We present a formal calculus, termed chemtainer calculus, able to capture the complexity of compartmentalized reaction systems such as populations of possibly nested vesicular compartments. Compartments contain molecular cargo as well as surface markers in the form of DNA single strands. These markers serve as compartment addresses and allow for their targeted transport and fusion, thereby enabling reactions of previously separated chemicals. The overall system organization allows for the setup of programmable chemistry in microfluidic or other automated environments. We introduce a simple sequential programming language whose instructions are motivated by state-of-the-art microfluidic technology. Our approach integrates electronic control, chemical computing, and material production in a unified formal framework that is able to mimic the integrated computational and constructive capabilities of the subcellular matrix. We provide a non-deterministic semantics of our programming language that enables us to analytically derive the computational and constructive power of our machinery. This semantics is used to derive the sets of all constructable chemicals and supermolecular structures that emerge from different underlying instruction sets. Since our proofs are constructive, they can be utilized to automatically infer control programs for the construction of target structures from a limited set of resource molecules. Finally, we present an example of our framework from the area of oligosaccharide synthesis.

I. INTRODUCTION

Living systems are unique in that they integrate molecular recognition and information processing with material production on the molecular scale. The predominant locus of this integration is the cytoplasm, where a multitude of biochemical compounds is highly organized in vesicular compartments that co-locate reactants of desired reactions while separating those of undesired reactions. Surface markers on these compartments are used for vesicular trafficking, as well as vesicle budding and fusion, thereby allowing for the fine-tuned control of biochemical reaction cascades [1–3].

The desire to employ this complex molecular organization in next generation chemical synthesis has led to various studies on supermolecular compartments as nanoscale "bioreactors" [4–7]. Several pathways for vesicle fusion [8–10] have been suggested. In particular, Hadorn et al. employ short single stranded DNA tags for the specific interaction of various reaction compartments [11–13]. In the European Commission funded project MATCHIT [14–17], we are working in creating
Figure 1. Example of a state of the artificial cellular matrix, where four locations \(x_1, x_2, x_3,\) and \(x_4\) hold content. All other locations are empty. Location \(x_1\) contains a number of molecules in solution (\(2A+2B+C\)), location \(x_2\) contains two chemtainers (depicted as green circles), one of them containing the molecules \(A\) and \(B\), location \(x_3\) holds a chemtainer encapsulated within another chemtainer, and location \(x_4\) holds a chemtainer with three \(A\)'s being decorated with a tag \(\sigma\) (red line attached to the green circle), with a separate complementary tag \(\tau\) in solution (free red line). Using the syntax defined in Section II A, this system state is described by the string \(x_1 : 2A + 2B + C \circ x_2 : \not\varepsilon + \not\varepsilon A + B \triangleright \not\varepsilon x_3 : \not\varepsilon 2A + \not\varepsilon C \triangleright \not\varepsilon x_4 : \tau + \not\varepsilon 3A \triangleright\).

an artificial cellular matrix that seamlessly integrates information processing and material production in much the same way as its biological counterpart: The MATCHIT framework employs DNA addressable bioreactors (termed chemtainers in the following), to mimic the topological organization of the cytoplasm. DNA tags open up for DNA computing operations akin to the “key-lock” information processing mechanism found in biological protein-protein interactions. This form of molecular information processing is governed by autonomous chemical reaction kinetics and allows for tight integration of chemical production and information processing. In particular, we here employ the DNA join-and-fork gates to implement Boolean computation [18].

 Whereas natural cells ultimately employ genomic information to regulate the resulting material production network, we envision programmable electronic control by embedding chemtainers into mechano-electronic microsystems [19, 20]. In such devices, dedicated microfluidic channels can be designed for vesicle generation [21], DNA tag insertion, tag and chemtainer trafficking [22], specific or unspecific fusion [23], vesicle rapture and encapsulation [24]. Possibly paired with real time feedback, this allows for control of chemical reaction cascades at the molecular level. This interplay of autonomous molecular computation and external electronic chemtainer manipulation enables the programmatic setup of complex reaction cascades that allow for automated chemical production of a desired target molecule from a limited set of chemical resources.

Here, we propose a formal calculus to describe system states and transitions in an abstract representation of the artificial cellular matrix. We call this the chemtainer calculus. In essence, the chemtainer calculus allows us to describe the organization and manipulation of chemical compounds down to the molecular level in possibly nested, addressable reaction compartments. Sets, or more precisely, multisets of such compartmentalized reagents are situated at locations, e.g., at positions of a microfluidic device. Several calculi for compartmentalized reaction systems have been proposed previously [25–27]. Our syntax for nested and tagged compartments follows relatively closely the one from brane calculi [27]. Those calculi offer transitions for compartment transformations, such as fusion and splitting, as well as molecular reactions. Transitions are integral components of the system state and the calculus employs a process algebra to deduce the set of possible transitions from the current state. In the chemtainer calculus, we additionally define an explicit transition system that operates on states externally. This organization better reflects the difference between the chemical state transitions and external mechano-electronic control.

The calculus was designed for the architecture described in Ref. [20]. The tool chain for compilation of high level directives of the chemtainer calculus into eventual electrode configurations
to control the mechano-electronic microfluidic hardware is presented in Ref. [28]. Yet, the general framework discussed here is not tailored toward one specific technology. For example, instead of microfluidic channel segments, locations could equally denote test tube arrays, or wells in a high throughput chip.

In this article, we first design a syntax that allows us to express the rather complex system states—or rather system arrangements—that we encounter in the artificial cellular matrix: located multisets of tagged and possibly nested chemtainers that carry cargo. Fig. 1 schematically depicts an example of a possible state. We then introduce transitions between states that model possible changes of the physical state. Some of these transitions, e.g. DNA hybridization, capture autonomous chemistry, whereas other transitions are thought to be induced by control operations of the artificial cellular matrix. We proceed by defining a simple programming language that consists of sequences of parametric instructions which induce transitions on the system state. In Section III, we discuss how the underlying instruction set of the chemtainer calculus affects the set of constructable objects, and we give our main result (Theorem 5) that a set of eight instructions is sufficient to construct any well-formed target state. Due to the fact that we use a non-deterministic semantics, this proofs only demonstrates the possibility, not the probability of
creating a certain desired state. In Section IV, we apply our framework to the area of chemical manufacturing, where we present an algorithm for synthesis of branched oligomer structures by means of controlled chemtainer fusion and DNA computing.

II. THE CHEMTAINER CALCULUS

A. System state

Objects of the calculus are molecules, address tags, and chemtainers. Chemtainers represent compact microreactors, such as vesicles, oil droplets in water or water droplets in oil, DNA nanocages, etc. They can be decorated with address tags, and can hold chemical content and even other chemtainers within them. Here, we do not distinguish between different chemtainer types, but we can imagine a type system for chemtainers to specify their physical properties. All components of the system are situated at specified, discrete locations.

Our notion of space is rather simplistic: we assume a fixed set of locations; each component of the system state is situated at a certain location; we will introduce transitions that allow collocated components to interact (while objects at different locations may not interact), and we will introduce transitions that induce transport from one location to another by means of state transitions. Note that we currently do not introduce a specific topology on the set of locations (e.g., to move from one location to another, one might need to cross a third one), but such an extension is possible.

For countable index sets \( J_L, J_M, \) and \( J_T, \) let \( L = \{ i; i \in J_L \} \) denote the set of locations, \( M = \{ m; j \in J_M \} \) some set of molecules, and \( T = \{ t; k \in J_T \} \) a set of DNA tags. We take \( T = \{ \rho, \sigma, \tau, \ldots \} \) if tags are explicitly given. Note that \( M \) might intersect with \( T \) or not.

The following context-free tree grammar \( G_S \) for system states formalizes the above verbal in-
terpretation:

\[
\begin{align*}
\text{global state} & \quad S := \emptyset \mid S \circ S \mid x_i : P \\
\text{local state} & \quad P := 0 \mid P + P \mid q^* \langle P \rangle \mid q \mid m_j \\
\text{tag} & \quad q := s \mid s^* \triangleright s^*
\end{align*}
\]

with the start symbol \( S \). Here, the vertical broken bar is a metasymbol that indicates syntactic choice. We denote the empty state by the symbol \( \emptyset \). The binary operator \( x_i : P \) denotes localization, where \( x_i \) is a location identifier; the binary operator \( S \circ S \) denotes composition of locations, while \( P + P \) denotes composition within locations; \( 0 \) denotes the empty local state; \( * \) is the Kleene operator and signifies no or arbitrarily many repetitions of its argument. We write

\[
\begin{align*}
q^* &= \circ \mid q + q^* \\
s^* &= \circ \mid s + s^*
\end{align*}
\]

with the empty tag \( \circ \) to explicitly list tag and gate sets.

Following convention, we denote chemtainers by half moon parentheses \( q^* \langle P \rangle \) that enclose the chemtainer content with address tags associated to the left parenthesis [27, 29]. Finally, the relation \( s^* \triangleright s^* \) denotes DNA gates, which will be explained later. We write \( G_s, G_p, G_q^*, \) and \( G_s^* \) for the grammars with the start symbols \( S, P, q^* \), and \( s^* \), respectively.

To generate a state with the grammar \( G_X, X \in \{ S, P, q^*, s^* \} \) being a nonterminal, one recursively applies the above production rules starting from \( X \) until the resulting state tree contains no more nonterminal symbols. The language \( L(G_X) \) is the set of all possible states that can be generated from the start symbol \( X \). In what follows, \( S, S', S'' \in L(G_S), P, P', P'', P_i \in L(G_P), q, q', q_k \in L(G_q^*), s, s', s'', s_k \in L(G_s^*) \) denote arbitrary states of the respective languages.

