Revisiting glue expressiveness in component-based systems*

Cinzia Di Giusto and Jean-Bernard Stefani

INRIA Rhône Alpes, Grenoble, France

Abstract. We take a fresh look at the expressivity of BIP, a recent influential formal component model developed by J. Sifakis et al. We introduce a process calculus, called CAB, that models composite components as the combination of a *glue* (using BIP terminology) and subcomponents, and that constitutes a conservative extension of BIP with more dynamic forms of glues. We study the Turing completeness of CAB variants that differ only in their language for glues. We show that limiting the glue language to BIP glues suffices to obtain Turing-completeness, whereas removing priorities from the control language loses Turing-completeness. We also show that adding a simple form of dynamic component creation in the control language without priorities is enough to regain Turing completeness. These results complement those obtained on BIP, highlighting in particular the key role of priorities for expressivity.

1 Introduction

Component-based software engineering is by now well entrenched in various areas, from embedded systems to Web applications, and is supported by numerous standards, including UML. Its central tenet is that complex systems can be built by composing, or *gluing* together possibly independently developed components.

In their paper on glue expressiveness [3] Bliudze and Sifakis have proposed to look at the expressive power of *glues* or composition operators in an effort to assess the relative merits of different component frameworks with respect to their composition capabilities. In essence, the criterion they use to compare two sets $\mathcal{G}_1$ and $\mathcal{G}_2$ of composition operators is whether it is possible, given a family of primitive components $\mathcal{B}$ and an equivalence relation $\sim$ between these components, to find, for a given operator $g_1 \in \mathcal{G}_1$, a corresponding operator $g_2 \in \mathcal{G}_2$ such that all their compositions are equivalent, i.e. $\forall B_1, \ldots, B_n \in \mathcal{B}$: $g_1(B_1, \ldots, B_n) \sim g_2(B_1, \ldots, B_n)$. As a notable result, they showed that their BIP component framework, whose glues feature multiparty synchronization and priorities, is universal with respect to a family of operators defined by inference rules in a subset of the GSOS format.

This work, however, leaves open a number of questions, in particular regarding the form glues can take, and their intrinsic expressivity. Indeed, the notion of

* Research partially funded by ANR Project PiCoq, Fondation de Coopération Scientifique Digiteo Triangle de la Physique, and Minalogic Project Mind.

glue in [3] is essentially a static one. One may legitimately argue in favor of more dynamic forms of composition, e.g. to allow the creation of new components or the replacement of existing ones to accommodate different forms of software update. Even without considering full dynamic reconfiguration, one may take into account changes in configuration or interconnection between components, e.g. to accommodate different *modes* of operation, where the notion of mode is loosely understood as a collection of execution states [9]. It thus appears beneficial to consider not just static glues but glue processes in their own right.

In the paper, we adopt this view: we model component assemblages as terms in a process calculus, called CAB (for Components And Behaviors). A component assemblage (or composite component) in CAB takes the form $l[C_1; \ldots; C_n \rhd B]$, where l is the name of the composite, $C_1, \ldots, C_n$ are the subcomponents of the composite, i.e. the components that are glued together (using BIP terminology) in the assemblage, and B is the *glue* – a term in a simple process calculus which we call the *glue language*. By construction, we recover BIP glues as essentially single state processes of our glue language.

With this view of glues as terms of a glue language, new expressivity questions arise, such as:

1. What is the expressivity of the resulting process calculus (in particular, if we restrict the glue language to terms corresponding to BIP glues only)?
2. What is the expressivity of the calculus if we remove the possibility of specifying priority constraints in the glue language ?
3. What is the expressivity of the calculus if we add more dynamic forms of control, such as component creation, in the glue language ?

In this paper we (begin to) answer these questions using classical Turing-completeness as our benchmark for expressivity. Following BIP, the CAB calculus is parametric over a family $\mathcal{P}$ of primitive components. So if we considered a large enough family, these questions would be trivial. Instead, we restrict our primitive components to be given by terms from the glue language itself – which form a strict non-Turing-complete subset of CCS – so as to characterize the intrinsic expressivity of the glue language. The questions then become non-trivial, and we obtain answers that may even appear surprising. Indeed, we first show that even with the restricted glue language consisting of static BIP glues only, the resulting variant of CAB is Turing-complete. Second, we show that this expressivity is lost if one restricts oneself to a subset of the glue language without priority constraints. These results confirms the expressive power of priorities, which was pointed out but not necessarily as clearly apparent in earlier works on BIP and process calculi with priorities. Finally, as a first answer to the last question, we show that we recover Turing-completeness if we add a very simple form of component creation in our glue language without priorities.

To summarize, our contributions are the following:

– We introduce a new process calculus, CAB, that extends the BIP framework with dynamic composition (or glue) capabilities.

- We demonstrate the expressiveness of priorities in the BIP framework by showing that BIP glues, composing simple CCS processes, is enough to obtain a Turing-complete language, and that Turing completeness is lost if we remove priorities.
- We show that Turing-completeness can be retained if we introduce more dynamic aspects in the language, namely a simple form of component creation.

The paper is organized as follows. Section 2 introduces the CAB process calculus and defines its operational semantics in SOS style. Section 3 proves our first result: CAB, restricted to a control language consisting of BIP glues, is Turing-complete. Section 4 proves our two other results: dropping priorities from CAB results in a non Turing-complete language; adding component creation to the control language without priorities is enough to regain Turing-completeness. Section 5 concludes the paper and discusses some related works.

2 CAB: syntax and semantics

We introduce in this section the CAB process calculus. In order to explain its constructs, as well as to make its relationship with the BIP framework clear, we begin by recalling the definition of the latter.

The BIP framework. We rely on the description of the BIP framework provided by [2,3]. A BIP component is simply a labeled transition system (LTS), whose labels are ports[1].

