedge_{\substack{d_j \in \mathcal{C} \\ d_k \text{ belongs to } d_j}} R(d_k, d_j, v_j)$$

is true. Let $d_j \in \mathcal{C}$ such that d_k belongs to d_j and d_j does not rule i , then by definition d_j is unchanged in i , so from $R(d_k, d_j, v_j)$ it immediately follows that $R(d_k, d_j, g_{i,j}(\mathbf{v}, \mathbf{e}_i))$. Now we easily obtain

$$\bigwedge_{\substack{d_j \in \mathcal{C} \\ d_k \text{ belongs to } d_j}} R(d_k, d_j, g_{i,j}(\mathbf{v}, \mathbf{e}_i)),$$

since we already assumed $R(d_k, d_j, \text{dest}(i, d_j))$ for all $d_j \in \mathcal{C}$ such that d_k belongs to d_j and d_j does rule i (and for those d_j by definition $\text{dest}(i, d_j) = g_{i,j}(\mathbf{v}, \mathbf{e}_i)$). Since CFPs do not belong to other CFPs, $g'_{i,j}(\mathbf{v}, \mathbf{e}_i) = g_{i,j}(\mathbf{v}, \mathbf{e}_i)$ for all $d_j \in \mathcal{C}$ and therefore

$$\bigwedge_{\substack{d_j \in \mathcal{C} \\ d_k \text{ belongs to } d_j}} R(d_k, d_j, g'_{i,j}(\mathbf{v}, \mathbf{e}_i)).$$

Now, by definition $Relevant(d_k, g'_i(\mathbf{v}, \mathbf{e}_i))$, so the invariant holds.

- Assume that $\neg Relevant(d_k, \mathbf{v})$. If there exists a d_j such that d_k belongs to d_j and d_j does not rule i , then by definition of belongs-to d_k is unchanged in i . By the induction hypothesis $v_k = \text{init}_k$, so $g'_{i,k}(\mathbf{v}, \mathbf{e}_i) = g_{i,k}(\mathbf{v}, \mathbf{e}_i) = v_k = \text{init}_k$, so the invariant holds.

From now on assume that all d_j that d_k belongs to rule i . Therefore, by the assumed truth of Equation (1) and the fact that $\text{dest}(i, d_j) = g_{i,j}(\mathbf{v}, \mathbf{e}_i)$ for all d_j that rule i , it follows that

$$\bigwedge_{\substack{d_j \in \mathcal{C} \\ d_k \text{ belongs to } d_j}} R(d_k, d_j, g_{i,j}(\mathbf{v}, \mathbf{e}_i)),$$

and, since $g'_{i,j}(\mathbf{v}, \mathbf{e}_i) = g_{i,j}(\mathbf{v}, \mathbf{e}_i)$ for all $d_j \in \mathcal{C}$, we obtain

$$\bigwedge_{\substack{d_j \in \mathcal{C} \\ d_k \text{ belongs to } d_j}} R(d_k, d_j, g'_{i,j}(\mathbf{v}, \mathbf{e}_i)),$$

hence by definition $Relevant(d_k, g'_i(\mathbf{v}, \mathbf{e}_i))$ and the invariant holds. \square

Using this invariant we can now prove the following lemma, providing a functional strong bisimulation relating the states of $X(\mathbf{init})$ and $X'(\mathbf{init})$.

Lemma 7. *Let h be a function over state vectors, given for any \mathbf{v} by*

$$h_k(\mathbf{v}) = \begin{cases} v_k & \text{if } Relevant(d_k, \mathbf{v}), \\ \text{init}_k & \text{otherwise,} \end{cases}$$

then h is a strong bisimulation relating the states of $X(\mathbf{init})$ and $X'(\mathbf{init})$.

Proof. Let \mathbf{v}_0 and \mathbf{v}'_0 be state vectors such that $h(\mathbf{v}_0) = \mathbf{v}'_0$. Furthermore, assume that $X(\mathbf{v}_0) \xrightarrow{a} X(\mathbf{v}_1)$. We show that there exists a transition $X'(\mathbf{v}'_0) \xrightarrow{a} X'(\mathbf{v}'_1)$ such that $h(\mathbf{v}_1) = \mathbf{v}'_1$ (the proof of the opposite direction is completely symmetric).