Informally, we interpret states of \( L(G_S) \) to signify the following: the global system state is a composition of local states, where each local state has a location identifier \( x_i \) and an associated local state description. The latter are compositions of molecules \( m_j \), gates \( q_k \), as well as chemtainers \( q^* \langle P \rangle \) with content \( P \) and gate set \( q^* \); gate sets, in turn, are compositions of gates \( q_k \) as well as individual tags \( s_k \). See Fig. 1 for an example of a global state.

In order to conform with this interpretation, we introduce the following structural congruence relation (an equivalence relation with additional axioms that guarantee substitutivity of equals in context) over \( L(G_S), L(G_P), L(G_q^*), \) and \( L(G_s^*) \):

\[
\begin{align*}
S \circ (S' \circ S'') & \equiv (S \circ S') \circ S'' & (6) \\
S \circ S' & \equiv S' \circ S & (7) \\
S \circ \emptyset & \equiv S & (8) \\
P + (P' + P'') & \equiv (P + P') + P'' & (9) \\
P + P' & \equiv P' + P & (10) \\
P + 0 & \equiv P & (11) \\
q_1 + (q_2 + q_3) & \equiv (q_1 + q_2) + q_3 & (12) \\
q_1 + q_2 & \equiv q_2 + q_1 & (13) \\
q + \circ & \equiv q & (14) \\
s_1 + (s_2 + s_3) & \equiv (s_1 + s_2) + s_3 & (15) \\
s_1 + s_2 & \equiv s_2 + s_1 & (16) \\
s + \circ & \equiv s & (17) \\
x_i : P \circ x_i : P' & \equiv x_i : P + P' & (18)
\end{align*}
\]
Thus, states that belong to the same equivalence class of \( \equiv \) represent the same physical state, even though they might be syntactically different. Equations (6) through (18) induce monoidal structures on \( (L(GS), \circ, \emptyset) \), \( (L(GP), +, 0) \), \( (L(Gq^*), +, \odot) \), and \( (L(Gs^*), +, \odot) \), where ‘:\' distributes over ‘+’, and Equations (19) through (27) guarantee that we can substitute equals in any context.

We introduce some notational shortcuts. We write \( P \% := \odot P \), \( q \% := q 0 \% \) and we introduce the notation

\[
nP := P + \ldots + P \quad \text{n times}
\]

which emphasizes the relation to mutlisets. We also allow ourselves to drop the explicit notation of empty locations by defining

\[
 x_i : 0 \equiv \emptyset.
\]

With these definitions, the example state depicted in Fig. 1 can be written as

\[
x_1 : 2A + 2B + C \circ x_2 : \emptyset + \emptyset A + B \circ x_3 : \emptyset 2A + \emptyset C \circ \circ x_4 : \tau + \sigma \emptyset 3A \emptyset.
\]

\[ \text{B. Transitions} \]

We are now ready to introduce a transition system that codifies the possible outcome of both autonomous chemical reactions, as well as externally induced operations that manipulate the system state. Autonomous reactions, in turn, are either arbitrary chemical reactions among molecules, which we refer to as application chemistry, or the working of DNA computing operations.

\[ \text{1. Application Chemistry} \]

Reactions are just transitions of the form

\[
P \longrightarrow P'
\]

where \( P = \sum_i \nu_i m_i \) and \( P' = \sum_j \mu_j m_j \) are multisets of reactants of products with stoichiometric coefficients \( \nu_i \) and \( \mu_i \).

We ensure that chemical reactions can occur among any co-located reactants that are not
separated by chemtainer walls:

\[
\begin{align*}
P & \rightarrow P' \\
P + P'' & \rightarrow P' + P'' \\
P & \rightarrow P' \\
q^* \ll P \gg & \rightarrow q^* \ll P' \gg \\
p_{i} : P & \rightarrow p_{i} : P' \\
S & \rightarrow S' \\
S \circ S'' & \rightarrow S' \circ S''
\end{align*}
\]  

(32)

We extend structural congruence to transitions by defining:

\[
\begin{align*}
P & \equiv P' \\
P' & \rightarrow P'' \\
P'' & \equiv P'''
\end{align*}
\]  

(36)

\[
\begin{align*}
S & \equiv S' \\
S' & \rightarrow S'' \\
S'' & \equiv S'''
\end{align*}
\]  

(37)

In its ground form, the chemtainer calculus does not offer any transitions of the form (31). Instead, the user of the calculus is assumed to provide a set \( \mathcal{R} \) of autonomous transitions as axioms. Note that there is no particular need to restrict \( \mathcal{M} \) to be finite. Our calculus can be applied without adaption to an infinite set of molecules, including polymers or branched structures—an example of which will be presented in Section IV. We also emphasize that we explicitly allow \( \mathcal{M} \) and \( \mathcal{T} \) to intersect. If they do, tags can occur in the reactant and product sides of chemical reactions, such that tags can be altered chemically.

2. DNA Gate Transitions

Here, DNA computation is implemented by join and fork gates [18] that irreversibly release a given set of output strands \( s^*_2 \) once they have bound a set of inputs \( s^*_1 \), written \( s^*_1 \triangleright s^*_2 \). Note that \( s^*_1 \triangleright s^*_2 \) does not physically contain the strands \( s^*_1 \). Rather it means that the gate accepts those strands as input.

If a gate is co-located with all its input tags, it can release its output tag:

\[
s^*_1 \triangleright s^*_2 + s^*_1 \rightarrow s^*_2
\]  

(38)

We have to ensure that these transitions can occur between two co-located complementary tags in any context, unless they are separated by a chemtainer boundary. This is allowed by introducing the following inference rules:

\[
\begin{align*}
q + q' & \rightarrow q'' \\
q + q^* \ll P \gg & \rightarrow q'' \ll P \gg \\
q + q' & \rightarrow q'' \\
(q + q') \ll P \gg & \rightarrow q'' \ll P \gg
\end{align*}
\]  

(39)

(40)

Examples of gate transitions are depicted in Fig. 3. Due to the inferences (32) through (37), its is guaranteed that DNA computing operations perform in any context.
3. Induced Transitions

We now introduce transitions that model the controlled manipulation of states through operations of the artificial cellular matrix. We introduce eight such operations, responsible for feeding and moving of chemtainers and tags, controlled fusion of chemtainers, encapsulation of material into chemtainers, chemtainer bursting, and flushing of content. Our exact transitions are motivated by the functionalities of the underlying microfluidics architecture [20], but they also serve as guideline how alternative transitions in other hardwares can be codified. The impact of the exact instruction set on the constructive capabilities of the resulting calculus will be discussed in Sections III.

Induced transitions are of the form

$$I : S \rightarrow S$$

where $I$ is a parameterized name. We first give the formal definition of these transitions (schematics are shown in Fig. 2) and will afterwards comment on their particular choice. In Sec. II C, we will introduce a small programming language based on these operations.

$$\text{feed}(x,m_i,\nu) : \emptyset \rightarrow x : \nu m_i \Downarrow$$ (41)

$$\text{feed-tag}(x,s,\nu) : \emptyset \rightarrow x : \nu s$$ (42)

$$\text{move}(s,x,z) : x : (s + q^*) \Downarrow P \rightarrow z : (s + q^*) \Downarrow P$$
$$x : s \rightarrow z : s$$ (43)

$$\text{tag}(x) : x : s + q^* \Downarrow P \rightarrow x : (s + q^*) \Downarrow P$$ (44)

$$\text{fuse}(x) : x : q^*_1 \Downarrow P + q^*_2 \Downarrow P' \rightarrow x : (q^*_1 + q^*_2) \Downarrow P + P'$$ (46)

$$\text{flush}(x) : x : P \rightarrow \emptyset$$ (47)

$$\text{wrap}(x,z) : x : P \rightarrow z : \Downarrow P$$ (48)

$$\text{burst}(x) : x : q^* \Downarrow P \rightarrow x : q^* \Downarrow P + P$$ (49)

Transitions (41) through (49) operate strictly on the top level of the state, meaning that we do not allow, for example, tagging or fusion of chemtainers that resides within another chemtainer. To ensure this, we simply do not provide inference rules that would allow us to derive such transitions. However, the above transitions are allowed to operate in any context through the following inference:

$$I : S \rightarrow S'$$

$$I : S \circ S \rightarrow S' \circ S$$ (50)

We now comment on the individual choice of operations and their transitions. \textbf{feed} allows one to feed chemtainers that are equipped with a certain number $\nu \in \mathbb{N}$ of molecules of a single molecular species into a location. This is the only means to provide elements of $\mathcal{M}$. Similarly,
feed_tag allows one to feed a certain number \( \nu \in \mathbb{N} \) of identical tags into a location. tag has been modeled to unspecifically attach some available tag to some container at a given location. No means are given to specify which tag or which chemtainer is transformed in this operation. The reason for this choice is that the actual linking of tags to a chemtainer involves linker molecules (biotin and streptavidin) that are common to all chemtainers and tags independent of the actual tag sequence or chemtainer type.

We provide a flush command to counteract the effect of feeding, where flushing simply removes content at a location. The operation is most easily implemented by literally flushing the material out of the system.