Definition 1. *A* component *is an LTS* $B = (Q, P, \rightarrow)$ *where*

1. Q *is a set of states*
2. P *is a set of ports*
3. $\rightarrow \subseteq Q \times P \times Q$ *is a set of transitions. We use* $q \xrightarrow{a} q'$ *to denote* $(q, a, q') \in \rightarrow$.

Components can be composed (glued) to form systems. A composition is given by a set of rules (the glue) that enforce synchronization and priority constraints among them.

Definition 2. *A* BIP system S *that glues together* n *components* $B_i = (Q_i, P_i, \rightarrow_i)$ *where ports and states are pairwise disjoint, is an LTS* $S = (\mathbb{Q}, \mathbb{P}, \rightarrow_S)$ *where* $\mathbb{Q} = \prod_{i=1}^{n} Q_i$, $\mathbb{P} = \bigcup_{i=1}^{n} P_i$ *and where* $\rightarrow_S$ *is a relation derivable as the least relation satisfying a finite set of rules*[2] *obeying the following format:*

$$r : \frac{\{B_i \xrightarrow{a_i} B_i'\}_{i \in I} \quad \{B_j \xrightarrow{b_j^k} \not\rightarrow \mid k \in [1..m_j]\}_{j \in J}}{(B_1, \ldots, B_n) \xrightarrow{a}_S (B_1', \ldots, B_n')} \tag{1}$$

[1] This is a difference with the definition in [3], where labels are defined to be *sets* of ports. We have adopted labels as simple ports in this paper to simplify the presentation. Our results are not impacted by this decision, however, for our processes only have a fixed finite number of distinct ports, so that we can always bijectively map a set of ports onto a single port.

[2] The finiteness of the set of rules defining a glue seems implicit in [3].

where I and J are sets of indexes in $[1,n]$, $B'_i = B_i$ if $I \notin I$, and $I \neq \emptyset$ (i.e. there is at least one positive premise).

Note that by definition there is at most one positive premise for each B_i in a rule in BIP format. The key features of the BIP framework are: (i) the ability to build hierarchical components; (ii) the concept of an explicit entity (the glue) responsible for the composition of components; (iii) the support of multipoint synchronizations, manifested by the positive premises in glue rules; (iv) The presence of priority constraints, given by the negative premises in glue rules.

The CAB calculus. As indicated in the introduction, we retain for CAB the general structure of composite components suggested by the BIP framework: a component in CAB takes the form $l[C_1; \ldots; C_n \rhd B]$, where l is the name of the component, $C_1, \ldots, C_n$ are its subcomponents and B is the glue. In contrast to glues in BIP, a glue in CAB can evolve over time, corresponding to changes in the synchronization and priority constraints among components, and is given by a term of a process calculus we call the *glue language*. We adopt in this paper a very simple glue language featuring:

- *Action prefix* $\alpha.B$, where α is an action, and B a continuation glue. The presence of action prefix in our glue language allows the definition of dynamic glues.
- *Parallel composition* $B_1 \parallel B_2$, where B_1 and B_2 are two glues. The parallel composition of glues can be interpreted as an *and* operator combining the synchronization and priority constraints embodied by B_1 and B_2. It is important to note that the two branches B_1 and B_2 in a parallel composition $B_1 \parallel B_2$ do *not* interact.
- *Recursion* $recX.B$, where X is a process variable, and B a glue. This allows the definition of glues with cyclic behaviors.

Formally, let $\mathcal{N}_P = \{a, b, c \ldots\}$ and $\mathcal{N}_C = \{h, k, l \ldots\}$ be denumerable sets of ports names and components names respectively. The CAB calculus is parametric over a set $\mathcal{P}$ of *primitive components* defined as labeled transition systems with labels in $\mathcal{N}_P$. We define CAB($\mathcal{P}$) processes as follows:

Definition 3 (CAB). *The set of CAB($\mathcal{P}$) processes is described by the following grammar, where P denotes an element of $\mathcal{P}$:*

$$\begin{array}{ll} S ::= l[C \rhd B] \mid l[P] & Act ::= \emptyset \mid \{evt\} \\ C ::= \mathbf{0} \mid S \mid C; C & evt ::= l : a \mid evt, evt \\ B ::= \mathbf{0} \mid \langle Act, Tag, Act \rangle.B \mid B \parallel B \mid rec\, X.B \mid X & Tag ::= \tau \mid a \end{array}$$

In order to simplify notation we write $l : a$ instead of $l : \{a\}$, and a instead of $l : a$ when it is clear from the context which component is providing event a. We abbreviate $\alpha.\mathbf{0}$ to α. We define $S.\mathtt{nm} = l$ if $S = l[P]$ or $S = l[C \rhd B]$ for some P, C, B (i.e. the function $\mathtt{nm}$ returns the name of an individual component S).

Actions in our glue language differ from those in classical process calculi, such as CCS, for they play different roles: they embody synchronization and priority

$$\textsc{Rec}\ \frac{B\{^{X}/_{rec\,X.B}\} \xrightarrow{\alpha} B'}{rec\,X.B \xrightarrow{\alpha} B'} \qquad \textsc{Par1}\ \frac{B \xrightarrow{\alpha} B'}{B \parallel B_2 \xrightarrow{\alpha} B' \parallel B_2} \qquad \textsc{Par2}\ \frac{B \xrightarrow{\alpha} B'}{B_2 \parallel B \xrightarrow{\alpha} B_2 \parallel B'}$$

$$\textsc{Act}\ \langle pr, tag, syn\rangle.B \xrightarrow{\langle pr,tag,syn\rangle} B$$

$$\textsc{Tau}\ \frac{C_i \xrightarrow{\tau} C_i'}{l[C_1;\ldots;C_i;\ldots;C_m \rhd B] \xrightarrow{\tau} l[C_1;\ldots;C_i';\ldots;C_m \rhd B]}$$

$$\textsc{Beh}\ \frac{C_{i_1} \xrightarrow{a_1} C_{i_1}' \ldots C_{i_n} \xrightarrow{a_n} C_{i_n}'\ B \xrightarrow{\langle pr,tag,\{l_{i_1}:a_1,\ldots,l_{i_n}:a_n\}\rangle} B'\ C_1 \ldots C_m \vdash pr}{l[C_1;\ldots;C_m \rhd B] \xrightarrow{tag} l[C_1';\ldots;C_m' \rhd B']}$$

where $I = \{i_1,\ldots,i_n\} \subseteq [1,m], \forall i \in I, C_i.\mathtt{nm} = l_i$ and $\forall j \in [1,m] \setminus I, C_j' = C_j$