By definition of h it follows that $\mathbf{v}_0 \approx \mathbf{v}'_0$, so by Lemma 6 and Theorem 2 there is a \mathbf{v}''_1 such that $X'(\mathbf{v}'_0) \xrightarrow{a} X'(\mathbf{v}''_1)$ and $\mathbf{v}_1 \approx \mathbf{v}''_1$. Assuming that for an arbitrary d_k it holds that $\text{Relevant}(d_k, \mathbf{v}_1)$, this implies that $v_{1_k} = v''_{1_k}$, so by definition of h we obtain $h_k(\mathbf{v}_1) = v_{1_k} = v''_{1_k}$. Assuming that $\neg \text{Relevant}(d_k, \mathbf{v}_1)$, by Lemma 1 and symmetry we have $\neg \text{Relevant}(d_k, \mathbf{v}''_1)$, so by Proposition 2 it follows that $v''_{1_k} = \text{init}_k$, so by definition of h we obtain $h_k(\mathbf{v}_1) = \text{init}_k = v''_{1_k}$.

In conclusion, for all d_k in all cases we have $h_k(\mathbf{v}_1) = v''_{1_k}$, so $h(\mathbf{v}_1) = \mathbf{v}''_1$. \square

Since the bisimulation relation between the states of X and X' is a function, and the domain of every function is at least as large as its image, the following corollary is immediate.

Corollary 2. *The number of reachable states in X' is at most as large as the number of reachable states in X .*

Example 8. Using the above transformation, the LPE of Example 6 becomes

$$\begin{aligned} X'(a: \{1, 2\}, b: \{1, 2\}, x: D, y: D) = \\ \sum_{d: D} \quad a = 1 & \quad \Rightarrow \text{read}(d) \cdot X'(2, b, d, y) & (1) \\ + \quad \quad b = 2 & \quad \Rightarrow \text{write}(y) \cdot X'(a, 1, x, d_1) & (2) \\ + \quad \quad a = 2 \wedge b = 1 & \Rightarrow \tau \cdot X'(1, 2, d_1, x) & (3) \end{aligned}$$

assuming that the initial state vector is $(1, 1, d_1, d_1)$. Note that for X' the state $(1, 1, d_i, d_j)$ is only reachable for $d_i = d_j = d_1$, whereas in the original specification X it is reachable for all $d_i, d_j \in D$ such that $d_i = d_j$.

6 Case Studies

The proposed method has been implemented in the context of the μCRL toolkit by a tool called `stategraph`. For evaluation purposes we applied it first on a model of a *handshake register*, modelled and verified by Hesselink [16]. We used a MacBook with a 2.4 GHz Intel Core 2 Duo processor and 2 GB memory.

A handshake register is a data structure that is used for communication between a single reader and a single writer. It guarantees *recentness* and *sequentiality*; any value that is read was at some point during the read action the last value written, and the values of sequential reads occur in the same order as they were written). Also, it is *waitfree*; both the reader and the writer can complete their actions in a bounded number of steps, independent of the other process. Hesselink provides a method to construct a handshake register of a certain data type based on four so-called safe registers and four atomic boolean registers.

We used a μCRL model of the handshake register, and one of the implementations using four safe registers. We generated their state spaces, minimised with

	constelm states	parelm time (expl.)	constelm time (sybm.)	constelm states	stategraph time (expl.)	constelm time (sybm.)
$ D = 2$	540,736	0:23.0	0:04.5	45,504	0:02.4	0:01.3
$ D = 3$	13,834,800	10:10.3	0:06.7	290,736	0:12.7	0:01.4
$ D = 4$	142,081,536	–	0:09.0	1,107,456	0:48.9	0:01.6
$ D = 5$	883,738,000	–	0:11.9	3,162,000	2:20.3	0:01.8
$ D = 6$	3,991,840,704	–	0:15.4	7,504,704	5:26.1	0:01.9

Table 1. Modelling a handshake register; **parelm** versus **stategraph**.

respect to τ^*a equivalence [9] and indeed obtained identical LTSs, showing that the implementation is correct. However, using a data type D of three values the state space before minimisation is already very large, such that its generation is quite time-consuming. So, we applied **stategraph** (in combination with the existing μ CRL tool **constelm** [12]) to reduce the LPE for different sizes of D . For comparison we also reduced the specifications in the same way using the existing, less powerful tool **parelm**.