The move command allows one to move a specific tag, or a chemtainer decorated with a specific tag from one location to another. Notably, this allows for content separation: for example, the state \( x : \sigma \sqsubseteq A \triangleright + \tau \sqsubseteq B \triangleright \) will transition into the state \( x : \sigma \sqsubseteq A \triangleright \circ \sigma : \tau \sqsubseteq B \triangleright \) in response to the operation move\((\tau, x, y)\). In a certain interpretation, this can be understood as chemtainer docking, where the locations correspond to volumes of bulk fluid that themselves move along a microfluidic channel decorated with the complementary tag (here \( \tau \)). Assuming that both chemtainers are initially at location \( x \) which is decorated with \( \tau \), the complementarily tagged chemtainer is allowed to temporarily hybridize to the channel wall while the bulk volume—and thus the locations—physically move along the channel. Non-matching content at the logical location \( x \) will remain at that location, although its physical position has changed. Matching content, however, will be at a different location, e.g. \( y \) and can be released into the corresponding bulk.

The operations fuse and wrap are more specific to the artificial cellular matrix and enable fusion and (vesicle) encapsulation. fuse induces unspecific fusion of co-located chemtainers, leading to mixing of both chemtainer surface and content. Besides unspecific fusion, tag specific fusion of vesicles has been achieved experimentally [30, 31]. A version of the chemtainer calculus featuring such tag specific fusion is presented in [32]. wrap enables encapsulation of material into vesicles, e.g. using interface transfer: it is assumed that the material at location \( x \) resides in aqueous phase which is first immersed into an oil phase where it forms surfactant coated water droplets—providing the inner layer of the future vesicle. Next, the water droplet passes a surfactant covered oil-water interface which provides the outer layer of the vesicle membrane. If the original state at \( x \) contains chemtainers, these are equally transferred into the interior of the new chemtainer, leading to nested chemtainers [12]. Microfluidic implementations of this protocol are presented in [33]. We will later analyze the power of this transition system with and without the wrap operation.

Complementing encapsulation, we provide a burst operation that releases the content of a chemtainer. We envision that burst does not fully dissolve the chemtainer, but rather raptures the chemtainer wall temporarily, for example by means of a heat or salt shock. After bursting, the chemtainer will reconstitute itself and remain in the system, available for further processing.

In order to capture obvious implementation constraints of the physical machine, some of the operations are restricted to subsets of \( \mathcal{L} \), e.g. feed\((x)\) might require \( x \in \mathcal{L}_{\text{feed}} \subset \mathcal{L} \). It can be shown that the constructive power of the calculus is not affected by this limitation, as long as move, tag, and burst are allowed to operate on the entire set \( \mathcal{L} \). Likewise, we could constrain the move command such that the target location has to be in a certain neighborhood set of the source location in order to capture, e.g., the channel topology of microfluidic devices.

We wish to clarify one point that might be counter-intuitive. Assume, for example, the state \( x : 10 \sigma \). What will be the action to the operation move\((\sigma, x, y)\)? Intuitively, we might expect that the system transitions into the state \( y : 10 \sigma \), meaning that all instances of \( \sigma \) have been moved. However, \( x : 10 \sigma \) is structurally equivalent to \( x : 9 \sigma \circ x : \sigma \) due to the distributive relation (18). To this state we can apply the transition \( S \circ x : \sigma \rightarrow S \circ y : \sigma \) equating \( S = x : 9 \sigma \), which results into \( x : 9 \sigma \circ y : \sigma \). Thus, move, and by similarity all other transitions, will only operate on a single instance. There is a simple way out however: in order to move all ten instances in the
example above, we can simply execute the move operation ten times in sequence. Our reason for this semantics is that it allows for consistent assignment of stochastic rates in a potential stochastic semantics [34, 35].

C. Programs

In the previous section we have informally given parametric names to transitions, such as \texttt{move}(s,x,y), \texttt{tag}(x), a.s.o. We next interpret parametric names as elemental operations of a programming language that can be used to operate the artificial cellular matrix. We do this by first defining another formal language, the formal language of chemtainer programs, and then expressing the semantics of this language by means of state transitions, the latter employing the language \(L(G_S)\) of chemtainer states.

As with states, we define programs by a recursive grammar, which we initially keep as simple as possible:

\[
\pi := \epsilon \mid I; \pi. 
\]  

A program \(\pi\) can be either the empty program \(\epsilon\), or a (parametric) instruction followed by another program (the remainder), with instructions being separated by semicolons. Instructions \(I\) can be any of the formerly introduced parametric names \(\texttt{feed}(x_a,m_j,\nu), \texttt{feed}\_\texttt{tag}(x_a,q_k,\nu), \texttt{move}(s_k,x_a,x_b), \texttt{tag}(x_a), \texttt{wrap}(x_a,x_b), \texttt{fuse}(x_a), \texttt{burst}(x_a), \texttt{and flush}(x_a)\) with \(x_a,x_b \in \mathcal{L}, m_j \in \mathcal{M}, q_k \in L(G_q)\) and \(\nu \in \mathbb{N}\). Each instruction \(I\) has a set of associated transitions and we write \(I : S \rightarrow S'\) to denote any such associated transition (as was already done in the previous part).

Strictly, the concatenation operator ';' is only defined to concatenate single instructions with programs. In order to concatenate arbitrary programs \(\pi\) and \(\pi'\), we introduce an additional operator ';' for program concatenation through the following recursive definition:

\[
\epsilon;\pi = \pi \\
(I;\pi);\pi' = I;(\pi;\pi') 
\]  

Since both concatenation operators have different domains, it is clear from the context whether the concatenation refers to single instructions or programs. Therefore, we will drop the syntactic differentiation and use the symbol ';' for both operators.

A program \(\pi\) together with a system state \(S\) is expressed by the tuple \(\langle \pi, S \rangle\) and we write \(\langle \pi, S \rangle \rightarrow S'\) to denote that the program \(\pi\) can transform the state \(S\) into the state \(S'\), called a result of \(\pi\). Results are defined by the following structural operational semantics [36]:

\[
\langle \epsilon, S \rangle \rightarrow S \\
\langle \pi, S'' \rangle \rightarrow S \quad I : S' \rightarrow S'' \quad \langle I;\pi, S' \rangle \rightarrow S \\
\langle \pi, S' \rangle \rightarrow S \quad \langle I;\pi, S' \rangle \rightarrow S 
\]

In words, given a program that starts with instruction \(I\) with some associated transition from \(S'\) to \(S''\), the second rule allows us to transform the system into \(S''\), thereby reducing the original program to whatever remains after execution of \(I\). If program execution encounters an instruction with no associated transition that is applicable to the current state \(S'\) (e.g. \texttt{tag} is asked to operate
on an empty cell), the third rule allows us to skip the instruction without modifying the system state. This latter rule primarily addresses erroneous conditions and ensures that a program does not get stuck, simply because the respective molecules are not present at some point of the execution. Using non-deterministic semantics, rule (56) might even be used when rule (55) is applicable, thereby skipping instructions of the program sequence. This could be remedied with a stochastic semantics that assigns an arbitrarily small transition rate to rule (56).

We can now distinguish between autonomous and induced transitions, by extending the semantics with the rules

\[ S \rightarrow S' \quad (\pi, S') \rightarrow S'' \]

\[ (\pi, S) \rightarrow S' \quad S' \rightarrow S'' \]

where \( S \rightarrow S' \) and \( S' \rightarrow S'' \) is an autonomous transition inferred through (35). These rules allow autonomous transitions to occur at any time during program execution without affecting the program.

We extend structural congruence to program application using the inference rule

\[ S \equiv S'' \quad (\pi, S) \rightarrow S' \quad S' \equiv S'' \]

\[ (\pi, S') \rightarrow S'' \]

The following two lemmas, proven in the appendix, allow for uncomplicated composition of instructions.

**Lemma 1** Program application is not context sensitive: if the application of program \( \pi \) on the initial state \( S \) can lead to the result \( S' \), the application on the composition \( S \circ \tilde{S} \) can lead to the result \( S' \circ \tilde{S} \), containing \( S' \) as a substate. \( \tilde{S} \) is an invariant of \( \pi \):

\[ (\pi, S) \rightarrow S' \quad \Rightarrow \quad (\pi, S \circ \tilde{S}) \rightarrow S' \circ \tilde{S}. \]  

**Lemma 2** Programs can be concatenated and the start configuration of the second program will be the result of the first program:

\[ (\pi', S) \rightarrow S' \quad \land \quad (\pi'', S') \rightarrow S'' \quad \Rightarrow \quad (\pi' \pi'', S) \rightarrow S'' \]

### D. Extension: control flow directives and parallel execution

We could easily define control flow directives known from standard programming languages, such as branched execution and loops. This is particularly interesting in the context of feedback control. Let

\[ p : S \rightarrow \{\text{True, False}\} \]

be a conditional over the system state. For example, \( f \) could measure whether a fluorescent signal at a certain location exceeds a certain threshold. We could then extend our grammar to

\[ \pi := \epsilon \mid I ; \pi \mid \text{while } f(S) \text{ do } \pi ; \text{ endwhile} \]
to allow for conditional loops, where the semantics of the \textbf{while} statement are given by
\[
\langle \pi, S \rangle \rightarrow S' \quad \langle \text{while } p \text{ do } \pi; \text{ endwhile}, S' \rangle \rightarrow S'' \quad p = \text{True} \quad (64)
\]
\[
\langle \text{while } p \text{ do } \pi; \text{ endwhile}, S \rangle \rightarrow S'' \quad p = \text{False} \quad (65)
\]
As evidenced with these definitions, there is nothing particular about control flow directives in the chemtainer calculus compared to other imperative languages. In order to introduce more elaborate control flows, the reader can simply apply standard textbook procedures.

Similarly, it is straightforward to define the semantics for a language of communicating sequential processes \cite{37, 38} that parallelizes the production process:
\[
\langle \pi', S' \rangle \rightarrow \bar{S}' \quad \langle \pi'', S'' \rangle \rightarrow \bar{S}'' \quad (66)
\]

E. Example

To make the above definitions more familiar, we employ the chemtainer calculus in order to decorate chemtainers with addresses and use those to mix the contents of two chemtainers.