Fig. 1: A labeled transition system semantics for CAB($\mathcal{P}$).

constraints that apply to subcomponents in a composition, and they provide a form of label renaming. An action is a triplet of the form $\langle pr, tag, syn\rangle$, where pr is a priority constraint (i.e. events in subcomponents which would preempt the synchronization syn), syn is a synchronization constraint (i.e. events to be synchronized between subcomponents), and tag is an event made visible by the composite as a result of a successful syn synchronization.

Hence a glue B of the form $\langle\{l : a\}, t, \{l_1 : c_1, l_2 : c_2\}\rangle.B'$ specifies a synchronization constraint between two subcomponents l_1 and l_2: if the first one is ready to perform event c_1, and the other is ready to perform event c_2, then the composition is ready to perform event t, provided that subcomponent l is not ready to perform event a. When the event t of the composite is performed (implying the two subcomponents l_1 and l_2 have performed events c_1 and c_2, respectively), a new glue B' is then put in place to control the behavior of the composite. Note that tag t can be either τ (which denotes an internal event) or a port (an event). Hence a tag $t = \tau$ results in a synchronization between subcomponents that takes place silently, with no implication from the environment of the composite. A tag $t \neq \tau$ subjects the evolution of the composite to the availability of an appropriate synchronization on t in the environment of the composite.

The operational semantics of CAB($\mathcal{P}$) is defined as the least labeled transition relation derivable by the inference rules in Figure 1. Rules for parallel composition and recursion are defined as usual. Rules Beh and Tau define the evolution of an aggregation of components inside a composite named l. Rule Beh stipulates that if a glue B is ready to perform an action $\langle pr, tag, \{l_1 : a_1, \ldots, l_n : a_n\}\rangle$ and components named $l_1, \ldots, l_n$ are ready to perform $a_1, \ldots, a_n$ respectively, then their composition is ready to perform action tag, provided priority constraint pr is satisfied. Having a priority constraint satisfied is defined as follows.

Let $pr = \{l_{j_1} : c_{j_1}, \ldots, l_{j_k} : c_{j_k}\}$ with $J = \{j_1 \ldots j_m\} \subseteq [1, m]$, we say that $C_1 \ldots C_m \vdash pr$ iff for every $i \in J$, $S_i \stackrel{c_i}{\not\longrightarrow}$ with $S_i.\mathtt{nm} = l_i$ and $S_i \in \{C_1, \ldots, C_m\}$. If $pr = \emptyset$ we are not imposing any priority policy on the synchronization. Similarly, with an action of the form $\langle pr, tag, \emptyset \rangle$ there is no synchronization requirement, but the environment of the composite must be ready to perform tag in order for the system to evolve.

Notation 1 *We denote with* $!\alpha.P$ *the process* $rec\, X.\ \alpha.(P \parallel X)$. *We use* $\longrightarrow$ *to denote the relation* $\stackrel{\tau}{\rightarrow}$. *We use* $\prod_{i=1}^{n} B_i$ *to denote* $B_1 \parallel \ldots \parallel B_n$[3].

Encoding BIP. The operational semantics, and in particular rule BEH, above was defined so as to mimic very closely the capabilities of glues in BIP. We now clarify the relationship between CAB($\mathcal{P}$) and BIP systems defined over a set $\mathcal{P}$ of components. We can encode a BIP glue G in CAB($\mathcal{P}$) as follows. By definition, G is given by a finite set of rules r that obey the format given in Definition 2. Let r be such a rule:

$$r : \frac{\{C_i \stackrel{a_i}{\longrightarrow} C_i'\}_{i \in I} \quad \{C_j \stackrel{c_j^k}{\not\longrightarrow} |\ k \in [1..m_j]\}_{j \in J}}{(C_1, \ldots, C_n) \stackrel{tag}{\longrightarrow} (C_1', \ldots, C_n')}$$

where I and J are set of indexes in $[1, n]$. The encoding $[\![r]\!]$ of rule r in CAB($\mathcal{P}$) is defined as:

$$[\![r]\!] = !\langle \{h_j : c_j^k \mid k \in [1, m_j]\}_{j \in J}, tag, \{h_i : a_i\}_{i \in I} \rangle$$

A BIP composition S with glue rules $r_1, \ldots, r_p$ and components $C_1, \ldots, C_n \in \mathcal{P}$ can thus be encoded as follows:

$$[\![S]\!] = l[h_1[C_1]; \ldots; h_n[C_n] \rhd \prod_{i=1}^{p} [\![r_i]\!]]$$

By construction, we obtain:

Theorem 2. *BIP systems defined over a set* $\mathcal{P}$ *of components can be encoded in CAB(*$\mathcal{P}$*): any BIP system* S *is strongly bisimilar to its encoding* $[\![S]\!]$.

