Encapsulation and Sharing in Dynamic Software Architectures: The Hypercell Framework

Jean-Bernard Stefani, Martin Vassor

Encapsulation and Sharing in Dynamic Software Architectures: The Hypercell Framework

Jean-Bernard Stefani, Martin Vassor

Univ. Grenoble-Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

Abstract. We present in this paper a novel framework for the definition of formal software component models, called the Hypercell framework. Models in this framework (hypercell models) allow the definition of dynamic software architectures featuring shared components, and different forms of encapsulation policies. Encapsulation policies in an hypercell model are enforced by means of runtime checks that prevent a component, in a given context, to evolve in violation of these policies. We present the main elements of the framework, its operational semantics and the first elements of its behavioral theory. We give some results concerning its ability to express different forms of composition, and show by means of examples its ability to deal with sharing and different forms of encapsulation.

1 Introduction

Motivations. How do we formally model dynamic software architectures featuring both encapsulation and sharing? Can we define an operational semantics and behavioral theory for these architectures? These are the questions we deal with in this paper. By dynamic software architectures, we understand structured collections of software components and their inter-relations [3], that can evolve over time, either spontaneously, for instance to adapt to changing operating conditions, or following external intervention, for instance for purposes of fault correction or functional update. By encapsulation, we understand forms of confinement and isolation between components, typically coupled with information hiding and abstraction, that ensure capabilities offered, and information maintained by a component, can only be accessed through designated interaction points or interfaces. Examples include the many forms of encapsulation that have been studied under the topic of aliasing control and ownership types in object-oriented programming [14]. By architectures with sharing, we understand architectures where components can take part in different ensembles, compositions or aggregations, possibly with different attendant properties, e.g. in terms of encapsulation, lifetime and existential dependencies [2]. Examples include architectures featuring common services, such as databases or logs, that can be used by software components at different levels in a software structure, and architectures featuring shared resources such as virtual machines or operating system processes.

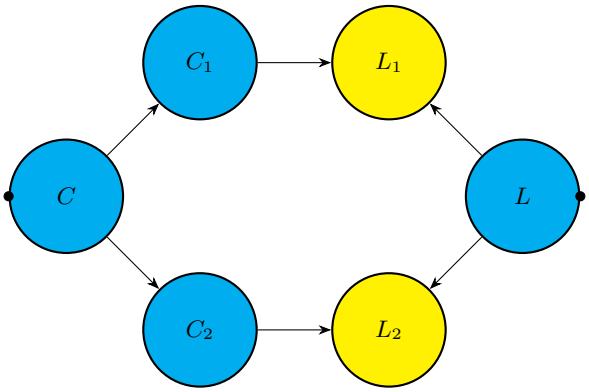

Fig. 1. Architecture with sharing: shared log service

An example can illustrate the questions we are concerned with. Consider the architecture depicted in Figure 1. In the figure, a composite component C has two subcomponents, C_1 and C_2, equipped with private log subcomponents L_1 and L_2, that are provided as client-specific logs by a composite log service component L (in the picture, components are depicted as circles, and an arrow from component X to component Y can be read as X *contains* Y, or Y *is a subcomponent of* X). The log service L is shared among the two subcomponents C_1 and C_2. One can argue that each private log component L_i $(i = 1, 2)$ is participating to three different ensembles, C_i, C and L. C_i, because L_i is existentially dependent on C_i, and is partially encapsulated in C_i as only updates originating from C_i are possible (it is a partial encapsulation because not all communication between a component L_i and its environment is mediated or controlled by C_i). C because each C_i is a subcomponent of C, and C is supposed to encapsulate its subcomponents. L, because L_i is existentially dependent on L, has the same lifetime as L (if L is deleted so are L_1 and L_2), and relies on private functions (e.g. data storage) provided by L. These ensembles also correspond to encapsulation scopes, whose meaning is, roughly, that no communication outside the scope is possible without explicitly passing through the top component of each scope (C, L, C_1 or C_2), except for the communications between C_i and L_i.