The left column shows which parametric instruction is executed and the right column shows the effect of its associated transition. Read from top to bottom, the left column therefore simply gives a program that can construct some desired state denoted at the right.

\[
\emptyset
\]
\[
\text{feed}(x, P) \quad x : \emptyset P \uparrow
\]
\[
\text{feed}_\text{tag}(y, \sigma) \quad x : \emptyset P \uparrow \circ y : \sigma \quad (67)
\]
\[
\text{move}(\sigma, y, x) \quad x : \sigma + \emptyset P \uparrow
\]
\[
\text{tag}(x) \quad x : \sigma \emptyset P \uparrow
\]

To use this sequence later on in programs, we define it as the parametric macro \textbf{resource}(x, \sigma, P).

Now, we use this macro to create two chemtainers and fuse them:

\[
\emptyset
\]
\[
\text{resource}(x, \sigma, P) \quad x : \sigma \emptyset P \uparrow
\]
\[
\text{resource}(y, \rho, Q) \quad x : \sigma \emptyset P \uparrow \circ y : \rho \emptyset Q \uparrow \quad (68)
\]
\[
\text{move}(\rho, y, x) \quad x : \sigma \emptyset P \uparrow + \rho \emptyset Q \uparrow
\]
\[
\text{fuse}(x) \quad x : (\sigma + \rho) \emptyset P + \emptyset Q \uparrow
\]

Keep in mind that the right column only shows a possible response of the system state to the program on the left, not the necessary response. In general, because of the ambiguity of transitions, the system could have transitioned into alternative states in response to the same program.

III. INSTRUCTION SETS AND THEIR CONSTRUCTIVE POWER

The instructions given in Eqs. (41) through (49) serve as examples of possible operations, rather than the final specification of an artificial cellular matrix. Potential real-world implementations
might involve only a subset of the operations mentioned, or might enable other means of chemtainer manipulation. It then becomes interesting to determine whether different instruction sets are equivalent in what they can construct or whether the set of constructable states differs.

In order to approach the problem formally, we define the constructive closure $\omega^+(\Pi_x) \subset L(G_S)$ of the language $\Pi_x$ as the set of states that can be constructed from the empty state through some program $\pi \in \Pi_x$. Formally:

$$\omega^+(\Pi_x) := \{ S \in L(G_S) : \exists \pi \in \Pi_x \land \langle \pi, \emptyset \rangle \rightarrow S \}$$  \hfill (69)

Similarly, we define the reset closure $\omega^-(\Pi_x) \subset L(G_S)$ as the set of states that can be transformed back into the empty set. Formally:

$$\omega^-(\Pi_x) := \{ S \in \omega^+(\Pi_x) : \exists \pi \in \Pi_x \land \langle \pi, S \rangle \rightarrow \emptyset \}$$  \hfill (70)

Of primary interest is of course the case $\omega^- = \omega^+$ where the machine can be reset from any state. In this case, one can formally transform any initial state into any target state, thereby programming sequences of states: if $\langle \pi, S \rangle \rightarrow \emptyset$ and $\langle \pi', \emptyset \rangle \rightarrow S'$, the concatenation $\pi; \pi'$ will transform $S$ into $S'$: $\langle \pi; \pi', S \rangle \rightarrow S'$. This result, however, is mostly of theoretical interest, as an interim reset of the machine might not be desired when programming such state sequences. In a practical application, one would define a distance measure between states and ensure that the transition path length of programs is minimized, thereby preferring for example a single move instruction over a sequence of flush;feed_tag;move instructions with equal outcome.

We consider the following three incremental instruction sets:

$$I_{\text{min}} = \{ \text{feed, feed\_tag, move, tag, fuse, flush} \}$$  \hfill (71)
$$I_{\text{wrap}} = I_{\text{min}} \cup \{ \text{wrap} \}$$  \hfill (72)
$$I_{\text{burst}} = I_{\text{wrap}} \cup \{ \text{burst} \}$$  \hfill (73)

Let $\Pi_{\text{min}}$, $\Pi_{\text{wrap}}$, and $\Pi_{\text{burst}}$ be the sets of programs over the respective instruction set. We will show the following relations between the corresponding constructive and reset closures:

$$\omega^+(\Pi_{\text{min}}) \subset \omega^+(\Pi_{\text{wrap}}) \subset \omega^+(\Pi_{\text{burst}}) = L(G_S)$$
$$\omega^-(\Pi_{\text{min}}) \subset \omega^-(\Pi_{\text{wrap}}) \subset \omega^-(\Pi_{\text{burst}}) \subseteq L(G_S).$$  \hfill (74)

If and only if all locations are flushable, i.e. if $L_{\text{flush}} = L$, the right-most chain of relations simplifies to

$$\omega^+(\Pi_{\text{burst}}) = \omega^-(\Pi_{\text{burst}}) = L(G_S).$$  \hfill (75)

Diagram (74) can be decomposed into several theorems, for which we need to define the nesting level $d : L(G_S) \rightarrow \mathbb{N}$ of a state $S$ or local state $P$ as the following:

$$d(\emptyset) = 0$$  \hfill (76)
$$d(0) = 0$$  \hfill (77)
$$d(m_j) = 0$$  \hfill (78)
$$d(q^*) = 1$$  \hfill (79)

$$d(x_i : P) = d(P)$$  \hfill (80)
$$d(S \circ S') = \max\{d(S), d(S')\}$$  \hfill (81)
$$d(P + P') = \max\{d(P), d(P')\}$$  \hfill (82)
$$d(q \not\in P \not\in) = 1 + d(P)$$  \hfill (83)

With this definition, we can state the relations of diagram (74) more precisely:

**Theorem 3** The constructive closure of $\Pi_{\text{min}}$ is the set of states that do not contain nested chemtainers nor free floating molecules other than tags:

$$\omega^+(\Pi_{\text{min}}) = \{ S \in L(G_S) : d(S) \leq 1 \land S \neq S' \circ x_i : m_j \text{ for } m_j \in M \setminus T \}$$  \hfill (84)
Theorem 4 The constructive closure of $\Pi_{\text{wrap}}$ is the set of states that do not contain free floating molecules other than tags:

$$\omega^+(\Pi_{\text{wrap}}) = \{ S \in L(G_5) : S \not\equiv S' \circ x_i : m_j \text{ for } m_j \in \mathcal{M} \setminus \mathcal{T} \}$$  \hspace{1cm} (85)

Theorem 5 The constructive closure of $\Pi_{\text{burst}}$ is the entire set of states:

$$\omega^+(\Pi_{\text{burst}}) = L(G_5)$$  \hspace{1cm} (86)

Theorem 6 The reset closure of all instruction sets is limited to states that do not contain free floating molecules other than tags in non-flushable locations:

$$\omega^-(\Pi_x) \subset \{ S \in L(G_5) : S \not\equiv S' \circ x_i : m_j \text{ for } x_i \not\in L_{\text{flush}} \cup L_{\text{wrap.in.}}, m_j \in \mathcal{M} \setminus \mathcal{T} \} \quad \forall x \in \{ \text{min, wrap, burst} \}$$  \hspace{1cm} (87)

Proofs for all theorems are given in the appendix. Note that these proofs are constructive: for any given target structure $S \in \omega^+(\Pi_x)$ we can automatically generate a program that will construct $S$. This observation forms the core of a compiler that is able to generate a sequence of transitions for building a given target structure.

The above theorems assume that the entire set of molecules $\mathcal{M}$ can be fed as resources. We now address the case where only a subset $\mathcal{M}_R \subset \mathcal{M}$ of compounds can be directly provided by feed and where chemical reactions are employed to produce compounds outside $\mathcal{M}_R$.

For some reaction $\nu_1 m_1 + \nu_2 m_2 \rightarrow \nu_3 m_3 + \nu_4 m_4$ with $m_1, m_2 \in \mathcal{M}_R$, we can employ the program

$$\emptyset$$

$$\text{resource}(x, \psi, \nu_1 m_1) \quad x : \psi \underline{\nu_1 m_1} \triangleright$$

$$\text{resource}(y, \rho, \nu_2 m_2) \quad x : \psi \underline{\nu_1 m_1} \triangleright \circ y : \rho \underline{\nu_2 m_2} \triangleright$$

$$\text{move}(\rho, y, x) \quad x : \psi \underline{\nu_1 m_1} \triangleright + \rho \underline{\nu_2 m_2} \triangleright$$

$$\text{fuse}(x) \quad x : (\psi + \rho) \underline{\nu_1 m_1 + \nu_2 m_2} \triangleright$$

$$x : (\psi + \rho) \underline{\nu_3 m_3 + \nu_4 m_4} \triangleright$$

(88)

To initiate the chemical production of $P' = \nu_3 m_3 + \nu_4 m_4$, where $m_3, m_4$ are not necessarily elements of $\mathcal{M}_R$. Applying this procedure repeatedly allows us to obtain successively bigger subsets of $\mathcal{M}$. However, without means for content separation, the set of producible states will be constrained by the stoichiometries of the reactions. If the artificial cellular matrix would offer content separation, e.g. by means of electrophoresis [39], we could encode this with the induced transition

$$\text{separate}(x, y) : \quad x : (q + q') \underline{\nu_3 m_3 + \nu_4 m_4} \triangleright \rightarrow y : q \underline{P} \triangleright + q' \underline{P'} \triangleright.$$  \hspace{1cm} (89)

where the exact partitioning of chmtainer content into $P$ and $P'$ would depend on its electrophoretic mobility. Continuing the above program, we could now derive

$$x : (\psi + \rho) \underline{\nu_3 m_3 + \nu_4 m_4} \triangleright$$

$$\text{separate}(x, y) \quad y : \psi \underline{\nu_3 m_3} \triangleright + \rho \underline{\nu_4 m_4} \triangleright$$  \hspace{1cm} (90)

The tagged chmtainers are ready to be used as input for subsequent reactions. This recovers the constructive power of the chmtainer calculus, provided that all elements of $\mathcal{M}$ can be constructed by some chain of reactions. However, in order to maintain a well-defined mapping from DNA tags to chmtainer content, the separate operation would need to assert the correct redistribution of surface tags along with the content separation such that products can be unambiguously identified by their tags.
IV. PROGRAMMABLE REACTION CASCADES

Here, we demonstrate how our calculus (even with the minimal instruction set $I_{\text{min}}$) can be used to integrate chemical production with molecular computation in order to control programmable reaction cascades. This example is motivated by and closely mimicks the synthesis of oligosaccharides in the Golgi apparatus [1, 2]. Oligosaccharides are branched, heterogeneous polymers composed of typically five to ten individual sugar monomers such as mannose, galactose, and glucose. This diverse class of biochemicals is involved in various physiological processes pertaining e.g. to cell-cell recognition, intra- and intercellular trafficking, and metabolic modulation [40]. However, their combinatorial richness poses a challenge for chemical oligosaccharide synthesis based on conventional chemical manufacturing techniques [41].