3 Turing-completeness of CAB

In this section as in the rest of the paper, we work within CAB($\emptyset$), which, for simplicity, we denote CAB. We show the Turing-completeness of CAB by proving we can encode Minsky machines into it. This gives us a result on the *intrinsic* expressive power of the CAB glue language, in the sense that it does not depend on the presence of primitive components: we only construct component systems using glue language terms. Note that this is equivalent to considering

[3] The parallel operator $\parallel$ is commutative and associative modulo strong bisimilarity.

$$\text{M-Inc}\ \frac{i : \texttt{INC}(r_j) \quad m'_j = m_j + 1 \quad m'_{1-j} = m_{1-j}}{(i, m_0, m_1) \longrightarrow_{\text{M}} (i+1, m'_0, m'_1)}$$

$$\text{M-Dec}\ \frac{i : \texttt{DECJ}(r_j, s) \qquad m_j \neq 0 \qquad m'_j = m_j - 1 \qquad m'_{1-j} = m_{1-j}}{(i, m_0, m_1) \longrightarrow_{\text{M}} (i+1, m'_0, m'_1)}$$

$$\text{M-Jmp}\ \frac{i : \texttt{DECJ}(r_j, s) \quad m_j = 0}{(i, m_0, m_1) \longrightarrow_{\text{M}} (s, m_0, m_1)} \qquad \text{M-Halt}\ \frac{i : \texttt{HALT}}{(i, m_0, m_1) \not\rightarrow_{\text{M}}}$$

Fig. 2: Semantics of Minsky machines

only primitive components which are labeled transition systems defined by CAB terms of the form $l[\mathbf{0} \triangleright B]$, where B is a term with actions of the form $\langle \emptyset, a, \emptyset \rangle$. For reference, these primitive processes are given by terms of the following grammar, whose operational semantics is given by rules Rec, Par1, Par2, and Act in Figure 1:

$$B ::= \mathbf{0} \mid \langle \emptyset, a, \emptyset \rangle.B \mid B \parallel B \mid rec\, X.B \mid X.$$

Minsky Machines. Minsky machines [10] provide a Turing-complete model of computation. A Minsky machine is composed of a set of sequential, labeled instructions, and at least two registers. Registers r_j ($j \in \{0, 1\}$) can hold arbitrarily large natural numbers. Instructions $(1 : I_1), \ldots, (n : I_n)$ can be of two kinds: $\texttt{INC}(r_j)$ adds 1 to register r_j and proceeds to the next instruction; $\texttt{DECJ}(r_j, s)$ jumps to instruction s if r_j is zero, otherwise it decreases register r_j by 1 and proceeds to the next instruction. A Minsky machine includes a program counter p indicating the label of the instruction being executed. In its initial state, the machine has both registers initialized to m_0 and m_1 respectively, and the program counter p is set to the first instruction. The Minsky machine stops whenever the program counter is set to the $\texttt{HALT}$ instruction. A *configuration* of a Minsky machine is a tuple (i, m_0, m_1); it consists of the current program counter and the values of the registers. Formally, the reduction relation over configurations of a Minsky machine , denoted $\longrightarrow_{\text{M}}$, is defined in Figure 2.

The encoding. The encoding of Minsky machines in CAB, denoted $[\![\cdot]\!]_1$, is given in Figure 3. We now give some intuitions on it. Given a Minsky machine M, we encode it as a system m. m contains three components: the two registers r_0 and r_1, and the program counter. The instructions of the machine are encoded in the glue of m. Numbers inside registers are encoded in the glue as the parallel composition of as many occurrences of the unit process $\langle \emptyset, u_j, \emptyset \rangle$ as the number to be encoded. An increment simply adds an occurrence of the unit process $\langle \emptyset, u_j, \emptyset \rangle$ to the register. The decrement and jump is encoded as the parallel composition

$$[\![R_j = m]\!]_1 = r_j[\mathbf{0} \rhd \prod_1^m \langle \emptyset, u_j, \emptyset\rangle \parallel !\langle \emptyset, z_j, \emptyset\rangle \parallel !\langle \emptyset, inc_j, \emptyset\rangle.\langle \emptyset, u_j, \emptyset\rangle]$$

INSTRUCTIONS $(i : I_i)$

$$\begin{aligned}
[\![(i : \texttt{INC}(r_j))]\!]_1 &= !\langle \emptyset, \tau, \{p_i, inc_j, next_{i+1}\}\rangle \\
[\![(i : \texttt{DECJ}(r_j, s))]\!]_1 &= !\langle \emptyset, \tau, \{p_i, u_j, next_{i+1}\}\rangle \parallel !\langle r_j : u_j, \tau, \{p_i, z_j, next_s\}\rangle) \\
[\![(i : \texttt{HALT})]\!]_1 &= \langle \emptyset, halt, p_i\rangle
\end{aligned}$$

Fig. 3: Encoding of Minsky machines into CAB.

of the two branches. The decrement branch simply removes one occurrence of the unit process $\langle \emptyset, u_j, \emptyset\rangle$, if such occurrence is available. The jump branch is guarded by the priority $r_j : u_j$. In other words, to be able to execute the jump, it is necessary to check that the register is indeed empty. If this is the case the program counter is updated accordingly. More formally, the encoding of a configuration in the Minsky machine is defined as follows:

Definition 4. *Let M be a Minsky machine and (k, m_0, m_1) one of its configurations. The encoding of $[\![k, m_0, m_1]\!]_1$ is defined as*

$$\begin{aligned}
m[[\![R_0 = m_0]\!]_1; [\![R_1 = m_1]\!]_1; pr[\mathbf{0} \rhd \prod_{i=1}^n !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle]; \\
pr[\mathbf{0} \rhd \langle \emptyset, p_k, \emptyset\rangle \parallel \prod_{i=1}^n !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle] \rhd \prod_{i=1}^n [\![i : I_i]\!]_1]
\end{aligned}$$

where the encoding of registers and instructions is defined in Figure 3.

Notice that in order to synchronize at the same time p_i and $next_i$ we have to duplicate the component representing the program counter. This does not introduce non determinism as only one instance of the action $\langle \emptyset, p_i, \emptyset\rangle$ is available at every step.

The correctness of the encoding follows by a case analysis on the type of instruction performed when the program counter reaches k. This is formalized by the following Lemma.

Lemma 1. *Let M be a Minsky machine and (k, m_0, m_1) one of its configuration then $(k, m_0, m_1) \longrightarrow_{\mathrm{M}} (k', m_0', m_1')$ iff $[\![k, m_0, m_1]\!]_1 \longrightarrow [\![k', m_0', m_1']\!]_1$.*

Proof (Sketch). Here we show only that if $(k, m_0, m_1) \longrightarrow_{\mathrm{M}} (k', m_0', m_1')$ then $[\![k, m_0, m_1]\!]_1 \longrightarrow [\![k', m_0', m_1']\!]_1$ when the k-th instruction is a decrement on register $m_0 > 0$. The other cases and the other direction are similar or simpler.