For each specification we measured the time for reducing its LPE and generating the state space. We also used a recently implemented tool¹ for symbolic reachability analysis [6] to obtain the state spaces when not using **stategraph**, since in that case not all specifications could be generated explicitly. Every experiment was performed ten times, and the average run times are shown in Table 1 (where $x:y.z$ means x minutes and $y.z$ seconds).

Observations. The results show that **stategraph** provides a substantial reduction of the state space. Using **parelm** explicit generation was infeasible with just four data elements (after sixteen hours about half of the states had been generated), whereas using **stategraph** we could easily continue until six elements. Note that the state space reduction for $|D| = 6$ was more than a factor 500. Also observe that **stategraph** is impressively useful for speeding up symbolic analysis, as the time for symbolic generation improves an order of magnitude.

To gain an understanding of why our method works for this example, observe the μ CRL specification of the four safe registers below.

$$\begin{aligned}
Y(i: \text{Bool}, j: \text{Bool}, r: \{1, 2, 3\}, w: \{1, 2, 3\}, v: D, vw: D, vr: D) = & \\
& r = 1 \quad \Rightarrow \text{beginRead}(i, j) \cdot Y(i, j, 2, w, v, vw, vr) \quad (1) \\
+ & r = 2 \wedge w = 1 \Rightarrow \tau \cdot Y(i, j, 3, w, v, vw, v) \quad (2) \\
+ \sum_{x: D} & r = 2 \wedge w \neq 1 \Rightarrow \tau \cdot Y(i, j, 3, w, v, vw, x) \quad (3) \\
+ & r = 3 \quad \Rightarrow \text{endRead}(i, j, vr) \cdot Y(i, j, 1, w, v, vw, vr) \quad (4) \\
+ \sum_{x: D} & w = 1 \quad \Rightarrow \text{beginWrite}(i, j, x) \cdot Y(i, j, r, 2, v, x, vr) \quad (5) \\
+ & w = 2 \quad \Rightarrow \tau \cdot Y(i, j, r, 3, vw, vw, vr) \quad (6) \\
+ & w = 3 \quad \Rightarrow \text{endWrite}(i, j) \cdot Y(i, j, r, 1, vw, vw, vr) \quad (7)
\end{aligned}$$

¹ Available from <http://fmt.cs.utwente.nl/tools/ltsmin>.

The boolean parameters i and j are just meant to distinguish the four components. The parameter r denotes the read status, and w the write status.

Reading consists of a beginRead action, a τ step, and an endRead action. During the τ step either the contents of v is copied into vr (summand 2), or, when writing is taking place at the same time, a random value is copied to vr (summand 3). In the final step, an endRead action is produced with the value of vr as a parameter (summand 4). Writing works by first storing the value to be written in vw (summand 5), and then copying vw to v (summand 6).

The tool discovered that after summand 4 the value of vr is irrelevant, since it will not be used before summand 4 is reached again. This is always preceded by summand 2 or 3, both overwriting vr . Thus, vr can be reset to its initial value in the next-state function of summand 4. This turned out to drastically decrease the size of the state space. Other tools were not able to make this reduction, since it requires control flow reconstruction. Note that using parallel processes for the reader and the writer instead of our solution of encoding control flow in the data parameters would be difficult, because of the shared variable v .

Although the example may seem artificial, it is an almost one-to-one formalisation of its description in [16]. Without our method for control flow reconstruction, finding the useful variable reset could not be done automatically.