Related work. Over the past three decades, an abundant literature has developed that aims at formally modeling distributed, component-based, dynamic and adaptive software architectures, systems and services. One can cite notably: process calculi for distributed systems, such as π-calculi with localities [12, 22, 31], Ambient Calculi and their variants [10, 11], and Milner's bigraphs [28, 35]; process calculi and formal models for service-oriented computing and adaptive systems [9, 17, 19, 20, 37]; formal software component models such as BIP [6], Ptolemy [18], Reo [23], CommUnity [40], and several others [24, 25]; formal software architecture description languages [26, 32]. However, to the best of our

knowledge, none of these previous works provide satisfactory support to model architectures featuring a combination of dynamicity, sharing and encapsulation. Synchronized Hyperedge Replacement (SHR) systems and location graphs constitute a direct inspiration for the work in this paper, but they do not allow the definition of encapsulation scopes with sharing. Ownership types allow the enforcement of encapsulation scopes in object-oriented programs, typically at the expense of restrictions in inter-object communication, but do not allow overlapping encapsulation scopes. Bigraphs with sharing [35] support the definition of nodes with overlapping containment scopes, but, as far as we are aware, it is not possible to use bigraph nodes to enforce the encapsulation policies considered in this paper. The Fractal component model [8] is one of the rare software component models that allows the description of component configurations with sharing, and that has been formally defined [27]. The architecture in Figure 1 can readily be described in Fractal, with encapsulation scopes captured by Fractal composites. But we do not have a formal operational semantics for Fractal that would allow us to define the exact semantics of these scopes, nor do we have a proper behavioral theory for Fractal architectures. Interesting approaches to enforcing encapsulation policies are works that rely on dynamic access protection instead of aliasing control. These include notably the Siaam actor abstract machine [15], which relies on runtime checks to enforce actor encapsulation in a Java virtual machine, and access contracts [39] which provide dynamic access protection to Java objects that can support a wide range of encapsulation policies, including encapsulation policies with sharing as in the small architecture depicted in Figure 1. However, Siaam and access contracts do not come with a formal behavioral theory, and the question of program equivalence in concurrent languages with Siaam-like or access contracts-like access protection mechanisms remains open.

Contributions. In this paper, we combine ideas from SHR systems [19], from location graphs [36], as well as Siaam [15] and access contracts [39] dynamic approaches to encasulation enforcement, to define a formal operational framework, called the *Hypercell framework*. This framework allows the definition of different software component models (*hypercell models*), that support the modelling of dynamic component ensembles (*hypercells*) with sharing and encapsulation. The Hypercell framework can be seen as a conservative extension and generalization of the BIP and Fractal components models [6, 8]. A main contribution of the Hypercell framework is how it handles encapsulation policies: to allow for maximum flexibility, they are enforced by runtime checks (*authorizations*), that prevent transitions of component ensembles that would violate the chosen policies. Defining a proper notion of authorization is not trivial however. In order to obtain a proper component theory (e.g. in the sense of [4]), we need a notion of hypercell equivalence that is a congruence for hypercell composition, and it is not clear how such a result can be obtained in presence of authorizations. The idea is to have authorizations operate only at the level of individual components (*cells*): this allows us to define a notion of hypercell bisimilarity where we can decouple the contribution of authorizations from the classical bisimulation game,

which in turn allows us to obtain the required congruence result. However, this local form of authorization raises another problem: encapsulation policies are not local in nature, so how can we enforce them via such local checks, notably in presence of evolving component ensembles? We show, by means of examples, how this can be done via a combination of local but context-dependent authorization predicates and dynamic component types (*cell sorts*).

Outline. The paper is organized as follows. Section 2 is a brief informal introduction to hypercells. Section 3 presents the hypercell framework, and preliminary elements of a behavioral theory for hypercell models. Section 4 shows, by means of examples, how to enforce different forms of encapsulation using sorts and authorizations. Section 5 concludes the paper.

