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 [12]. Although LPEs are a restricted part of μCRL, every specification can be transformed to an LPE by a procedure called *linearisation* [13, 19]. 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).

Z. Liu and A.P. Ravn (Eds.): ATVA 2009, LNCS 5799, pp. 54–68, 2009.

on the current state is true. It has already been shown useful to reduce LPEs directly (e.g. [5, 14]), 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.

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 [10], 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 μCRL 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 [11]. 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 μCRL 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.

Due to space limitations, we refer the reader to [20] for the full version of the current paper, containing all the complete proofs, and further insights about additional reductions, potential limitations, and potential adaptions to our theory.

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 [16]. Usenko showed how to transform a general μCRL specification into an LPE [13, 19].

Each LPE is of the form

$$X(\boldsymbol{d}\colon \boldsymbol{D}) = \sum_{i \in I} \sum_{\boldsymbol{e_i}\colon \boldsymbol{E_i}} c_i(\boldsymbol{d}, \boldsymbol{e_i}) \Rightarrow a_i(\boldsymbol{d}, \boldsymbol{e_i}) \cdot X(g_i(\boldsymbol{d}, \boldsymbol{e_i})),$$

where $\boldsymbol{D}$ is a type for *state vectors* (containing the global variables), I a set of *summand indices*, and $\boldsymbol{E_i}$ 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 $\boldsymbol{init}$.

Given a vector of formal state parameters $\boldsymbol{d}$, we use d_j to refer to its j^{th} parameter. An actual state is a vector of values, denoted by $\boldsymbol{v}$; 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, $g_{i,j}(\boldsymbol{d}, \boldsymbol{e_i})$ denotes the j^{th} element of $g_i(\boldsymbol{d}, \boldsymbol{e_i})$, 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 $\boldsymbol{v}$ to $\boldsymbol{v'}$ by an atomic action $a(\boldsymbol{p})$ (parameterised by the possibly empty vector $\boldsymbol{p}$) if and only if there is a summand i for which a vector of local variables $\boldsymbol{e_i}$ exists such that the enabling function is true, the action is $a(\boldsymbol{p})$ and the next-state function yields $\boldsymbol{v'}$. Formally, for all $\boldsymbol{v}, \boldsymbol{v'} \in \boldsymbol{D}$, there is a transition $\boldsymbol{v} \xrightarrow{a(\boldsymbol{p})} \boldsymbol{v'}$ if and only if there is a summand i such that

$$\exists \boldsymbol{e_i} \in \boldsymbol{E_i} \cdot c_i(\boldsymbol{v}, \boldsymbol{e_i}) \wedge a_i(\boldsymbol{v}, \boldsymbol{e_i}) = a(\boldsymbol{p}) \wedge g_i(\boldsymbol{v}, \boldsymbol{e_i}) = \boldsymbol{v'}.$$

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

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

put in parallel and communicating on w and r. Linearised [19], they become

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

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* [17], 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 $g_{i,j}(\boldsymbol{d}, \boldsymbol{e_i}) \neq d_j$, otherwise it is* unchanged *in i. It is* directly used *in i if $d_j \in \mathrm{pars}(a_i(\boldsymbol{d}, \boldsymbol{e_i})) \cup \mathrm{pars}(c_i(\boldsymbol{d}, \boldsymbol{e_i}))$, and* used *in i if it is directly used in i or $d_j \in \mathrm{pars}(g_{i,k}(\boldsymbol{d}, \boldsymbol{e_i}))$ 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 $f\colon I \times (d_j{:}J) \to D_j \cup \{\bot\}$ 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*

$$\forall \boldsymbol{v} \in \boldsymbol{D}, \boldsymbol{e_i} \in \boldsymbol{E_i} \cdot c_i(\boldsymbol{v}, \boldsymbol{e_i}) \implies v_j = s.$$

Furthermore, $f(i, d_j) = \bot$ 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 $\mathrm{source}(i, d_j)$ is allowed to be $\bot$ 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 $\bot$ 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 $c_i(\boldsymbol{d}, \boldsymbol{e_i})$ be given by $(d_j = 3 \vee d_j = 5) \wedge d_j = 3 \wedge d_k = 10$, then obviously $\mathrm{source}(i, d_j) = 3$ is valid (because $(\{3\} \cup \{5\}) \cap \{3\} = \{3\}$), but also (as always) $\mathrm{source}(i, d_j) = \bot$.

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 $f\colon I \times (d_j{:}J) \to D_j \cup \{\bot\}$ 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*

$$\forall \boldsymbol{v} \in \boldsymbol{D}, \boldsymbol{e_i} \in \boldsymbol{E_i} \cdot c_i(\boldsymbol{v}, \boldsymbol{e_i}) \implies g_{i,j}(\boldsymbol{v}, \boldsymbol{e_i}) = s.$$

Furthermore, $f(i, d_j) = \bot$ 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 $\mathrm{dest}(i, d_j)$ just substitute $\mathrm{source}(i, d_j)$ for d_j in the next-state function of summand i, and try to rewrite it to a closed term.