Other specifications. We also applied **stategraph** to all the example specifications of μ CRL, and five from industry:

- Two versions of an Automatic In-flight Data Acquisition unit for a helicopter of the Dutch Royal Navy [10];
- A cache coherence protocol for a distributed JVM [19];
- An automatic translation from Erlang to μ CRL of a distributed resource locker in Ericsson’s AXD 301 switch [2];
- The sliding window protocol (with three data elements and window size two) [3].

The same analysis as before was performed, but now also counting the number of summands and parameters of the reduced LPEs. Decreases of these quantities are due to **stategraph** resetting variables to their initial value, which may turn them into constants and have them removed. As a side effect, some summands might be removed as their enabling condition is shown to never be satisfied. These effects provide a syntactical cleanup and fasten state space generation, as seen for instance from the **ccp221** and **locker** specifications.

The reductions obtained are shown in Table 2; values that differ significantly are listed in boldface. Not all example specifications benefited from **stategraph** (these are omitted from the table). This is partly because **parelm** already performs a rudimentary variant of our method, and also because the lineariser removes parameters that are syntactically out of scope. However, although optimising LPEs has been the focus for years, **stategraph** could still reduce some of the standard examples. Especially for the larger, industrial specifications reductions in state space, but also in the number of summands and parameters of the

specification	constelm parelm constelm				constelm stategraph constelm			
	time	states	summands	pars	time	states	summands	pars
bke	0:47.9	79,949	50	31	0:48.3	79,949	50	21
ccp33	–	–	1082	97	–	–	807	94
onebit	0:25.1	319,732	30	26	0:21.4	269,428	30	26
AIDA-B	7:50.1	3,500,040	89	35	7:11.9	3,271,580	89	32
AIDA	0:40.1	318,682	85	35	0:30.8	253,622	85	32
ccp221	0:28.3	76,227	562	63	0:25.6	76,227	464	62
locker	1:43.3	803,830	88	72	1:32.9	803,830	88	19
swp32	0:11.7	156,900	13	12	0:11.8	156,900	13	12

Table 2. Modelling several specifications; `parelm` versus `stategraph`.

linearised form were obtained. Both results are shown to speed up state space generation, proving `stategraph` to be a valuable addition to the μ CRL toolkit.

7 Conclusions and Future Work

We presented a novel method for reconstructing the control flow of linear processes. This information is used for data flow analysis, aiming at state space reduction by resetting variables that are irrelevant given a certain state. We introduced a transformation and proved both its preservation of strong bisimilarity, and its property to never increase the state space. The reconstruction process enables us to interpret some variables as program counters; something other tools are not able to. Case studies using our implementation `stategraph` showed that although for some small academic examples the existing tools already suffice, impressive state space reductions can be obtained for larger, industrial systems. Since we work on linear processes, these reductions are obtained before the entire state space is generated, saving valuable time. Surprisingly, a recently implemented symbolic tool for μ CRL also profits much from `stategraph`.

As future work it would be interesting to find additional applications for the reconstructed control flow. One possibility is to use it for invariant generation, another (already implemented) is to visualise it such that process structure can be understood better. Also, it might be used to optimise confluence checking [5], since it could assist in determining which pairs of summands may be confluent.

Another direction for future work is based on the insight that the control flow graph is an abstraction of the state space. It could be investigated whether other abstractions, such as a control flow graph containing also the values of important data parameters, might result in more accurate data flow analysis.

Acknowledgements. We thank Jan Friso Groote for his specification of the handshake register, upon which our model has been based. Furthermore, we thank Michael Weber for fruitful discussions about Hesselink’s protocol.