Biological oligosaccharide synthesis proceeds in the Golgi apparatus by adding individual monomers one unit at a time to specific binding sites of the growing oligomer. Monomers are attached to enzymes that promote the specific binding reactions:

$$P_i + E_j^* M_i \rightarrow P_{i+1} + E_j.$$  \hfill (91)

Here, $P_i$ denotes an intermediate oligomer, to which monomer $M_i$ is added at the site $j$. If binding sites are unique, a given enzyme-monomer complex $E_j^* M_i$ contains all the information required to build the specific product $P_{i+1}$ from $P_i$. Prior to these polymerization steps, monomers have to be attached to the respective enzymes:

$$M_i + E_j^* \rightarrow E_j^* M_i.$$  \hfill (92)

Chemical one-pot synthesis of a given target structure is challenging, because repetition of bindings sites in the oligomer structure can lead to undesired side products. The number of potential side products can be controlled, however, by forcing some reaction steps to occur sequentially while others are allowed to proceed in parallel [42]. Weyland et al. [43] present an algorithm that identifies such optimal reaction cascades. For example, assume that the structure shown in Fig. 4 can be produced with the reaction cascade

$$\text{Gal} + E_{\text{Gal-4}}^* \text{Gal} + E_{\text{Gal-4}}^* \text{Man} \rightarrow P_0 + 2E_{\text{Gal-4}}$$  \hfill (93)

$$E_{\text{Man-6}}^* \text{Man} + E_{\text{Man-6}}^* \text{Glc} + E_{\text{Man-3}}^* \text{Glc} \rightarrow P_1 E_{\text{Man-6}}^* + E_{\text{Man-6}} + E_{\text{Man-3}}$$  \hfill (94)

$$E_{\text{Man-3}}^* \text{Man} + E_{\text{Man-2}}^* \text{Gal} + E_{\text{Gal-4}}^* \text{Gal} \rightarrow P_2 E_{\text{Man-3}}^* + E_{\text{Man-2}} + E_{\text{Gal-4}}$$  \hfill (95)

$$P_0 + E_{\text{Man-6}}^* P_1 + E_{\text{Man-3}}^* P_2 \rightarrow P_3 + E_{\text{Man-6}} + E_{\text{Man-3}}$$  \hfill (96)

where it has to be ensured that reactions (93) through (95) occur in isolation and prior to reaction (96).
Figure 6. Steps in the programmed synthesis of compound $P_0$ from Fig. 4. a) Initial system state, where resource chemtainers are provided at the storage location $x_S$ and the DNA gate $(\alpha + \beta + \gamma) \triangleright \kappa$ is provided at the operation location $x_0$. b) System state after chemtainers labelled $\alpha$, $\beta$, and $\gamma$ are moved to $x_0$. c) System state after fusion has been initiated in $x_0$ and the tag has been bound. d) Chemical reactions and gate transitions occur autonomously at $x_0$. e) The newly created chemtainer labelled $\kappa$ with product $P_0$ is transferred back to $x_S$.

Our strategy here is to employ the chemtainers in order to control the encounter of reactants and hence the order of reactions by encapsulating reactants within chemtainers. Fusion of chemtainers co-locates desired reactants and triggers their polymerization. Simultaneously, DNA gate computation on the chemtainer surface in order will reflect the change of chemtainer content after reaction. The same DNA computation can ensure that reactions are started if and only if other reactions have been performed beforehand. We use one location $x_0$ for processing and one location $x_S$ for storing intermediate products.

We start by preparing chemtainers with enzyme complexes using the program given in Equation (88). For example,

$$\text{resource}(x_0, \sigma, \text{Gal}); \text{resource}(x_0, \rho, E_{\text{Gal-4}}^*); \text{feed_tag}(x_0, (\sigma + \rho) \triangleright \beta); \text{fuse}(x_0); \text{move}(\beta, x_0, x_S)$$

(97)

creates the state $x_S : \beta \ll E_{\text{Gal-4}}^* \text{Gal} \gg$. See Fig. 5 for a graphical representation of these steps. We
use this algorithm repeatedly to set up the machine in the following state:

\[ x_S : \alpha \langle \text{Gal} \rangle + \beta \langle \text{E}_{\text{Gal}-4}\text{Gal} \rangle + \gamma \langle \text{E}_{\text{Gal}-4}\text{Man} \rangle + \delta \langle \text{E}_{\text{Man}-6}\text{Man} \rangle + \epsilon \langle \text{E}_{\text{Man}-6}\text{Glc} \rangle + \zeta \langle \text{E}_{\text{Man}-3}\text{Glc} \rangle + \eta \langle \text{E}_{\text{Man}-3}\text{Man} \rangle + \theta \langle \text{E}_{\text{Man}-2}\text{Gal} \rangle + \iota \langle \text{E}_{\text{Gal}-4}\text{Glc} \rangle \]  

(98)

With this mapping from chemical compounds to DNA tags, we translate the reaction cascade (93) and (96) into a set of DNA gates:

\[
\begin{align*}
(\alpha + \beta + \gamma) &\triangleright \kappa \\
(\delta + \epsilon + \zeta) &\triangleright \lambda \\
(\eta + \theta + \iota) &\triangleright \mu \\
(\kappa + \lambda + \mu) &\triangleright \omega
\end{align*}
\]

(99)(100)(101)(102)

To carry out reaction (93), we feed the gate (99) at location \(x_0\). We then progressively move chemtainers from \(x_A, x_B, x_C\) to \(x_0\), initiate their fusion, and transport the intermediate product \(P_0\) back to \(x_S\) (see Fig. 6 for a schematic representation):

\[
\begin{align*}
x_0 : (\alpha + \beta + \gamma) &\triangleright \kappa \circ x_S : \alpha \langle \text{Gal} \rangle + \beta \langle \text{E}_{\text{Gal}-4}\text{Gal} \rangle + \gamma \langle \text{E}_{\text{Gal}-4}\text{Man} \rangle \\
\text{move}(\alpha, x_S, x_0) &\quad x_0 : (\alpha + \beta + \gamma) \triangleright \kappa + \alpha \langle \text{Gal} \rangle \circ x_S : \beta \langle \text{E}_{\text{Gal}-4}\text{Gal} \rangle + \gamma \langle \text{E}_{\text{Gal}-4}\text{Man} \rangle \\
\text{move}(\beta, x_S, x_0) &\quad x_0 : (\alpha + \beta + \gamma) \triangleright \kappa + \alpha \langle \text{Gal} \rangle \circ x_S : \beta \langle \text{E}_{\text{Gal}-4}\text{Gal} \rangle + \gamma \langle \text{E}_{\text{Gal}-4}\text{Man} \rangle \\
\text{move}(\gamma, x_S, x_0) &\quad x_0 : (\alpha + \beta + \gamma) \triangleright \kappa + \alpha \langle \text{Gal} \rangle \circ x_S : \beta \langle \text{E}_{\text{Gal}-4}\text{Gal} \rangle + \gamma \langle \text{E}_{\text{Gal}-4}\text{Man} \rangle \\
\text{tag}(x_0) &\quad x_0 : (\alpha + (\alpha + \beta + \gamma) \triangleright \kappa) \langle \text{Gal} \rangle + \beta \langle \text{E}_{\text{Gal}-4}\text{Gal} \rangle + \gamma \langle \text{E}_{\text{Gal}-4}\text{Man} \rangle \\
\text{fuse}(x_0) &\quad x_0 : (\alpha + \beta + (\alpha + \beta + \gamma) \triangleright \kappa) \langle \text{Gal} + \text{E}_{\text{Gal}-4}\text{Gal} \rangle + \gamma \langle \text{E}_{\text{Gal}-4}\text{Man} \rangle \\
\text{fuse}(x_0) &\quad x_0 : (\alpha + \beta + \gamma (\alpha + \beta + \gamma) \triangleright \kappa) \langle \text{Gal} + \text{E}_{\text{Gal}-4}\text{Gal} + \text{E}_{\text{Gal}-4}\text{Man} \rangle \\
\text{flush}(x_0) &\quad x_0 : \kappa \langle P_0 + 2 \text{E}_{\text{Gal}-4} \rangle \\
\text{move}(\kappa, x_0, x_S) &\quad x_S : \kappa \langle P_0 + 2 \text{E}_{\text{Gal}-4} \rangle \\
\text{flush}(x_0) &\quad x_S : \kappa \langle P_0 + 2 \text{E}_{\text{Gal}-4} \rangle 
\end{align*}
\]

(103)

The flush instruction cleans the processing location in the case that the execution of an instruction got skipped because of rule (56). This prevents the subsequent fuse operations from operating on chemtainers that where intended to be kept separate.