Example 3. Let $c_i(\boldsymbol{d}, \boldsymbol{e_i})$ be given by $d_j = 8$ and $g_{i,j}(\boldsymbol{d}, \boldsymbol{e_i})$ by $d_j + 5$, then $\mathrm{dest}(i, d_j) = 13$ is valid, but also (as always) $\mathrm{dest}(i, d_j) = \bot$. If for instance $c_i(\boldsymbol{d}, \boldsymbol{e_i}) = d_j = 5$ and $g_{i,j}(\boldsymbol{d}, \boldsymbol{e_i}) = e_3$, then $\mathrm{dest}(i, d_j)$ can only yield $\bot$, 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* $\mathrm{source}(i, d_j) \neq \bot$ *and* $\mathrm{dest}(i, d_j) \neq \bot$.

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,*

$$V_{d_j} = \{\, \mathrm{source}(i, d_j) \mid i \in R_{d_j} \,\} \cup \{\, \mathrm{dest}(i, d_j) \mid i \in R_{d_j} \,\} \cup \{\, init_j \,\}.$$

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* $\mathcal{D}$.

Example 4. Consider the LPE of Example 1 again. For the first summand we may define $\mathrm{source}(1, a) = 1$ and $\mathrm{dest}(1, a) = 2$. 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*

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

(a) Control flow graph for a. (b) Control flow graph for b.

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

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 $\boldsymbol{v} = (v_1, \ldots, v_j, \ldots, v_n)$ and local variable vector $\boldsymbol{e_i}$, then the control flow graph of d_j contains an edge from v_j to $g_{i,j}(\boldsymbol{v}, \boldsymbol{e_i})$.

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

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 $b =$ `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 $d_k \in \text{pars}(g_{i,l}(\boldsymbol{d}, \boldsymbol{e_i}))$, 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 $d_k \in \text{pars}(g_{i,l}(\boldsymbol{d}, \boldsymbol{e_i}))$, 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 [20] 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

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

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 $\boldsymbol{v}$, 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 $\boldsymbol{v}$, denoted $Relevant(d_k, \boldsymbol{v})$, is defined by*

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

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 $\boldsymbol{v} = (1, 2, d_5, d_2)$, so we write $\neg\, \mathit{Relevant}(x, \boldsymbol{v})$.

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

$$\boldsymbol{v} \cong \boldsymbol{v'} \iff \forall d_k \in J\colon\ (\mathit{Relevant}(d_k, \boldsymbol{v}) \implies v_k = v'_k)\,,$$

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

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

Proof (sketch). It is easy to see that $\cong$ is an equivalence relation (1). Then, it can be proven that if a summand i is enabled given a state vector $\boldsymbol{v}$, it is also enabled given a state vector $\boldsymbol{v'}$ such that $\boldsymbol{v} \cong \boldsymbol{v'}$ (2). Finally, it can be shown that if a summand i is taken given $\boldsymbol{v}$, its action is identical to when i is taken given $\boldsymbol{v'}$ (3), and their next-state vectors are equivalent according to $\cong$ (4).

Now, let $\boldsymbol{v_0}$ and $\boldsymbol{v'_0}$ be state vectors such that $\boldsymbol{v_0} \cong \boldsymbol{v'_0}$. Also, assume that $\boldsymbol{v_0} \xrightarrow{a} \boldsymbol{v_1}$. By (1) $\cong$ is symmetric, so we only need to prove that a transition $\boldsymbol{v'_0} \xrightarrow{a} \boldsymbol{v'_1}$ exists such that $\boldsymbol{v_1} \cong \boldsymbol{v'_1}$.

By the operational semantics there is a summand i and a local variable vector $\boldsymbol{e_i}$ such that $c_i(\boldsymbol{v_0}, \boldsymbol{e_i})$ holds, $a = a_i(\boldsymbol{v_0}, \boldsymbol{e_i})$, and $\boldsymbol{v_1} = g_i(\boldsymbol{v_0}, \boldsymbol{e_i})$. Now, by (2) we know that $c_i(\boldsymbol{v'_0}, \boldsymbol{e_i})$ holds, and by (3) that $a = a_i(\boldsymbol{v'_0}, \boldsymbol{e_i})$. Therefore, $\boldsymbol{v'_0} \xrightarrow{a} g_i(\boldsymbol{v'_0}, \boldsymbol{e_i})$. Using (4) we get $g_i(\boldsymbol{v_0}, \boldsymbol{e_i}) \cong g_i(\boldsymbol{v'_0}, \boldsymbol{e})$, proving the theorem. □