2 Informal introduction

A hypercell is a finite set of *cells*. A cell has (we also say "offers") *roles* (a term we borrow from location graphs). A *role* is a point of attachment for cells, as well as a point of interaction between cells. A hypercell, much like a SHR system, constitutes a hypergraph, where the roles are vertices, and cells are the edges of the hypergraph. A role corresponds to a point of attachment and interaction between cells, and may be offered at most by two distinct cells. Hypercells are thus limited forms of hypergraphs, where hyperedges can connect any number of vertices, but a given vertex can only be connected by at most two edges. As in standard software component model ontology [16], roles are classified as *provided* or *required*: a provided role in a cell signals some service offered by the cell, whereas a required role signals some expected service. When a role belongs to two cells in a given hypercell (in required position in one cell, and in provided position in the other cell), we say that the role is *bound*, and that it *binds* the two cells that offer it. Otherwise, we say that the role is *unbound*.

A cell is a locus of computation, as are localities in process calculi such as the Distributed π-calculus [31], and Klaim [29]. One can understand a cell as a basic software component or as a connector, as in the component-and-connector view of software architecture [3] and in software component models [16]. A hypercell can be understood as a composite software component or component ensemble. In this sense, the hypercell concepts align well with the standard concepts of software component models [16] and software architecture, as present e.g. in the ACME [21] and Fractal [8] component models (cells and hypercells correspond to ACME and Fractal *components*, roles to ACME *ports* and Fractal *interfaces*, bound roles to Fractal *primitive bindings*).

Figure 1 depicts a small hypercell with roles drawn as black dots (or arrows when bound) and cells as ellipses. Interactions in a hypercell take the form of simple point-to-point bidirectional interactions between pairs of cells bound by some role. In Figure 1, cells C_1 and L_1 can interact directly because they are bound, but C_2 cannot directly interact with C_1, nor C with L. In the architecture depicted in Figure 1, the scopes discussed in the introduction are manifested by bound roles and the sorts adorning the different cells (hinted at by arrows).

The behavior of a hypercell is the result of the composition of the behavior of its cells. A cell evolves by transforming into some hypercell, in the process possibly interacting with, or removing, cells it is bound to. The evolution of a hypercell corresponds to the parallel firing of a number of such cell transitions. An interaction between two cells bound at a role r amounts to several binary rendez-vous on communication channels succeeding at role r. An interaction will typically result in the simultaneous exchange of values at each of the channels participating in the interaction.

For instance, a client-server interaction at role r between a client cell C and a server cell S, may, on the server side, take the form $\overline{r} : \{\mathtt{op}\langle v, \mathtt{resp}\rangle, \overline{\mathtt{resp}}\langle w\rangle\}$, where r is the role which appears in provided position in the server (hence the overline $\overline{r}$), $\mathtt{op}$ is the channel on which the value v is sent, along with the return channel $\mathtt{resp}$, and w is the value which is (instantly) returned by the server, in response to the request $\mathtt{op}\langle v, \mathtt{resp}\rangle$, on the requested return channel $\mathtt{resp}$. On the client side, the interaction would take the conjugate form $r : \{\overline{\mathtt{op}}\langle v, \mathtt{resp}\rangle, \mathtt{resp}\langle w\rangle\}$. Notice that we use an early form for interactions: this allows us, in the operational semantics of hypercell models, to abstract from syntactic details such as a distinction between sent values and receiving parameters (the latter typically under the scope of some binding constructs). Our operational semantics thus has no mention of substitution of values to formal parameters, but we do distinguish with channels between originating side (e.g. $\overline{\mathtt{op}}$, on the client side) and receiving side (e.g. $\mathtt{op}$, on the server side).

Interactions between cells can be higher-order. In particular cells can be exchanged as values on channels during interactions. This allows the removal or passivation of cells as in the Kell calculus [34], which in turns allows to model objective reconfigurations in software architectures, where certain components can exercize explicit control over other ones.

Interactions between bound cells in a hypercell can be guarded by priorities. Priorities are crucial for the expressive power of the framework and the definition of different forms of composition operators as cells or hypercells. A priority allows a cell to check for the presence or absence of a signal from another cell it is bound to, in the form of the ability or inability to communicate on a given channel. For instance the client side communication above could be guarded by the absence of communication on channel $\mathtt{sig}$ on role s, which we would write thus: $\langle\{s : \neg\overline{\mathtt{sig}}\} \cdot \overline{r} : \{\mathtt{op}\langle v, \mathtt{resp}\rangle, \overline{\mathtt{resp}}\langle w\rangle\}\rangle$. In effect, the possible emission of signal $\mathtt{sig}$ on role s preempts (takes priority over) the emission of the request $\mathtt{op}$ on role r.