References

- [1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [2] T. Arts, C.B. Earle, and J. Derrick. Verifying Erlang code: A resource locker case-study. In *Proceedings of the 11th International Symposium of Formal Methods (FM '02)*, volume 2391 of *Lecture Notes in Computer Science*, pages 184–203. Springer, 2002.
- [3] B. Badban, W. Fokkink, J.F. Groote, J. Pang, and J. van de Pol. Verification of a sliding window protocol in μ CRL and PVS. *Formal Aspects of Computing*, 17(3):342–388, 2005.
- [4] M. Bezem and J.F. Groote. Invariants in process algebra with data. In *Proceedings of the 5th International Conference on Concurrency Theory (CONCUR '94)*, volume 836 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 1994.
- [5] S. Blom and J. van de Pol. State space reduction by proving confluence. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 596–609. Springer, 2002.
- [6] S. Blom and J. van de Pol. Symbolic reachability for process algebras with recursive data types. In *Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing (ICTAC '08)*, volume 5160 of *Lecture Notes in Computer Science*, pages 81–95. Springer, 2008.
- [7] M. Bozga, J.-C. Fernandez, and L. Ghirvu. State space reduction based on live variables analysis. In *Proceedings of the 6th International Symposium on Static Analysis (SAS '99)*, volume 1694 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 1999.
- [8] K.M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley, 1988.
- [9] J.-C. Fernandez and L. Mounier. “On the Fly” Verification of Behavioural Equivalences and Preorders. In *Proceedings of the 3rd International Workshop on Computer Aided Verification (CAV '91)*, volume 575 of *Lecture Notes in Computer Science*, pages 181–191. Springer, 1991.
- [10] W. Fokkink, N. Ioustinova, E. Kessler, J. van de Pol, Y.S. Usenko, and Y.A. Yushtein. Refinement and verification applied to an in-flight data acquisition unit. In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR '02)*, volume 2421 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2002.
- [11] H. Garavel and W. Serwe. State space reduction for process algebra specifications. *Theoretical Computer Science*, 351(2):131–145, 2006.
- [12] J.F. Groote and B. Lissner. Computer assisted manipulation of algebraic process specifications. Technical report, SEN-R0117, CWI, 2001.
- [13] J.F. Groote and A. Ponse. The syntax and semantics of μ CRL. In *Proceedings of the 1st Workshop on the Algebra of Communicating Processes (ACP '94)*, pages 26–62. Springer, 1994.
- [14] J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1-2):39–72, 2001.
- [15] J.F. Groote and J. van de Pol. State space reduction using partial τ -confluence. In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS '00)*, volume 1893 of *Lecture Notes in Computer Science*, pages 383 – 393, 2000.

- [16] W.H. Hesselink. Invariants for the construction of a handshake register. *Information Processing Letters*, 68(4):173 – 177, 1998.
- [17] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.
- [18] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [19] J. Pang, W. Fokkink, R.F.H. Hofman, and R. Veldema. Model checking a cache coherence protocol of a Java DSM implementation. *Journal of Logic and Algebraic Programming*, 71(1):1–43, 2007.
- [20] Y.S. Usenko. *Linearization in μCRL* . PhD thesis, Eindhoven University of Technology, 2002.
- [21] B.D. Winters and A.J. Hu. Source-level transformations for improved formal verification. In *Proceedings of the 18th IEEE International Conference On Computer Design (ICCD '00)*, pages 599–602, 2000.
- [22] K. Yorav and O. Grumberg. Static analysis for state-space reductions preserving temporal logics. *Formal Methods in System Design*, 25(1):67–96, 2004.

A Further insights

This section provides further insights about the theory developed in this paper.

First, we describe two additional reduction techniques that can be implemented based on our control flow analysis. Although they do not reduce the state space, they are still useful to cleanup the LPE (sometimes even enabling further reductions).

Second, we discuss some (potential) limitations that were noticed while developing the theory; the bad influence of action clustering on the reductions that can be obtained, and the lack of idempotency of our method.

Third, we cover some potential adaptations to the definitions, that intuitively would seem like an improvement, but can also cause difficulties.

A.1 Additional reduction: removing dead code

For each CFP d_j , we can easily compute an overapproximation of the set of all its reachable values $Reach_j$ using Algorithm 1. Note that it might contain values that are in fact never reachable, as conditions containing other parameters might prevent some summands from being executed.

Proposition 3. *All values that a CFP d_j might obtain are contained in $Reach_j$ after executing Algorithm 1.*

Proof. The proof is by induction on the number of transitions (by summands in the cluster of d_j) that has to be taken to reach a value. We will prove that all values that d_j might take in n such transitions are included before or during iteration n (where the first iteration has index 0). Note that the number of values d_j might take is finite, as it is limited by the number of summands.