Repeating the above algorithm with gate (100) and tags \(\delta, \epsilon, \zeta\) as well as gate (101) and tags \(\eta, \theta, \iota\) produces chemtainers with the intermediate products \(P_0, P_1\), and \(P_2\) taged with \(\kappa, \lambda, \mu\), respectively. Eventually, the above algorithm with gate (102) and tags \(\kappa, \lambda, \mu\) produces the chemtainer \(\omega \langle P_3 + \ldots \rangle\) which can readily be moved to some output location.

Note that the above program operates over a finte set of resources, tags, and locations, in order to build a compound from a potentially unlimited universe of target molecules, and the mapping between tags and compounds is established only within the controlling program. This demonstrates again the universality of the programmatic synthesis approach.

V. DISCUSSION

We have formally introduced the chemtainer calculus which is capable of capturing system states and transitions of an artificial cytoplasm. The chemtainer calculus allows us to describe the organization and manipulation of chemical compounds in possibly nested, addressable bioreactors.
Elemental operations of a simple programming language are proposed, that allow for the program-
matic control of chemtainer transitions. A constructive proof is given that a programming language
based on eight elemental instructions is capable of constructing any possible system configuration
that can be expressed in the grammar of the chemtainer calculus. This proof can serve as a core
component of a compiler for automated program inference. We have presented how our machin-
ery can be used to compile and execute complex chemical synthesis protocols and have given an
example from oligosaccharide synthesis which still poses a challenging task for conventional chemi-
cal manufacturing techniques. This framework for programmable chemical synthesis demonstrates
how molecular computing and chemical production can be integrated in much the same way as in
biological systems, for example in the Golgi apparatus.

One might object that chemical synthesis in chemtainers works the same as sequential mixings
of reaction solutions in test tubes, which is a conventional methodology. What then is the added
values of chemtainers? Firstly, we point out that microfluidics in general (even without employ-
ning chemtainers), allows for the programmed set-up of reaction cascades in small volumes. This
gives a general advantage over setting up reactions in test tubes either manually or even by liquid
handling robots: as pointed out by Füchslin et al. [42], small spatial dimensions offer the possi-
bility for rapid transport of intermediate compounds among different reaction environments. The
accompanying reduction in processing times can be of crucial advantage in synthesis steps where
intermediate compounds are only stable for short periods of time. Consequently, compounds not
viable in present processing may become interesting candidates for e.g. catalysis in miniaturized
systems. Employing vesicles as reaction compartments allows for the use of even smaller reaction
volumes on the femtoliter scale. Secondly, DNA addressable reaction compartments can sometimes
avoid the need of distinct reaction environments, in the sense that all reactants can be provided
simultaneously in the same “pot”, but individual reactions are controlled by DNA-mediated vesicle
fusion. This “automated assembly” is programmed not in the microfluidic control, but in the DNA
tagging of vesicle. Although this has been not exemplified in this paper, we can envision synthesis
pathways where DNA tags participate in the reaction cascades, e.g., as aptamers. Folding the DNA
tag set with the set of chemicals would allow chemical reactions to directly “report” about their
progress, such that, e.g., a DNA tag operation is only triggered after a chemical compound has
been consumed. DNA tagged reaction compartments further allow for the extraction and recovery
of unreacted compounds by their unaltered DNA tag, and the extraction of reaction products by
their altered DNA tag. Thirdly, encapsulation of compounds into vesicular reaction compartments
offers additional advantages for chemical manufacturing in microfluidics. Compounds do not con-
taminate microfluidic channels as they are physically contained in vesicles. Vesicles can expose a
unified physical “interface” (in terms of friction, buoyancy, charge density, etc.) for microfluidic
control independent of their content. By altering the composition of membrane molecules, these
properties can be altered vastly independently from the vesicle content.

In all our derivations, we have made ample use of non-deterministic semantics: we have proven
that there exists a program that is able to induce desired transitions, and is thus able to construct
a desired state. We have not taken into account the likelihood of those transitions—especially
with respect to possible but undesired side reactions. As this is an important issue in the area
of programming chemistry, we have carefully designed our transition system with an extension
toward stochastic semantics in mind [34]. Application of these known techniques to the chemtainer
calculus would be subject of future work.
ACKNOWLEDGMENTS

The research leading to these results has received funding from the Danish National Research Foundation through the Center for Fundamental Living technologies (FLinT) and from the European Community’s Seventh Framework Programme (FP7/2007-2013) under grant agreements no.249032 (MATCHIT) and no.318671 (MICReagents). Steen Rasmussen, Andrew Phillips, Rasmus Petersen, Rudolf Füchslin, Ehud Shapiro, and members of the MatchIT consortium are acknowledged for helpful discussions.

Appendix A: Grammar and equivalence relations

States:

\[
\begin{align*}
S & := \emptyset \mid S \circ S \mid x_i : P \\
P & := 0 \mid P + P \mid q^* \ll P \gg \\
q & := s \mid s^* \triangleright s^*
\end{align*}
\]

(1)

\[
\begin{align*}
S \circ (S' \circ S'') & \equiv (S \circ S') \circ S'' \\
S \circ S' & \equiv S' \circ S \\
P + (P' + P'') & \equiv (P + P') + P'' \\
P + P' & \equiv P' + P \\
q_1 + (q_2 + q_3) & \equiv (q_1 + q_2) + q_3 \\
q_1 + q_2 & \equiv q_2 + q_1 \\
q + q & \equiv q \\
s_1 + (s_2 + s_3) & \equiv (s_1 + s_2) + s_3 \\
s_1 + s_2 & \equiv s_2 + s_1 \\
s + o & \equiv s \\
x_i : P \circ x_i : P' & \equiv x_i : P + P'
\end{align*}
\]

(2)

(3)

(4)

(5)

(6)

(7)

(8)

(9)

(10)

(11)

(12)

(13)

(14)

(15)

(16)

(17)

(18)

(19)

(20)

(21)

(22)

(23)

(24)

(25)

(26)

(27)

(28)

(29)

(30)

(31)

(32)

(33)

(34)

(35)

(36)

(37)

(38)

(39)

(40)

(41)

(42)

(43)

(44)

(45)

(46)

(47)

(48)

(49)

(50)

(51)

(52)

(53)

(54)

(55)

(56)

(57)

(58)

(59)
Appendix B: Proofs

1. Proof of Lemma 1

We prove the statement:

\[ \langle \pi, S \rangle \rightarrow S' \implies \langle \pi, S \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} . \]  \hspace{1cm} (60)

Given that there exists an inference tree that derives the clause \( \langle \pi, S \rangle \rightarrow S' \), we show that there exists a parallel inference tree that derives \( \langle \pi, S \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} \). The statement is proven by induction over the structure of these derivation.

**Proof:**

A. Assume that \( \langle \pi, S \rangle \rightarrow S' \) was derived by (54) in the last step. Hence, \( \pi = \epsilon \) and \( S' = S \). It trivially follows that \( \langle \epsilon, S \circ \bar{S} \rangle \rightarrow S \circ \bar{S} \), as axiom (54) holds for any state.

B. Assume that \( \langle \pi, S \rangle \rightarrow S' \) was derived by (55) in the last step. Hence, \( \pi = I; \pi' \).

\[
\begin{align*}
\langle \pi', S'' \rangle & \rightarrow S' & \ \ \ \ \ \ I : S \rightarrow S'' \\
\langle I; \pi', S \rangle & \rightarrow S' & \ \ \ \ \ \ (55)
\end{align*}
\]

We have as induction hypothesis that \( \langle \pi', S'' \rangle \rightarrow S' \implies \langle \pi', S'' \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} \). We can then use rule (55) to derive:

\[
\begin{align*}
\langle \pi', S'' \circ \bar{S} \rangle & \rightarrow S' \circ \bar{S} & \ \ \ \ \ \ I : \bar{S} \circ S \rightarrow S'' \circ S \\
\langle I; \pi', S \circ \bar{S} \rangle & \rightarrow S' \circ \bar{S} & \ \ \ \ \ \ (50)
\end{align*}
\]

Hence, if \( \langle \pi', S'' \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} \) is derivable, \( \langle I; \pi', S \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} \) is derivable by induction.

C. Assume that \( \langle \pi, S \rangle \rightarrow S' \) was derived by (56) in the last step. Hence, \( \pi = I; \pi' \).

\[
\begin{align*}
\langle \pi', S \rangle & \rightarrow S' & \ \ \ \ \ \ (56)
\end{align*}
\]

We have as induction hypothesis that \( \langle \pi', S \rangle \rightarrow S' \implies \langle \pi', S \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} \). Rule (56) holds for any state, thus we can derive, with \( S \) substituted by \( S \circ \bar{S} \):

\[
\begin{align*}
\langle \pi', S \circ \bar{S} \rangle & \rightarrow S' \circ \bar{S} & \ \ \ \ \ \ (56)
\end{align*}
\]

Hence, if \( \langle \pi', S \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} \) is derivable, \( \langle I; \pi', S \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} \) is derivable by induction.