Individual cell transitions are also guarded by authorizations. An authorization is a runtime check that determines whether a cell transition is licit or not. Authorizations rely on the hypercell context of an individual cell to make this determination. For instance, a cell within an encapsulation scope can be prevented from making a transition that would allow it to bind to a cell outside this scope, whereas the same transition of the cell outside of such a scope can be allowed to proceed.

3 The hypercell framework

We define in this section our Hypercell framework. This framework can be instantiated to yield different hypercell models. Each hypercell model must define the following sets: a set $\mathbb{P}$ of *processes*; a set $\mathbb{S}$ of *sorts*; an infinite set $\mathbb{R}$ of *roles*; a set $\mathbb{V}$ of values; an infinite set $\mathbb{A}$ of *names*; an infinite set $\mathbb{Ch}$ of *channels*; a set $\mathcal{T}_u$ of *unconstrained transitions*; and an authorization predicate `Auth`. We require $\mathbb{R} \subset \mathbb{A}$, $\mathbb{Ch} \subset \mathbb{A}$, and that the sets $\mathbb{P}$, $\mathbb{S}$, and $\mathbb{A}$ be mutually disjoint. Values can comprise processes, sorts, and names as well as elements of other datatypes (booleans, integers, etc). Values can be exchanged between cells on channels at bound roles. We require that the set $\mathbb{Ch}$ contain the special channel `rmv`, which is used in hypercell models with objective cell removal. We require the set of names $\mathbb{A}$ to be equipped with an involution, called the conjugate operation, which sends a name a to its conjugate $\overline{a}$. By definition, we have $\overline{\overline{a}} = a$, and we write $\widehat{a}$ to denote a or its conjugate $\overline{a}$.

We require that each of the sets above be equipped with an operation for swapping names: for any element x of the above sets, $(r\ s) \cdot x$ yields an element of the same set where names r and s have been permuted, i.e. where r is replaced by s. (in the long version of this paper, we require the datatypes above to be nominal sets [30], but for lack of space we do not go into details here). We also require the existence of an operation `supp` that extracts from an element the set of names it contains, and we write $a\#X$ for $a \notin \mathtt{supp}(X)$.

Formally, a cell in a hypercell model is a 4-tuple of the form $[P : \mathfrak{s} \triangleleft \mathbf{p} \bullet \mathbf{r}]$, where P is the process of the cell, $\mathfrak{s}$ is the sort of the cell, $\mathbf{p}$ and $\mathbf{r}$ are the sets of provided and required roles of the cell, respectively. If $C = [P : \mathfrak{s} \triangleleft \mathbf{p} \bullet \mathbf{r}]$, we have $C.\mathtt{process} = P$, $C.\mathtt{sort} = \mathfrak{s}$, $C.\mathtt{prov} = \mathbf{p}$, and $C.\mathtt{req} = \mathbf{r}$. Any cell C must meet the following constraints: $C.\mathtt{prov} \cap C.\mathtt{req} = \emptyset$. The set of cells in a hypercell model is noted $\mathbb{C}$. The process of a cell embodies its behavior; the fact that a process can be a value means that cells can potentially update their behavior dynamically. The sort of a cell is a dynamic type associated with the cell; sorts are used to enforce runtime constraints on cells, as is shown in Section 4.