The only value d_j might have after 0 transitions is $init_j$. By step 1 of the algorithm, this value is indeed included, so all values d_j might take in 0 transitions are included in $Reach_j$ before or during iteration 0.

Now assume that all values d_j might take in k transitions are included in $Reach_j$ before or during iteration k . Furthermore, let v be a value d_j might

Algorithm 1: Reachable values

```
1  $Reach_j = \{ init_j \};$ 
2  $Prev_j = \emptyset;$ 
3 while  $Reach_j \neq Prev_j$  do
4    $Prev_j := Reach_j;$ 
5   forall  $s \in Reach_j$  do
6     forall  $i \in R_{d_j}$  such that  $s = \text{source}(i, d_j)$  do
7        $Reach_j := Reach_j \cup \{ \text{dest}(i, d_j) \};$ 
8     end
9   end
10 end
```

obtain in $k + 1$ transitions, so $init_j = v^0 \rightarrow v^1 \rightarrow \dots \rightarrow v^k \rightarrow v^{k+1} = v$. By the induction hypothesis $v^k \in Reach_j$ during iteration $k + 1$. Furthermore, since $v^k \rightarrow v$, there must be some summand $i \in R_{d_j}$ such that $v^k = source(i, d_j)$ and $v = dest(i, d_j)$. Hence, v^{k+1} is added during iteration $k + 1$ by step 7 of the algorithm.

Termination of the algorithm immediately follows from the observation that the number of values a CFP might take is finite. \square

Now, it immediately follows that a summand $i \in I$ can be removed if, for some CFP d_j that rules i , $source(i, d_j) \notin Reach_j$. After all, in this case the enabling condition of i will never be satisfied. From the operational semantics it then follows that i does not contribute to the behaviour of the system. This reduction technique has already been implemented in `stategraph`.

A.2 Additional reduction: changing the initial state

The lineariser of the μ CRL toolkit chooses dummy values for parameters whose initial value does not matter. These values are chosen locally per component. However, after generating an LPE global information is available, making it possible to choose more intelligent values. If possible, the initial value of a parameter should be chosen such that it is not changed by any summand. In that case, it can be removed by constant elimination [12].

From Theorem 1, it immediately follows that the initial value of a DP d_k , belonging to a CFP d_j , can be changed if $\neg R(d_k, d_j, init_j)$. This strategy is already applied by `stategraph`.

A.3 Limitation: bad effect of clustering

For efficiency purposes, the μ CRL lineariser performs clustering; it takes summands with the same action labels together, by replacing action parameters and conditions by conditionals. For example, consider the following LPE.

$$\begin{aligned}
X(p: \{1, 2\}, q: \{1, 2\}, x: \text{Nat}) = & \\
p = 2 \wedge q = 2 \Rightarrow a(x) \cdot X(1, 1, x + 1) & \quad (1) \\
+ \quad p = 1 \quad \quad \quad \Rightarrow \tau \cdot X(2, q, x) & \quad (2) \\
+ \quad \quad \quad q = 1 \Rightarrow \tau \cdot X(p, 2, 0) & \quad (3)
\end{aligned}$$

Clearly, q is a CFP and x belongs to q . Also, it is easy to see that x is relevant when $q = 2$, but not when $q = 1$. Therefore, the next-state vector of the first summand can be changed to $(1, 1, 0)$.

Since the second and the third summand perform the same action, they can be (and normally are) clustered as follows.

$$\begin{aligned}
X(p: \{1, 2\}, q: \{1, 2\}, x: \text{Nat}) = & \\
p = 2 \wedge q = 2 \quad \Rightarrow a(x) \cdot X(1, 1, x + 1) & \quad (1) \\
+ \sum_{e: \text{Bool}} \text{if}(e, p = 1, q = 1) \Rightarrow \tau \cdot X(\text{if}(e, 2, p), \text{if}(e, q, 2), \text{if}(e, x, 0)) & \quad (2)
\end{aligned}$$

where $\text{if}(e, x, y)$ is assumed to evaluate to x when $e = \text{true}$ and to y when $e = \text{false}$. Now, however, p and q do not have a unique source for the second summand anymore, so none of them rules this summand. Since x is used in this summand, it belongs to neither p nor q , and no reduction can be made. Therefore, it seems wise to disable the μCRL clustering feature (experiments indeed showed that more reductions are obtained with clustering disabled).