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

$$X'(\boldsymbol{d}\colon \boldsymbol{D}) = \sum_{i \in I} \sum_{\boldsymbol{e_i}\colon \boldsymbol{E_i}} c_i(\boldsymbol{d}, \boldsymbol{e_i}) \Rightarrow a_i(\boldsymbol{d}, \boldsymbol{e_i}) \cdot X'(g'_i(\boldsymbol{d}, \boldsymbol{e_i})),$$

with

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

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

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

Lemma 1. *For every $i \in I$, state vector $\boldsymbol{v}$, and local variable vector $\boldsymbol{e_i}$, given that $c_i(\boldsymbol{v}, \boldsymbol{e_i}) = \texttt{true}$ it holds that $g_i(\boldsymbol{v}, \boldsymbol{e_i}) \mathrel{\tilde{=}} g'_i(\boldsymbol{v}, \boldsymbol{e_i})$.*

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

Theorem 2. *Let $\mathrel{\tilde{\tilde{=}}}$ be defined by*

$$X(\boldsymbol{v}) \mathrel{\tilde{\tilde{=}}} X'(\boldsymbol{v'}) \iff \boldsymbol{v} \mathrel{\tilde{=}} \boldsymbol{v'},$$

then $\mathrel{\tilde{\tilde{=}}}$ is a strong bisimulation. The relation $\mathrel{\tilde{=}}$ is used as it was defined for X.

Proof. Let $\boldsymbol{v_0}$ and $\boldsymbol{v'_0}$ be state vectors such that $X(\boldsymbol{v_0}) \mathrel{\tilde{\tilde{=}}} X'(\boldsymbol{v'_0})$, so $\boldsymbol{v_0} \mathrel{\tilde{=}} \boldsymbol{v'_0}$.

Assume that $X(\boldsymbol{v_0}) \xrightarrow{a} X(\boldsymbol{v_1})$. We need to prove that there exists a transition $X'(\boldsymbol{v'_0}) \xrightarrow{a} X'(\boldsymbol{v'_1})$ such that $X(\boldsymbol{v_1}) \mathrel{\tilde{\tilde{=}}} X'(\boldsymbol{v'_1})$. By Theorem 1 there exists a state vector $\boldsymbol{v''_1}$ such that $X(\boldsymbol{v'_0}) \xrightarrow{a} X(\boldsymbol{v''_1})$ and $\boldsymbol{v_1} \mathrel{\tilde{=}} \boldsymbol{v''_1}$. By the operational semantics, for some i and $\boldsymbol{e_i}$ we thus have $c_i(\boldsymbol{v'_0}, \boldsymbol{e_i})$, $a_i(\boldsymbol{v'_0}, \boldsymbol{e_i}) = a$, and $g_i(\boldsymbol{v'_0}, \boldsymbol{e_i}) = \boldsymbol{v''_1}$. By Definition 10, we have $X'(\boldsymbol{v'_0}) \xrightarrow{a} X'(g'_i(\boldsymbol{v'_0}, \boldsymbol{e_i}))$, and by Lemma 1 $g_i(\boldsymbol{v'_0}, \boldsymbol{e_i}) \mathrel{\tilde{=}} g'_i(\boldsymbol{v'_0}, \boldsymbol{e_i})$. Now, by transitivity and reflexivity of $\mathrel{\tilde{=}}$ (Statement (1) of the proof of Theorem 1), $\boldsymbol{v_1} \mathrel{\tilde{=}} \boldsymbol{v''_1} = g_i(\boldsymbol{v'_0}, \boldsymbol{e_i}) \mathrel{\tilde{=}} g'_i(\boldsymbol{v'_0}, \boldsymbol{e_i})$, hence $X(\boldsymbol{v_1}) \mathrel{\tilde{\tilde{=}}} X'(g'_i(\boldsymbol{v'_0}, \boldsymbol{e_i}))$. By symmetry of $\mathrel{\tilde{=}}$, this completes the proof. □

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

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

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

Proposition 2. *For $X'(\boldsymbol{init})$ invariably $\neg\, Relevant(d_k, \boldsymbol{v})$ implies $v_k = init_k$.*

Using this invariant it is possible to prove the following lemma, providing a functional strong bisimulation relating the states of $X(\boldsymbol{init})$ and $X'(\boldsymbol{init})$.

Lemma 2. *Let h be a function over state vectors, such that for any $\boldsymbol{v} \in \boldsymbol{D}$ it is given by $h_k(\boldsymbol{v}) = v_k$ if $Relevant(d_k, \boldsymbol{v})$, and by $h_k(\boldsymbol{v}) = init_k$ otherwise. Then, h is a strong bisimulation relating the states of $X(\boldsymbol{init})$ and $X'(\boldsymbol{init})$.*

Since the bisimulation relation 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

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

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 [15]. 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 implementation using four safe registers. We generated their state spaces, minimised, 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 μCRL tool `constelm` [11]) 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 tool[1] 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).