A hypercell G is just a set of cells that meets the following constraints: for any partition G_1, G_2 of G ($G = G_1 \cup G_2$ and $G_1 \cap G_2 = \emptyset$), one must have $G_1.\mathtt{prov} \cap G_2.\mathtt{prov} = \emptyset$ and $G_1.\mathtt{req} \cap G_2.\mathtt{req} = \emptyset$, where the set $G.\mathtt{prov}$ of provided roles of hypercell G is defined as $\bigcup_{C \in G} C.\mathtt{prov}$ (and likewise for the set $G.\mathtt{req}$ of required roles of G). We note $\mathtt{0}$ the empty hypercell, and $\mathbb{H}$ the set of hypercells in a hypercell model. We define the set of roles, of bound roles and unbound roles of a hypercell G:

$$G.\mathtt{roles} \triangleq G.\mathtt{prov} \cup G.\mathtt{req} \qquad G.\mathtt{bound} \triangleq G.\mathtt{prov} \cap G.\mathtt{req}$$
$$G.\mathtt{unbound} \triangleq G.\mathtt{roles} \setminus G.\mathtt{bound}$$

When G and G' are two disjoint hypercells, we write $G \parallel G'$ to denote $G \cup G'$ when $G \cup G'$ is indeed a hypercell (i.e. a set of cells meeting the above constraints).

3.1 Operational semantics of a hypercell model

The operational semantics of a hypercell model is defined as a set $\mathcal{T}$ of labelled (contextual) transitions. A transition is an element of $\mathbb{T} = \mathbb{E} \times \mathbb{H} \times \mathbb{\Lambda} \times \mathbb{H}$, where $\mathbb{E}$ is the set of *environments*, and $\mathbb{\Lambda}$ is the set of *labels*. A transition $t = \langle \Gamma, G, \Lambda, G' \rangle \in \mathbb{T}$ is noted $\Gamma \vdash G \xrightarrow{\Lambda} G'$, with $\Gamma \in \mathbb{E}$ the environment $t.\texttt{env}$ of the transition, $G \in \mathbb{H}$ the initial hypercell $t.\texttt{init}$ of the transition, $\Lambda \in \mathbb{\Lambda}$ the label $t.\texttt{label}$ of the transition, and $G' \in \mathbb{H}$ the final hypercell $t.\texttt{final}$ of the transition. Intuitively, if $\Gamma \vdash G \xrightarrow{\Lambda} G'$, then hypercell G, when placed in environment Γ, can evolve into hypercell G' provided the synchronizations in label Λ are met. The environment in a transition represents both the set of known names prior to the transition, and the hypercell context in which the initial hypercell of the transition is placed.

A *label* Λ is a pair $\langle \pi \cdot \sigma \rangle$, where π is a finite set of priorities, and σ is a finite set of interactions. We note ϵ the empty set of priorities or interactions. and we set $\langle \pi \cdot \sigma \rangle.\texttt{prior} = \pi$, $\langle \pi \cdot \sigma \rangle.\texttt{sync} = \sigma$.

An *interaction* corresponds to an exchange of a value V on a channel c at a role r. An interaction takes the form $r : \widehat{c}\langle V \rangle$ if the role r is provided, and $\overline{r} : \widehat{c}\langle V \rangle$ if the role is required. An interaction $\widehat{r} : c\langle V \rangle$ corresponds to a receipt on channel c at role r of value V, whereas an interaction $\widehat{r} : \overline{c}\langle V \rangle$ corresponds to the emission of value V on channel c at role r. An interaction $\widehat{r} : \widehat{c}\langle V \rangle$ succeeds when matched with its conjugate interaction $\overline{\widehat{r}} : \overline{\widehat{c}}\langle V \rangle$. Notice that the value V in a successful interaction must be the same on both emitter and receiver sides. For this reason, our presentation of an hypercell model transition relation can be said to follow an *early style* [33]. This allows us in the presentation of the hypercell framework to abstract away from syntactic details of interactions in hypercell models. We set $(r : \widehat{c}\langle V \rangle).\texttt{prov} = \{r\}$, $(r : \widehat{c}\langle V \rangle).\texttt{req} = \emptyset$, and the dual for $\overline{r} : \widehat{c}\langle V \rangle$. We set $(\widehat{r} : \widehat{c}\langle V \rangle).\texttt{roles} = \{r\}$ and $(\widehat{r} : \widehat{c}\langle V \rangle).\texttt{channels} = \{c\}$. The set of interactions in a hypercell model is noted $\mathbb{I}$.