Then, from Definition 4, we have that

$$\begin{aligned}
m[[\![R_0 = m_0]\!]_1; [\![R_1 = m_1]\!]_1; pr[\mathbf{0} \rhd \prod_{i=1}^n !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle]; \\
pr[\mathbf{0} \rhd \langle \emptyset, p_k, \emptyset\rangle \parallel \prod_{i=1}^n !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle] \rhd \prod_{i=1}^n [\![i : I_i]\!]_1]
\end{aligned}$$

where the k-th instruction is encoded as

$$!\langle\emptyset, \tau, \{p_k, u_0, next_{k+1}\}\rangle \parallel !\langle r_0 : u_0, \tau, \{p_k, z_0, next_s\}\rangle)$$

and $m_0' = m_0 - 1$, $k' = k + 1$. In this case, the only possible evolution is the one that synchronizes the program counter p_k, the unit u_0 inside register r_0 and $next_{k+1}$, evolving into the system:

$$m[[\![R_0 = m_0 - 1]\!]_1; [\![R_1 = m_1]\!]_1; pr[\mathbf{0} \rhd \langle\emptyset, p_{k+1}, \emptyset\rangle \parallel \prod_{i=1}^{n} !\langle\emptyset, next_i, \emptyset\rangle.\langle\emptyset, p_i, \emptyset\rangle];$$
$$pr[\mathbf{0} \rhd \prod_{i=1}^{n} !\langle\emptyset, next_i, \emptyset\rangle.\langle\emptyset, p_i, \emptyset\rangle] \rhd \prod_{i=1}^{n} [\![i : I_i]\!]_1]$$

Now, it is easy to see that the system above corresponds to $[\![k', m_0', m_1']\!]_1$. □

By means of the previous lemma, we can state the operational correspondence between M and its encoding $[\![M]\!]_1$.

Theorem 3. *Let M be a Minsky machine and $[\![M]\!]_1$ as defined in Definition 4. Then M halts with registers $R_i = m_i'$ for $i \in [0, 1]$ iff $[\![M]\!]_1 \xrightarrow{halt}$ and locations r_i for $i \in [0, 1]$ is $[\![R_i = m_i']\!]_1$.*

It is important to notice that our encoding relies on elementary components of the form $l[\mathbf{0} \rhd B]$, which are glued together by glue terms which are essentially in BIP format, as discussed in Section 2. The above theorem gives us actually a stronger result which says that the subset of CAB where glues are restricted to be in BIP format, and where primitive components correspond to labeled transition systems given by elementary components of the form $l[\mathbf{0} \rhd B]$, is Turing-complete.

4 Expressivity of CAB variants

We have shown that CAB is Turing powerful. We now investigate the sources of expressiveness in the language. The first thing we show is that in the encoding given in Section 3 the presence of priorities is essential. Indeed we can prove that if we consider a fragment of CAB without priorities the resulting language is not Turing powerful anymore. This can be proven by providing an encoding into Petri nets, a well known non Turing-powerful model.

4.1 CAB without priorities

A *Petri net* (see e.g. [6]) is a tuple $N = (P, T, m_0)$, where P and T are finite sets of *places* and *transitions*, respectively. A finite multiset over the set S of places is called a *marking*, and m_0 is the initial marking. Given a marking m and a place p, we say that the place p contains $m(p)$ *tokens* in the marking m if there

are $m(p)$ occurrences of p in the multiset m. A transition is a pair of markings written in the form $m' \Rightarrow m''$. The marking m of a Petri net can be modified by means of transitions firing: a transition $m' \Rightarrow m''$ can fire if $m(p) \geq m'(p)$ for every place $p \in S$; upon transition firing the new marking of the net becomes $n = (m \setminus m') \uplus m''$ where $\setminus$ and $\uplus$ are the difference and union operators for multisets, respectively. This is written as $m \to n$.

We denote the fragment of CAB without priorities as CAB^{-p}. This fragment is obtained by replacing production $\langle Act, Tag, Act\rangle$ with $\langle \emptyset, Tag, Act\rangle$ in Definition 3. Before presenting the encoding into Petri Nets, we introduce some more terminology: we define a notion of top level actions in the glue of a component.

Definition 5 (top). *Let $l[C \rhd B]$ be a system in CAB. $top(B)$ is defined inductively on the structure of the glue B as follows:*

$$\begin{array}{ll} top(\mathbf{0}) = top(X) ::= \emptyset & top(\langle pr, tag, syn\rangle.B) ::= \{\langle pr, tag, syn\rangle\} \\ top(recX.B) ::= top(B) & top(B_1 \parallel B_2) ::= top(B_1) \cup top(B_2) \end{array}$$

We also define how to build the graph of precedence of a glue B:

Definition 6. *Let $l[C \rhd B]$ be a system in CAB. The graph of B, denoted with $\mathcal{G}(B) = (Nodes(B), Edges(B))$, is a directed graph, inductively defined as:*

$$\begin{array}{lll} \mathcal{G}(\mathbf{0}) & ::= & Nodes(B) = \{\mathbf{0}\}, \\ & & Edges(B) = \emptyset \\ \mathcal{G}(\langle act\rangle.B_1) & ::= & Nodes(B) = \{\langle act\rangle\} \cup Nodes(B_1), \\ & & Edges(B) = \{\langle act\rangle \to x \mid x \in top(B_1)\} \cup Edges(B_1) \\ \mathcal{G}(B_1 \parallel B_2) & ::= & Nodes(B) = Nodes(B_1) \cup Nodes(B_2), \\ & & Edges(B) = Edges(B_1) \cup Edges(B_2) \\ \mathcal{G}(recX.B_1) & ::= & Nodes(B) = Nodes(B_1) \\ & & Edges(B) = Edges(B_1) \text{ where every time we encounter} \\ & & \qquad X \text{ we add an edge to the nodes in } top(B_1) \end{array}$$

Let $n \in Nodes(B)$, we denote with $Adj(n)$ the list of nodes adjacent to n.