A.4 Limitation: lack of idempotency

Generally it is considered desirable for a reduction technique to be idempotent; applying it once yields the maximum effect it can possibly achieve, and applying it more often does not have any additional effect. Unfortunately, our method is not idempotent in general, although experiments show that in practice it often is. As an example of the potential lack of idempotency, consider the following LPE.

$$\begin{aligned}
X(p: \{1, 2, 3\}, q: \{1, 2\}, x: \text{Nat}) = & \\
p = 2 \wedge q = 1 & \Rightarrow a(x) \cdot X(1, 1, x) \quad (1) \\
+ \quad p = 1 \wedge q = 1 & \Rightarrow \tau \cdot X(2, 2, x + 1) \quad (2) \\
+ \quad p = 3 & \Rightarrow \tau \cdot X(1, q, x) \quad (3)
\end{aligned}$$

For this LPE, p and q are both CFPs. Furthermore, p rules all summands, whereas q only rules the first two. Since x is unchanged in the third summand, however, it still belongs to both.

Clearly, x is relevant when $p = 2$ and when $q = 1$, since in that case the first summand (which directly uses x) might be taken. Moreover, x is relevant when $p = 1$, since in that case the second summand can be taken, which uses x to go to $p = 2$. Finally, x is relevant when $p = 3$ because of the third summand.

However, because the second summand changes q to 2 and x is not relevant when $q = 2$, its next-state vector can be changed into $(2, 2, 0)$, obtaining

$$\begin{aligned}
X(p: \{1, 2, 3\}, q: \{1, 2\}, x: \text{Nat}) = & \\
p = 2 \wedge q = 1 & \Rightarrow a(x) \cdot X(1, 1, x) \quad (1) \\
+ \quad p = 1 \wedge q = 1 & \Rightarrow \tau \cdot X(2, 2, 0) \quad (2) \\
+ \quad p = 3 & \Rightarrow \tau \cdot X(1, q, x) \quad (3)
\end{aligned}$$

The next-state vector of the third summand could not (yet) be transformed, since it changes p to 1 and it was established that x is relevant when $p = 1$.

However, starting over based on the transformed LPE, it can be seen that x is not relevant anymore when $p = 1$, since it is not used in the next-state function. Therefore, x is also not relevant anymore for $p = 3$, so that now the next-state vector of the third summand can be changed into $(1, q, 0)$.

A.5 Potential adaption: allowing CFPs to belong to other CFPs

A possible adaption to the theory is to allow CFPs to belong to other CFPs. This, however, would raise problems in case cycles occur in the belongs-to relation.

Consider for example the following LPE.

$$\begin{aligned} X(p: \{1, 2\}, q: \{1, 2\}) = \\ p = 1 \wedge q = 1 \Rightarrow \tau \cdot X(2, 2) \end{aligned} \quad (1)$$

with $(2, 2)$ as the initial state vector. Clearly, this system can perform no actions.

If CFPs could belong to CFPs, in this case p would belong to q and q would belong to p . Furthermore, we obtain $\neg R(p, q, 2)$ and $\neg R(q, p, 2)$. Therefore, the initial condition seems to be allowed to change to $(1, 1)$. However, in that case we obtain a specification that is not strongly bisimilar to the original anymore, since it can perform a τ .