D. Assume that \( \langle \pi, S \rangle \rightarrow S' \) was derived by (57) in the last step.

\[
\begin{align*}
S & \rightarrow S'' & \ \ \ \ \ \ (57)
\end{align*}
\]

We have as induction hypothesis that \( \langle \pi, S'' \rangle \rightarrow S' \implies \langle \pi, S'' \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} \). We can then use rule (55) to derive:

\[
\begin{align*}
\langle \pi, S'' \circ \bar{S} \rangle & \rightarrow S' \circ \bar{S} & \ \ \ \ \ \ (55)
\end{align*}
\]

Hence, if \( \langle \pi, S'' \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} \) is derivable, \( \langle \pi, S \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} \) is derivable by induction.
E. Assume that \( \langle \pi, S \rangle \rightarrow S' \) was derived by (58) in the last step.

\[
\frac{\langle \pi, S \rangle \rightarrow S''}{\langle \pi, S \rangle \rightarrow S'} \quad (58)
\]

We have as induction hypothesis that \( \langle \pi, S \rangle \rightarrow S'' \implies \langle \pi, S \circ \bar{S} \rangle \rightarrow S'' \circ \bar{S} \). Again, we can use rule (35) to derive:

\[
\frac{S'' \rightarrow S'}{\langle \pi, S \circ \bar{S} \rangle \rightarrow \langle \pi, S \circ \bar{S} \rangle \circ \bar{S}} \quad (35)
\]

\[
\frac{S'' \rightarrow \bar{S}}{\langle \pi, S \circ \bar{S} \rangle \rightarrow \langle \pi, S \circ \bar{S} \rangle \circ \bar{S}} \quad (58)
\]

Hence, if \( \langle \pi, S \circ \bar{S} \rangle \rightarrow S'' \circ \bar{S} \) is derivable, \( \langle \pi, S \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} \) is derivable by induction.

F. Finally, assume that \( \langle \pi, S \rangle \rightarrow S' \) was derived by (59) in the last step.

\[
\frac{S \equiv S''}{\langle \pi, S'' \rangle \rightarrow \langle \pi, S'' \rangle \circ \bar{S} \circ \bar{S}} \quad (59)
\]

We have as induction hypothesis that \( \langle \pi, S'' \rangle \rightarrow S'' \implies \langle \pi, S'' \circ \bar{S} \rangle \rightarrow S'' \circ \bar{S} \). Thus, we can infer:

\[
\frac{S \equiv S''}{S \circ S \equiv S'' \circ \bar{S} \circ \bar{S}} \quad (19)
\]

\[
\frac{\langle \pi, S'' \circ \bar{S} \rangle \rightarrow S'' \circ \bar{S}}{\langle \pi, S'' \circ \bar{S} \rangle \rightarrow \langle \pi, S'' \circ \bar{S} \rangle \circ \bar{S} \circ \bar{S}} \quad (19)
\]

\[
\frac{S'' \equiv S'}{S'' \circ \bar{S} \equiv S'} \quad (59)
\]

Hence, if \( \langle \pi, S'' \circ \bar{S} \rangle \rightarrow S'' \circ \bar{S} \) is derivable, then \( \langle \pi, S \circ \bar{S} \rangle \rightarrow S' \circ \bar{S} \) is derivable by induction.

2. Proof of Lemma 2

We prove the statement:

\[\langle \pi', S \rangle \rightarrow S' \land \langle \pi'', S' \rangle \rightarrow S'' \implies \langle \pi'; \pi'', S' \rangle \rightarrow S''\] \hspace{1cm} (61)

**Proof:** Again, the proof is performed inductively over the structure of the derivation, in particular the derivation of the clause \( \langle \pi', S \rangle \rightarrow S' \) for arbitrary derivations of the clause \( \langle \pi'', S' \rangle \rightarrow S'' \).

A. Assume that \( \langle \pi', S \rangle \rightarrow S' \) was derived by (54) in the last step. Hence, \( \pi' = \epsilon \) and \( S' = \bar{S} \).

\[
\frac{\epsilon, S}{\langle \epsilon, S \rangle \rightarrow S} \quad (54)
\]

For any \( \pi'' \) with \( \langle \pi'', S' \rangle \rightarrow S'' \), we then have \( \langle \pi'', S' \rangle \rightarrow S'' = \langle \epsilon; \pi'', S' \rangle \rightarrow S'' \) because \( \epsilon; \pi'' = \pi'' \) by definition of ‘;’ among programs.

B. Assume that \( \langle \pi', S \rangle \rightarrow S' \) was derived by (55) in the last step. Hence, \( \pi' = I; \pi'' \). As induction hypothesis we have that \( \langle \pi''; \pi', S' \rangle \rightarrow S'' \) is derivable for any \( \pi'' \). We can then infer that

\[
\frac{\langle \pi''; \pi', S' \rangle \rightarrow S''}{I; S \rightarrow S'} \quad (55)
\]

Hence, if \( \langle \pi''; \pi', S' \rangle \rightarrow S'' \) is derivable, we know by induction that \( \langle \pi'; \pi'', S' \rangle \rightarrow S'' \) is derivable.
C. Assume that \( \langle \pi', S \rangle \rightarrow S' \) was derived by (56) in the last step. Hence, \( \pi' = I; \pi''' \). We have as induction hypothesis that \( \langle \pi'''; \pi'', S' \rangle \rightarrow S'' \) is derivable for any \( \pi'' \). We can then infer that

\[
\frac{\langle \pi''', \pi'', S \rangle \rightarrow S''}{I; \pi'''; \pi'', S \rightarrow S''} \quad (56)
\]

Hence, if \( \langle \pi'''; \pi'', S \rangle \rightarrow S'' \) is derivable, we know by induction that \( \langle \pi'; \pi'', S \rangle \rightarrow S'' \) is derivable.

D. Assume that \( \langle \pi', S \rangle \rightarrow S' \) was derived by (57) in the last step. As induction hypothesis we have that \( \langle \pi'; \pi'', S''' \rangle \rightarrow S'' \) is derivable for any \( \pi'' \). We can then infer that \( S \rightarrow S''' \)

\[
\frac{S \rightarrow S''' \quad \langle \pi'; \pi'', S''' \rangle \rightarrow S''}{\langle \pi'; \pi'', S \rangle \rightarrow S''} \quad (57)
\]

Hence, if \( \langle \pi'; \pi'', S''' \rangle \rightarrow S'' \) is derivable, we know by induction that \( \langle \pi'; \pi'', S \rangle \rightarrow S'' \) is derivable.

E. Assume that \( \langle \pi', S \rangle \rightarrow S' \) has been derived by (58) in the last step:

\[
\frac{\langle \pi', S \rangle \rightarrow S \quad S \rightarrow S'}{\langle \pi', S \rangle \rightarrow S'} \quad (58)
\]

We have the second hypothesis \( \langle \pi'', S' \rangle \rightarrow S'' \), and we need to show that \( \langle \pi'; \pi'', S \rangle \rightarrow S'' \). By second hypothesis and (57) with \( S \rightarrow S' \) we obtain:

\[
\frac{\bar{S} \rightarrow S' \quad \langle \pi'', S' \rangle \rightarrow S''}{\langle \pi'', \bar{S} \rangle \rightarrow S''} \quad (57)
\]

Hence, if we have as induction hypothesis that \( \langle \pi', S \rangle \rightarrow \bar{S} \) and \( \langle \pi'', \bar{S} \rangle \rightarrow S'' \) then we can follow \( \langle \pi'; \pi'', S \rangle \rightarrow S'' \).

F. Finally, assume that \( \langle \pi', S \rangle \rightarrow S' \) was derived by (59) in the last step:

\[
\frac{S \equiv S_0 \quad \langle \pi', S_0 \rangle \rightarrow S_1 \quad S_1 \equiv S'}{\langle \pi', S \rangle \rightarrow S'} \quad (59)
\]

We are in the case \( \langle \pi', S_0 \rangle \rightarrow S_1 \) with \( S \equiv S_0 \) and \( S_1 \equiv S' \). Our second hypothesis is that \( \langle \pi'', S' \rangle \rightarrow S'' \), and we need to show that \( \langle \pi'; \pi'', S \rangle \rightarrow S'' \). By (59) we then have also that \( \langle \pi'', S_1 \rangle \rightarrow S'' \):

\[
\frac{S_1 \equiv S' \quad \langle \pi'', S_1 \rangle \rightarrow S''}{\langle \pi'', \bar{S} \rangle \rightarrow S''} \quad (59)
\]

By induction hypothesis we have that \( \langle \pi', S_0 \rangle \rightarrow S_1 \) and \( \langle \pi'', S_1 \rangle \rightarrow S'' \) implies \( \langle \pi'; \pi'', S_0 \rangle \rightarrow S'' \). By (59) again, \( \langle \pi'; \pi'', S \rangle \rightarrow S'' \).

3. Proof of Theorem 3

**Proof:** by induction over the structure of \( S \).
A. Case $S \equiv \emptyset$.
   This holds in the initial state. The program to create $S \equiv \emptyset$ is $c: (\epsilon, \emptyset) \rightarrow \emptyset$.

B. Case $S \equiv S' \circ S''$ with $S', S'' \in \omega^+(\Pi_{\min})$.
   By induction, there exist programs $\pi'$ and $\pi''$ such that $\langle \pi', \emptyset \rangle \rightarrow S'$ and $\langle \pi'', \emptyset \rangle \rightarrow S''$.
   Lemma 1 then implies that $\langle \pi', \emptyset \circ \emptyset \rangle \rightarrow S' \circ \emptyset$, and $\langle \pi'', S' \circ \emptyset \rangle \rightarrow S' \circ S''$. It follows with lemma 2 that $\langle \pi'; \pi'', \emptyset \circ \emptyset \rangle \rightarrow S' \circ S''$. But $\emptyset \circ \emptyset \equiv \emptyset$ and $S' \circ S'' \equiv S$, hence $\langle \pi'; \pi'', \emptyset \rangle \rightarrow S$ due to inference rule (59).