[1] Available from `http://fmt.cs.utwente.nl/tools/ltsmin`

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

	`constelm` | `parelm` | `constelm`			`constelm` | `stategraph` | `constelm`		
	states	time (expl.)	time (symb.)	states	time (expl.)	time (symb.)
$\|D\| = 2$	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

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 μCRL specification of the four safe registers below.

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

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, or, when writing is taking place at the same time, a random value is copied to vr. Writing works by first storing the value to be written in vw, and then copying vw to v.

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.

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

	`constelm | parelm | constelm`				`constelm | stategraph | constelm`			
specification	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

Although the example may seem artificial, it is an almost one-to-one formalisation of its description in [15]. 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 μCRL, and five from industry: two versions of an Automatic In-flight Data Acquisition unit for a helicopter of the Dutch Royal Navy [9]; a cache coherence protocol for a distributed JVM [18]; an automatic translation from Erlang to μCRL of a distributed resource locker in Ericsson's AXD 301 switch [2]; and 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 linearised form were obtained. Both results are shown to speed up state space generation, proving `stategraph` to be a valuable addition to the μCRL 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 μCRL 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] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
[2] Arts, T., Earle, C.B., Derrick, J.: Verifying Erlang code: A resource locker case-study. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 184–203. Springer, Heidelberg (2002)
[3] Badban, B., Fokkink, W., Groote, J.F., Pang, J., van de Pol, J.: Verification of a sliding window protocol in μCRL and PVS. Formal Aspects of Computing 17(3), 342–388 (2005)
[4] Bezem, M., Groote, J.F.: Invariants in process algebra with data. In: Jonsson, B., Parrow, J. (eds.) CONCUR 1994. LNCS, vol. 836, pp. 401–416. Springer, Heidelberg (1994)
[5] Blom, S., van de Pol, J.: State space reduction by proving confluence. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 596–609. Springer, Heidelberg (2002)
[6] Blom, S., van de Pol, J.: Symbolic reachability for process algebras with recursive data types. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 81–95. Springer, Heidelberg (2008)
[7] Bozga, M., Fernandez, J.-C., Ghirvu, L.: State space reduction based on live variables analysis. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 164–178. Springer, Heidelberg (1999)
[8] Chandy, K.M., Misra, J.: Parallel program design: a foundation. Addison-Wesley, Reading (1988)

[9] Fokkink, W., Ioustinova, N., Kesseler, E., van de Pol, J., Usenko, Y.S., Yushtein, Y.A.: Refinement and verification applied to an in-flight data acquisition unit. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 1–23. Springer, Heidelberg (2002)
[10] Garavel, H., Serwe, W.: State space reduction for process algebra specifications. Theoretical Computer Science 351(2), 131–145 (2006)
[11] Groote, J.F., Lisser, B.: Computer assisted manipulation of algebraic process specifications. Technical report, SEN-R0117, CWI (2001)
[12] Groote, J.F., Ponse, A.: The syntax and semantics of μCRL. In: Proc. of the 1st Workshop on the Algebra of Communicating Processes (ACP 1994), pp. 26–62. Springer, Heidelberg (1994)
[13] Groote, J.F., Ponse, A., Usenko, Y.S.: Linearization in parallel pCRL. Journal of Logic and Algebraic Programming 48(1-2), 39–72 (2001)
[14] Groote, J.F., van de Pol, J.: State space reduction using partial τ-confluence. In: Nielsen, M., Rovan, B. (eds.) MFCS 2000. LNCS, vol. 1893, pp. 383–393. Springer, Heidelberg (2000)
[15] Hesselink, W.H.: Invariants for the construction of a handshake register. Information Processing Letters 68(4), 173–177 (1998)
[16] Lynch, N., Tuttle, M.: An introduction to input/output automata. CWI-Quarterly 2(3), 219–246 (1989)
[17] Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
[18] Pang, J., Fokkink, W., Hofman, R.F.H., Veldema, R.: Model checking a cache coherence protocol of a Java DSM implementation. Journal of Logic and Algebraic Programming 71(1), 1–43 (2007)
[19] Usenko, Y.S.: Linearization in μCRL. PhD thesis, Eindhoven University (2002)
[20] van de Pol, J., Timmer, M.: State space reduction of linear processes using control flow reconstruction (extended version). Technical report, TR-CTIT-09-24, CTIT, University of Twente (2009)
[21] Winters, B.D., Hu, A.J.: Source-level transformations for improved formal verification. In: Proc. of the 18th IEEE Int. Conference on Computer Design (ICCD 2000), pp. 599–602 (2000)
[22] Yorav, K., Grumberg, O.: Static analysis for state-space reductions preserving temporal logics. Formal Methods in System Design 25(1), 67–96 (2004)