A.6 Potential adaption: relieving the definition of belongs-to

The definition of belongs-to could be relieved to also allow DPs to be changed to a constant value in summands that are not ruled by the CFP they belong to. That is, a DP d_k belongs to a CFP d_j if all summand $i \in I$ that use or change d_k to a *non-constant value* are ruled by d_j . In this case, the theorems stating validity after reduction based on the definition of belongs-to still hold; the proofs can easily be adapted. As an example of a useful application of this change, observe

$$\begin{aligned} X(p: \{1, 2\}, x: \text{Nat}) = \\ p = 1 \Rightarrow \tau \cdot X(1, x + 1) \quad (1) \\ + \quad p = 2 \Rightarrow a(x) \cdot X(1, x) \quad (2) \\ + \quad \tau \cdot X(p, 0) \quad (3) \end{aligned}$$

For this infinite-state system parameter p rules the first two summands, but x is changed in the last so according to the original definition x would not belong to p and no reductions can be made. However, using the new definition, x does belong to p , and we observe that x is only relevant when $p = 2$. Therefore, we can reduce to the following LPE, obtaining a finite-state system.

$$\begin{aligned} X(p: \{1, 2\}, x: \text{Nat}) = \\ p = 1 \Rightarrow \tau \cdot X(1, 0) \quad (1) \\ + \quad p = 2 \Rightarrow a(x) \cdot X(1, 0) \quad (2) \\ + \quad \tau \cdot X(p, 0) \quad (3) \end{aligned}$$

However, using the adapted definition it is also possible to obtain a growth of the state space because of a ‘reduction’; consider for instance the following LPE.

$$\begin{aligned} X(p: \{1, 2\}, q: \{1, 2\}, x: \{d_1, d_2\}) = \\ p = 1 \Rightarrow \tau \cdot X(2, q, d_2) \quad (1) \\ + \quad q = 1 \Rightarrow a(x) \cdot X(p, 2, x) \quad (2) \end{aligned}$$

Using the initial state vector $(1, 1, d_1)$, this system has a state space consisting of four states. Using the adapted definition of belongs-to we find that x belongs to q , and that x is not relevant when $q = 2$, so that reduction results in

$$X(p: \{1, 2\}, q: \{1, 2\}, x: \{d_1, d_2\}) =$$

$$p = 1 \Rightarrow \tau \cdot X(2, q, d_2) \quad (1)$$

$$+ q = 1 \Rightarrow a(x) \cdot X(p, 2, d_1) \quad (2)$$

This specification has five states, so clearly Corollary 2 does not apply anymore.

A.7 Potential adaption: simplifying the definition of relevance

It seems reasonable to merge the second and third clause of relevance. That is, we would replace the second and the third clause by

2. If $R(d_i, d_p, t)$, and there exists an $i \in I$ and an r such that $(r, i, t) \in E_{d_p}$, and $d_k \in \text{pars}(g_{i,i}(\mathbf{d}, \mathbf{e}_i))$, and d_k belongs to some cluster d_j , and $s = \text{source}(i, d_j)$, then $R(d_k, d_j, s)$.

Indeed, the proof of Lemma 5, which uses this definition, can easily be adapted to still be valid for the merged definition. However, this decreases the number of reductions that can be made. As an example, observe the following LPE.

$$X(p: \{1, 2\}, q: \{1, 2\}, x: \text{Nat}) =$$

$$p = 1 \wedge q = 1 \Rightarrow a(x) \cdot X(2, 1, x) \quad (1)$$

$$+ p = 2 \wedge q = 1 \Rightarrow \tau \cdot X(1, 1, 2) \quad (2)$$

$$+ p = 2 \wedge q = 2 \Rightarrow \tau \cdot X(2, 1, x) \quad (3)$$

Clearly x belongs to both p and q . We first use the original definition of relevance. It follows immediately from the first clause of this definition, and the first summand of X , that $R(x, p, 1)$ and $R(x, q, 1)$. Using the second clause of relevance we then obtain $R(x, q, 2)$ by looking at the third summand. Now no clause applies anymore, resulting in the observation that $\neg R(x, p, 2)$, hence x can be reset in the next-state function of the first summand.

Using the merged definition, on the other hand, the third summand combined with $R(x, q, 1)$ would yield $R(x, p, 2)$, so no reductions can be made anymore. The problem here is that x seems to be relevant when $p = 2$, since in that case the third summand might be taken, after which $q = 1$ (and we already knew $R(x, q, 1)$). However, after taking the third summand p will still be 2, preventing the first summand from being taken.