The idea is that every system is a Petri Net and the marking represents the components that are ready to interact at a given instant. Transitions mimic the semantics of CAB^{-p} systems. The construction of the Petri Net is inductive on the hierarchy of components: let $S = l_S[S_1; \ldots; S_m \rhd B_S]$ be a system in CAB^{-p}. We assume that k is the maximum number of levels of nesting in S. We decorate every location in S with the corresponding level of nesting in S, from 1 the innermost, to k the outermost level.

Let $\mathcal{PN}(S_i) = (P(S_i), T(S_i), m_0(S_i))$ be the Petri Net for the subsystem S_i for all $i \in [1, m]$. $\mathcal{PN}(S)$ is built by taking:

- as set of places, the set of all places of the subnets for $S_1 \ldots S_n$ plus all the nodes in the graph of the behavior B_S:

$$P(S) = \bigcup_{i=1}^{n} P(S_i) \cup \{[l_S^k : \langle \emptyset, tag, syn\rangle] \mid \langle \emptyset, tag, syn\rangle \in Nodes(B_S)\};$$

Notice that there is a bijection between nodes in the graphs of glues and the places in the Petri Net. Hence for every node n in the graph of glue located at l in level j there exists a distinctive place $[l^j : n]$ and vice-versa.

- as set of transitions all the transitions of subnets $\mathcal{PN}(S_1) \ldots \mathcal{PN}(S_n)$ plus for all nodes $\langle \emptyset, tag, syn \rangle$ in $Nodes(B_S)$ where $tag = \tau$ we add a set of transitions that:
  - Take as precondition, recursively on the part syn of the nodes considered, all the places $[l^j : \langle \emptyset, t, s \rangle]$ for $j \in [1, k-1]$ and such that $l : t$ appears in the synchronization part syn in one of the nodes. Notice that, this accounts in considering in a single transition all the components involved in a τ step: i.e. the places involved in the precondition correspond to all the leafs in the derivation tree of the τ step.
  - Take as postcondition all the places built from nodes in the adjacent list of all the nodes obtained by places in the preconditions.

  For instance, consider the system

$$l^3[l^2[l^1[\mathbf{0} \rhd \langle \emptyset, a, \emptyset \rangle.\mathbf{0}] \rhd \langle \emptyset, b, \{l^1 : a\} \rangle.\mathbf{0}] \rhd \langle \emptyset, \tau, \{l^2 : b\} \rangle.\mathbf{0}]$$

  here there is a single transition that takes as precondition the places: $\{[l^3 : \langle \emptyset, \tau, \{l^2 : b\} \rangle], [l^2 : \langle \emptyset, b, \{l^1 : a\} \rangle], [l^3 : \langle \emptyset, a, \emptyset \rangle]\}$ and as post condition the places $\{[l^1 : \mathbf{0}], [l^2 : \mathbf{0}], [l^3 : \mathbf{0}]\}$
- as initial marking, the initial marking of all subnets plus the nodes corresponding to the top level actions in B_S:

$$m_0(S) = \uplus_{i=1}^{n} m_0(S_i) \uplus \{[l_S^k : n] \mid n \in top(B_S)\}$$

The correctness of the above construction follows by induction on the nesting of components.

Theorem 4. *Let $S = l_S[S_1; \ldots; S_m \rhd B_S]$ be a system in CAB^{-p}, and $\mathcal{PN}(S) = (P(S), T(S), m_0(S))$ the corresponding Petri Net. Then $S \longrightarrow S'$ iff there exists a marking m' such that $m_0(S) \Rightarrow m'$ and m' is a marking that takes all the top level actions in S'.*

Proof. Here we show only the correctness direction, soundness is similar. Let $S = l_S[S_1; \ldots; S_m \rhd B_S]$ be a system in CAB^{-p}, and $m_0(S)$ the initial marking in the Petri Net constructed as described above. The proof proceeds by induction on the nesting of components in S. If $S \longrightarrow S'$ then we have that either rule Beh or Tau has been used. The case of Tau follows by inductive hypothesis. Instead if the τ step comes from Beh , we have that there exists an action $\langle \emptyset, \tau, \{a_1 \ldots a_n\} \rangle$ at top level in B_S. Moreover we have $C_{i_1} \ldots C_{i_n}$ components that are offering actions $a_1 \ldots a_n$ respectively. Hence at top level in these components we have an action $\langle \emptyset, a_{i_j}, syn \rangle$ for $j \in [1, n]$. Therefore, by construction we have a token in all these places and the transition can fire, moving all tokens in the successors of the action: i.e. in all the nodes of the adjacency list, that by construction corresponds to the new action at top level in S'. □

4.2 Recovering expressiveness

We, now, introduce a new construct to CAB^{-p} to recover the loss of expressiveness due to the absence of priorities. We consider an operator that adds new components inside a system. To this aim, we add to Definition 3 the following production:

$$B ::= new\ S$$

with this operational semantics:

$$\textsc{New}\ new\ S \xrightarrow{new\ S} \mathbf{0} \qquad \textsc{Cre}\ \frac{B \xrightarrow{new\ S} B'}{l[C \rhd B] \xrightarrow{\tau} l[C; S \rhd B']}$$