A *priority* takes the following form: $\widehat{r} : \neg c$, where r is a role and c is a channel. Intuitively, a contraint $\widehat{r} : \neg c$ stipulates that the cell bound at role r is not ready to perform an interaction on channel c. The set of priorities is noted $\mathbb{\Pi}$. Priorities are inherited from location graphs and provide hypercell models with significant expressive power (see Proposition 1 below). We set $(\widehat{r} : \neg c).\texttt{roles} = \{r\}$.

An *environment* Γ is a pair $\Delta \cdot \Sigma$ comprising a set of known *names* (roles or channels) $\Delta \subseteq \mathbb{A} = \mathbb{R} \cup \mathbb{Ch}$, and a *skeleton hypercell* (or *skeleton*, for brevity) Σ. For $\Gamma = \Delta \cdot \Sigma$ we define $\Gamma.\texttt{names} = \Delta$ and $\Gamma.\texttt{graph} = \Sigma$. The set of known names in an environment corresponds intuitively to the set of already generated names during a hypercell execution. New names created in a transition are names that do not belong to this set. The skeleton in an environment gathers information about the hypercell that surrounds the initial hypercell in a transition. It is used in determining authorizations for individual cell transitions (see rule Trans below). A skeleton cell is a triplet $[\mathfrak{s} \triangleleft \mathbf{p} \bullet \mathbf{r}]$. The set of skeleton cells in a hypercell model is noted $\mathbb{C}_s$. The set of skeleton hypercells in a hypercell model is noted $\mathbb{H}_s$. Essentially, a skeleton is a hypercell where one has erased all the

$$\textsc{Trans}\quad \frac{\Gamma.\mathtt{names}\cdot \mathsf{0} \triangleright C \xrightarrow{\Lambda} G \qquad \Sigma(C) \in \Gamma.\mathtt{graph} \qquad \mathtt{Auth}(\Gamma, C, \Lambda, G)}{\Gamma \vdash C \xrightarrow{\Lambda} G}$$

$$(\textsc{Comp})\quad \frac{\begin{array}{c}\Gamma \vdash G_1 \xrightarrow{\langle \pi_1 \cdot \sigma_1 \rangle} G_1' \\ \Gamma \vdash G_2 \xrightarrow{\langle \pi_2 \cdot \sigma_2 \rangle} G_2' \qquad \mathtt{Cond}_P(s, \pi, \pi_1, \pi_2, \Gamma, C_1, C_2) \\ \mathtt{Cond}_I(\sigma, \sigma_1, \sigma_2, G_1 \parallel G_2) \qquad \mathtt{Cond}(\Gamma, G_1 \parallel G_2)\end{array}}{\Gamma \vdash G_1 \parallel G_2 \xrightarrow{\langle \pi \cdot \sigma \rangle} G_1' \parallel G_2'}$$

$$(\textsc{Ctx})\quad \frac{\begin{array}{c}\Gamma \vdash G \xrightarrow{\langle \varpi \cdot \sigma \rangle} G' \\ \mathtt{Ind}_P(s, \pi, \varpi, \Gamma, C, E) \qquad \mathtt{Ind}_I(\sigma, E) \qquad \mathtt{Cond}(\Gamma, G \parallel E, G' \parallel E)\end{array}}{\Gamma \vdash G \parallel E \xrightarrow{\langle \pi \cdot \sigma \rangle} G' \parallel E}$$

Fig. 2. Transition rules for a hypercell model

processes. The skeleton $\Sigma(G)$ of a hypercell G is defined inductively as follows:

$$\Sigma(\mathsf{0}) = \mathsf{0} \quad \Sigma([P : \mathfrak{s} \lhd \mathbf{p} \bullet \mathbf{r}]) = [\mathfrak{s} \lhd \mathbf{p} \bullet \mathbf{r}] \quad \Sigma(G_1 \cup G_2) = \Sigma(G_1) \cup \Sigma(G_2)$$

We denote by $\mathsf{0}$ the empty skeleton, and by $\mathbb{E}$ the set of environments in a hypercell model. We define $\Delta_1 \cdot \Sigma_1 \subseteq \Delta_2 \cdot \Sigma_2 \triangleq \Delta_1 \subseteq \Delta_2 \wedge \Sigma_1 \subseteq \Sigma_2$, and $\Delta_1 \cdot \Sigma_1 \cup \Delta_2 \cdot \Sigma_2 \triangleq \Delta_1 \cup \Delta_2 \cdot \Sigma_1 \cup \Sigma_2$.