C. Case $S \equiv x_i : P$.
   
   C.1. Subcase $P \equiv 0$.
   This holds in the initial state. The program to create $S \equiv x_i : 0$ is $c: (\epsilon, \emptyset) \rightarrow x_i : 0$.

   C.2. Subcase $P \equiv P' + P''$.
   This case can be reduced to case B. by means of the distributive relation (18): $x_i : P' + P'' \equiv x_i : P' \circ x_i : P''$.

   C.3. Subcase $P \equiv q \ll P'$ with $d(P') = 0$.
   
   C.3.1. Subsubcase $P \equiv q \ll 0 \ll$.
   This is achieved by the program $\pi = \text{feed}(x_j, m, 0); \text{transport}(x_j, x_i, x_k, \sigma)$ for arbitrary $m \in M$, $x_j \in \mathcal{L}_{\text{feed}}, x_k \in \mathcal{L}_{\text{feed}, \sigma} : \sigma \in \mathcal{T}$.

   C.3.2. Subsubcase $P \equiv q \ll m_j \ll$.
   This is achieved by the program $\pi = \text{feed}(x_j, m_j, 1); \text{transport}(x_j, x_i, x_k, \sigma)$ for $x_j \in \mathcal{L}_{\text{feed}}, x_k \in \mathcal{L}_{\text{feed}, \sigma} : \sigma \in \mathcal{T}$.

   C.3.3. Subsubcase $P \equiv q \ll s_k \ll$.
   Since $d(q \ll s_k \ll) = 2$, we have to show that $x_i : q \ll s_k \ll \not\equiv \omega^+(\Pi_{\min})$. Observe that $\text{feed}$ and $\text{feed}_\text{tag}$ either raise the nesting level of a state from 0 to 1, or leave it invariant if it was higher than 1. The instructions $\text{tag}$, $\text{move}$ and $\text{fuse}$ leave the nesting level unaltered, whereas $\text{flush}$ reduces the nesting level to 0 or leaves it invariant. Therefore, the instruction set $I_{\min}$ does not contain any instruction that would increase the nesting level from 1 to 2. Therefore, $S \equiv x_i : q \ll s_k \ll$ is not constructable from $\emptyset$.

   C.3.4. Subsubcase $P \equiv q \ll P' + P'' \ll$.
   By induction, there exists a program $\pi$ such that $\langle \pi, \emptyset \rangle \rightarrow x_i : q \ll P' \circ x_j : q \ll P'' \ll$.
   Then, for some $x_k \in \mathcal{L}_{\text{feed}, \sigma} : \sigma \in \mathcal{T}$, the program $\pi; \pi'$ will produce $S$, where $\pi'$ is the following program:

   - $\text{feed}_\text{tag}(x_k, \sigma, 1)$
   - $\text{move}(x_k, x_j, \sigma)$
   - $\text{tag}(x_j)$
   - $\text{move}(x_j, x_i, \sigma)$
   - $\text{fuse}(x_i)$
   - $\text{feed}_\text{tag}(x_k, \sigma \triangleright \circ, 1)$
   - $\text{move}(x_k, x_i, \sigma \triangleright \circ, 1)$

   And $x_i : q \ll P' \circ x_j : q \ll P'' \ll$.
C.3.5. Subcase $P \equiv (s_k + q)P'$.

By induction, there exists a program $\pi$ such that $(\pi, \emptyset) \rightarrow x_i : qP'\upharpoonright$. Then, for any $x_k \in \mathcal{L}_{\text{feed_tag}}$, the program $\pi; \pi'$ will produce $S$, where $\pi'$ is the program:

\[
\begin{align*}
&x_i : qP' \\
&\text{feed_tag}(x_j, s_k, 1) \\
&\quad x_i : qP' \circ x_j : s_k \\
&\text{move}(x_j, x_i, s_k) \\
&\quad x_i : s_k + qP' \\
&\text{tag}(s_k) \\
&\quad x_i : (s_k + q)P'
\end{align*}
\]

C.4. Subcase $P \equiv m_j$.

We have to show that $x_i : m_j \not\in \omega^+(\Pi_{\text{min}})$. First, observe that $d(x_i : m_j) = 0$. The only way to introduce $m_j$ is by means of the command $\text{feed}(x_i, m_j, 1)$ which will result in a nesting level of 1. As discussed in subcase C.3.3., the only means to decrease the nesting level is by means of the instruction $\text{flush}$. However, $\text{flush}$ will transform the state $x_i : (m_j)\upharpoonright$ into the empty state, which is not equivalent to $S$. Thus, there is no program in $\Pi_{\text{min}}$ able to generate $S \equiv x_i : m_j$. Therefore, $S \not\in \omega^+(\Pi_{\text{min}})$.

C.5. Subcase $P \equiv s_k$.

This is achieved by the program $\pi = \text{feed_tag}(x_j, s_k, 1); \text{move}(x_j, x_i, s_k)$, for any $x_k \in \mathcal{L}_{\text{feed_tag}}$.

4. Proof of Theorem 4

Proof: The proof is identical to the one of theorem 3, where subcase C.3.3. is replaced by the more general subsubcase:

C.3.3’ Subcase $P \equiv (\emptyset)P'$ with $d(P') > 0$, $P \neq P' + P''$, and $P \neq m_j$.

By induction, there exists a program $\pi \in \Pi_{\text{wrap}}$, such that $(\pi, \emptyset) \rightarrow x_j : P'$ for $x_j \in \mathcal{L}_{\text{wrap_in}}$. The desired state is obtained by the program $\pi; \pi'$, where $\pi'$ is the program

\[
\begin{align*}
&x_j : P' \\
&\text{wrap}(x_j, x_k) \\
&\quad x_k : (\emptyset)P' \\
&\text{transport}(x_k, x_i, x_l, \sigma) \\
&\quad x_i : (\emptyset)P'
\end{align*}
\]

where $x_k \in \mathcal{L}_{\text{wrap_out}}, x_l \in \mathcal{L}_{\text{feed_tag}}, \sigma \in T$.

5. Proof of Theorem 5

Proof: The proof is identical to the one of theorem 4 where case C.4. is replaced by:

C.4’ Case $P \equiv m_j$.

By induction, there exists a program $\pi \in \Pi_{\text{burst}}$, such that $(\pi, \emptyset) \rightarrow x_i : \sigma m_j \upharpoonright$. The desired state is obtained by the program $\pi; \pi'$, where $\pi'$ is the program

\[
\begin{align*}
&x_i : \sigma m_j \\
&\text{burst}(x_i) \\
&\quad x_i : \sigma \emptyset + m_j \\
&\text{move}(x_i, x_k, \sigma) \\
&\quad x_i : m_j \circ x_k : \sigma \emptyset \\
&\text{flush}(x_k) \\
&\quad x_i : m_j
\end{align*}
\]
6. Proof of Theorem 6

Proof: Again, we proof the theorem by induction over the structure of $S$.

A. Case $S \equiv \emptyset$.
Nothing needs to be done in this case: $\langle \epsilon, \emptyset \rangle \rightarrow \emptyset$.

B. Case $S \equiv S' \circ S''$.
Just as in the proof for theorem 3, we know by induction that there exist $\pi', \pi''$ such that $\langle \pi', S' \rangle \rightarrow \emptyset$ and $\langle \pi'', S'' \rangle \rightarrow \emptyset$. If follows through lemmas 1 and 2 that $\langle \pi'; \pi'', S' \circ S'' \rangle \rightarrow \emptyset$.

C. Case $S \equiv x_i : P$.

C.1. Subcase $P \equiv 0$.
Nothing needs to be done in this case: $\langle \epsilon, \emptyset \rangle \rightarrow \emptyset$.

C.2. Subcase $P \equiv P' + P''$.
This is structurally equivalent to $x_i : P' \circ x_i : P''$ and is therefore reduced to case B.

C.3. Subcase $P \equiv q \langle P' \rangle$.

C.3.1. Subsubcase $q \equiv \diamond$.

This reduces the problem to subsubcase C.3.2.

C.3.2. Subsubcase $q \equiv s_k + q'$.

For $x_j \in L_{\text{flush}}$, the following program resets the state:

$x_i : (s_k + q) \langle P \rangle$

move($x_i, x_j, s_k$) $x_j : (s_k + q) \langle P \rangle$
flush($x_j$) $\emptyset$

C.4. Subcase $P \equiv m_j$.
If $x_i \in L_{\text{flush}}$, the program flush($x_i$) will reset the state. Likewise, if $x_i \in L_{\text{wrap in}}$, the program wrap($x_i, x_j$) will transform the state into $x_j : \langle m_j \rangle$ for any $x_j \in L_{\text{wrap out}}$ and reduces the problem to subcase C.3.1.. On the other hand, if $x_i \notin L_{\text{flush}} \cup L_{\text{wrap in}}$, we have to show that $S$ is not an element of $\omega^-(\Pi_s)$. Note that all instructions feed, feed_tag tag, move, burst leave invariant under transition. Thus, there is no sequence of instructions that would transform $x_i : m_j$ into the empty state. Therefore, $x_i : m_j \notin \omega^-(\Pi_s)$ for $x_i \notin L_{\text{flush}} \cup L_{\text{wrap in}}$.

C.5. Subcase $P \equiv s_k$.
For $x_j \in L_{\text{flush}}$, the following program resets the state:

$x_i : s_k$

move($x_i, x_j, s_k$) $x_j : s_k$
flush($x_j$) $\emptyset$