Thanks to the interplay between the creation of new components and recursion we can re-obtain Turing equivalence. The result, similarly to the one in Section 3, is obtained by resorting to an encoding of Minsky machines. We proceed by giving some intuitions on the encoding given in Figure 4. Registers are encoded as a hierarchy of components that handle both the representation of the number and a mechanism to increment or decrement. The nesting of these components represents the number contained. At every instant, the mechanism controlling the register is placed in the innermost position. Thus, whenever an increment takes place, a new component is created inside the deepest level and all the control is transfered to the newly created object: this is the role of $a[\mathbf{0} \rhd \langle \emptyset, act_j, \emptyset \rangle]$ which activates the current instance. On the contrary, in case of a decrement, the current instance is deactivated: i.e. it remains as garbage but it cannot be used anymore and a signal is passed to the upper component so to activate decrements and increments at the proper level of nesting. Notice, that in order to communicate with the active instance, it is necessary to equip every level of the nesting with a process Fwd. This process is responsible for forwarding increment and decrement events to reach the component that controls the simulation of the computation. Without loss of generality, we assume that registers are initialized to zero. The following definition formalizes the encoding of a Minsky machine M:

Definition 7. *Let M be a Minsky machine with registers initialized to 0 and program counter set to 1: its encoding $[\![M]\!]_2$ is*

$$m[[\![R_0 = 0]\!]_2; [\![R_1 = 0]\!]_2; pr[\mathbf{0} \rhd \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset \rangle.\langle \emptyset, p_i, \emptyset \rangle];$$
$$pr[\mathbf{0} \rhd \langle \emptyset, p_1, \emptyset \rangle \parallel \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset \rangle.\langle \emptyset, p_i, \emptyset \rangle] \rhd \prod_{i=1}^{n} [\![i : I_i]\!]_2]$$

where the encoding of registers and instructions is defined in Figure 4. [4]

[4] Notice that the interplay of recursion and creation of new components is implicit in the definition of INC and $Level$. The same thing could have been written as: $!recX\langle \emptyset, inc_j, act_j \rangle.new\ a[a[\mathbf{0} \rhd \langle \emptyset, act_j, \emptyset \rangle] \rhd Fwd \parallel DEC \parallel X]$.

$$
\begin{array}{lll}
[\![R_j = 0]\!]_2 & ::= & r_j[a[\mathbf{0} \rhd \langle \emptyset, act_j, \emptyset\rangle \parallel !\langle \emptyset, zero_j, \emptyset\rangle.\langle \emptyset, act_j, \emptyset\rangle] \rhd Fwd \parallel Z \parallel INC] \\
Fwd & ::= & !\langle \emptyset, inc_j, inc_j\rangle \parallel !\langle \emptyset, dec_j, dec_j\rangle \\
Z & ::= & !\langle \emptyset, z_j, act_j\rangle.\langle \emptyset, \tau, zero_j\rangle \\
INC & ::= & !\langle \emptyset, inc_j, act_j\rangle.new\ Level \\
Level & ::= & a[a[\mathbf{0} \rhd \langle \emptyset, act_j, \emptyset\rangle] \rhd Fwd \parallel DEC \parallel INC] \\
DEC & ::= & \langle \emptyset, dec_j, act_j\rangle.\langle \emptyset, act_j, \emptyset\rangle
\end{array}
$$

INSTRUCTIONS $(i : I_i)$

$$
\begin{array}{ll}
[\![(i : \texttt{INC}(r_j))]\!]_2 & = !\langle \emptyset, \tau, \{p_i, inc_j, next_{i+1}\}\rangle \\
[\![(i : \texttt{DECJ}(r_j, s))]\!]_2 & = !\langle \emptyset, \tau, \{p_i, dec_j, next_{i+1}\}\rangle \parallel !\langle \emptyset, \tau, \{p_i, z_j, next_s\}\rangle) \\
[\![(i : \texttt{HALT})]\!]_2 & = \langle \emptyset, halt, p_i\rangle
\end{array}
$$

Fig. 4: Encoding of Minsky machines into CAB without priorities.

Similarly as before, the correctness of the encoding follows by a case analysis on the type of instruction performed when the program counter reaches k. Notice that, depending on the specific computation there can be components as $a[a[\mathbf{0} \rhd \mathbf{0}] \rhd Fwd \parallel INC]$ "floating" in the system. Nevertheless this garbage can be ignored as it is never re-used: i.e. it cannot interact with the rest of the system.

Lemma 2. *Let M be a Minsky machine and (k, m_0, m_1) one of its configuration then $(k, m_0, m_1) \longrightarrow_{\mathrm{M}} (k', m_0', m_1')$ iff $[\![k, m_0, m_1]\!]_2 \longrightarrow [\![k', m_0', m_1']\!]_2$.*

Proof (Sketch). Here we show only that if $(k, m_0, m_1) \longrightarrow_{\mathrm{M}} (k', m_0', m_1')$ then $[\![k, m_0, m_1]\!]_2 \longrightarrow [\![k', m_0', m_1']\!]_2$ when the k-th instruction is a decrement on register $m_0 > 0$. The other cases and the other direction are similar or simpler.

We first define $[\![k, m_0, m_1]\!]_2$, for the sake of simplicity we will not consider the occurrences of garbage objects, taking for grant that those will not interfere with the computation.

$$
\begin{array}{rl}
[\![k, m_0, m_1]\!]_2 ::= & m[[\![R_0 = m_0]\!]_2; [\![R_1 = m_1]\!]_2; pr[\mathbf{0} \rhd \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle]; \\
& pr[\mathbf{0} \rhd \langle \emptyset, p_k, \emptyset\rangle \parallel \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset\rangle.\langle \emptyset, p_i, \emptyset\rangle] \rhd \prod_{i=1}^{n} [\![i : I_i]\!]_2]
\end{array}
$$

where

$$
\begin{array}{l}
[\![R_j = m_j]\!]_2 ::= r_j[a[\mathbf{0} \rhd \langle \emptyset, act_j, \emptyset\rangle \parallel !\langle \emptyset, zero_j, \emptyset\rangle.\langle \emptyset, act_j, \emptyset\rangle], \\
C[\ldots C[a[a[\mathbf{0} \rhd \langle \emptyset, act_j, \emptyset\rangle] \rhd Fwd \parallel DEC \parallel INC]] \ldots] \rhd Fwd \parallel Z \parallel INC]
\end{array}
$$

and $C[\bullet] = a[a[\mathbf{0} \rhd \mathbf{0}], \bullet \rhd Fwd \parallel DEC \parallel INC]$ is repeated m_j times. The k-th instruction is encoded as

$$
!\langle \emptyset, \tau, \{p_k, dec_0, next_{k+1}\}\rangle \parallel !\langle r_0 : u_0, \tau, \{p_k, z_0, next_s\}\rangle)
$$

and $m'_0 = m_0 - 1$, $k' = k + 1$. In this case, the only possible evolution is the one that synchronizes the program counter p_k, the message dec_0 inside register r_0 and $next_{k+1}$, evolving into the system:

$$m[[\![R_0 = m_0 - 1]\!]_2; [\![R_1 = m_1]\!]_2; pr[\mathbf{0} \rhd \langle \emptyset, p_{k+1}, \emptyset \rangle \parallel \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset \rangle.\langle \emptyset, p_i, \emptyset \rangle];$$
$$pr[\mathbf{0} \rhd \prod_{i=1}^{n} !\langle \emptyset, next_i, \emptyset \rangle.\langle \emptyset, p_i, \emptyset \rangle] \rhd \prod_{i=1}^{n} [\![i : I_i]\!]_2]$$

Notice that the message on dec_0 will start a chain of synchronizations between components $a[\dots]$ through the Fwd event to reach the deepest component and then activate the real decrement. It is easy to conclude that the system above corresponds to $[\![k', m'_0, m'_1]\!]_2$. □

The previous lemma allows us to conclude:

Theorem 5. *Let M be a Minsky machine and $[\![M]\!]_2$ as defined in Definition 7. Then M halts with registers $R_i = m'_i$ for $i \in [0, 1]$ iff $[\![M]\!]_2 \xrightarrow{halt}$ and locations r_i for $i \in [0, 1]$ is $[\![R_i = m'_i]\!]_2$.*

5 Final Remarks

We have taken in this paper a decidedly process algebraic view of glues in component-based systems, introducing an alternate view, and an extension, of the BIP framework in the form of the CAB process calculus. We have studied the expressiveness of CAB, which gave us a way to characterize the intrinsic (i.e. not relatively to a predefined family of components) expressive power of its glue language. We have shown that, while being very simple, the calculus is Turing-complete thanks mainly to the presence of priorities. As a matter of fact, we have shown that the fragment of CAB where priorities have been removed is only as expressive as Petri nets, which is a testament to the gain in expressive power obtained through the use of priorities. However expressiveness can be recovered in a calculus without priorities if dynamic operators are added to the language.

We have already discussed in the introduction the relations with the BIP framework and seen how the present paper brings new light on BIP expressiveness. Here we relate our paper to other works studying the expressiveness of multiparty synchronization or priority. Multiparty synchronization has been proposed in several process calculi. One of the first proposals is CSP [7] where synchronization can take place among all processes that share a channel with the same name. A recent work by Laneve and Vitale [8] has shown that a calculus able to synchronize on n channels is strictly more expressive than one that can only synchronize up to $n - 1$ channels. [5] shows a similar result in the context of a concurrent logic calculus. In the current paper we have mostly shown the benefit of priorities for expressiveness. However we suspect that multiparty

synchronization is also important for expressiveness. In our two encodings of Minsky machines in Section 3 and in Section 4, we rely decisively on 3-way synchronization; whether it is absolutely required is a question for further study.

Several works tackle the problem of adding priority mechanisms in a process calculus [4]. In [11] it has been shown that CCS enriched with a form of priority guards is strictly more expressive than CCS: essentially, it is possible to model the leader election problem in CCS with priorities, which is not the case with plain CCS. Analogously, [12] shows that a core calculus similar to CCS, if extended with several kinds of priorities, can model the leader election problem while the core calculus can not. Both these studies state the impossibility to encode the calculus with priorities in the plain calculus. In contrast, we show in this paper an absolute increase in expressiveness from Petri Nets to Minsky machines. Closer to the present work is the paper in [1], where the authors show that CCS without restriction, and with replication instead of recursion, can be encoded into Petri Nets while the same calculus enriched with priorities and a weak form of restriction is Turing-powerful. Compared to [1] we are considering recursive processes instead of replicated ones thus the drop of expressiveness when not using priorities is stronger in our case.

As for future work, we plan to investigate other, more involved, forms of dynamic configuration of components. Moreover we are interested in understanding if our result of Turing completeness can be related to the ability of simulating all recursively enumerable LTSs thus making unnecessary the presence of the parameter $\mathcal{P}$ in the full calculus CAB($\mathcal{P}$).

References

1. J. Aranda, F. Valencia, and C. Versari. On the expressive power of restriction and priorities in ccs with replication. In *FOSSACS*, pages 242–256. Springer, 2009.
2. S. Bliudze and J. Sifakis. The algebra of connectors - structuring interaction in bip. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.
3. S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *CONCUR*, volume 5201 of *LNCS*, pages 508–522. Springer, 2008.
4. R. Cleaveland, G. Lüttgen, and V. Natarajan. Priority in process algebra. Technical report, Nasa, 1999.
5. C. Di Giusto, M. Gabbrielli, and M. C. Meo. On the expressive power of multiple heads in chr. *To appear in ACM Transactions on Computational Logic*, 2010.
6. J. Esparza and M. Nielsen. Decidability issues for petri nets - a survey. *Bulletin of the EATCS*, 52:244–262, 1994.
7. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, 1985.
8. C. Laneve and A. Vitale. The expressive power of synchronizations. In *LICS '10*, pages 382–391, Washington, DC, USA, 2010. IEEE Computer Society.
9. F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46(3), 2003.
10. M. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.
11. I. Phillips. CCS with priority guards. *J. Log. Algebr. Progr.*, 75(1):139–165, 2008.
12. C. Versari, N. Busi, and R. Gorrieri. An expressiveness study of priority in process calculi. *Mathematical. Structures in Comp. Sci.*, 19:1161–1189, 2009.