An hypercell model must define the set $\mathcal{T}_u$ of unconstrained transitions of its individual cells, i.e. transitions that do not rely on any knowledge of the execution context of individual cells. This fits with the idea that software components can be reused in different contexts (in our case, hypercells), and that their behavior should be defined as independently as possible from their context of use. We write $\Gamma \triangleright C \xrightarrow{\Lambda} G$ for $\langle \Gamma, C, \Lambda, G \rangle \in \mathcal{T}_u$. Environments in unconstrained transitions are of the form $\Delta \cdot \mathsf{0}$. An unconstrained transition $\Delta \cdot \mathsf{0} \triangleright C \xrightarrow{\Lambda} G$ for an individual cell C must obey the following conditions: (i) names in the support of C must be known names, i.e. names in Δ: $\mathtt{supp}(C) \subseteq \Delta$; (ii) interactions and priorities in $\Lambda = \langle \pi \cdot \sigma \rangle$ must be offered at roles from C: $\sigma.\mathtt{prov} \subseteq C.\mathtt{prov} \wedge \sigma.\mathtt{req} \subseteq C.\mathtt{req}$ and $\pi.\mathtt{roles} \subseteq C.\mathtt{roles}$. In addition, we require $\mathcal{T}_u$ to be insensitive to name changes, namely: $\forall t \in \mathcal{T}_u, n, m \in \mathbb{A},\ (n\ m) \cdot t \in \mathcal{T}_u$. An hypercell model can define cells that allow their removal by other cells they are bound to. A cell C that allows its removal on some role r must provide an unconstrained transition of the form $\Delta \cdot \mathsf{0} \triangleright C \xrightarrow{\langle \epsilon \cdot \{\widehat{r}:\overline{\mathtt{rmv}}\langle C \rangle\} \rangle} \mathsf{0}$.

A hypercell model must define an authorization predicate $\mathtt{Auth}$. Predicate $\mathtt{Auth} \subseteq \mathbb{E} \times \mathbb{C} \times \mathbb{\Lambda} \times \mathbb{H}$ determines whether an individual cell transition is possible in a given context (a surrounding hypercell). We require $\mathtt{Auth}$ to be name insensitive, namely: for all $n, m \in \mathbb{A}$, and cell transition $t \in \mathbb{E} \times \mathbb{C} \times \mathbb{\Lambda} \times \mathbb{H}$, we have $\mathtt{Auth}(t) \iff \mathtt{Auth}((n\ m) \cdot t)$.

Using terminology from [38], the operational semantics of a hypercell model is defined as the set of transitions $\mathcal{T} \subseteq \mathbb{T}$ that is the least well-suported model of the rules in Figure 2.

Rule TRANS turns an unconstrained transition into a regular transition, provided that it be authorized in the current context.

The predicates $\texttt{Cond}, \texttt{Cond}_I, \texttt{Ind}_I$ in the premises of rules COMP and CTX are defined as follows:

$$
\begin{array}{l}
\texttt{Cond}(\Gamma, G) \triangleq \texttt{supp}(G) \cdot \Sigma(G) \subseteq \Gamma \\
\texttt{Cond}_I(\sigma, \sigma_1, \sigma_2, G) \triangleq \sigma = \texttt{seval}(\sigma_1 \cup \sigma_2) \ \wedge \ \sigma.\texttt{roles} \subseteq G.\texttt{unbound} \\
\texttt{Ind}_I(\sigma, E) \triangleq \sigma.\texttt{roles} \cap E.\texttt{roles} = \emptyset \\
\texttt{seval}(\sigma) = \ \mathsf{if}\ \sigma = \{\widehat{r} : \widehat{c}\langle V \rangle, \overline{\widehat{r}} : \overline{\widehat{c}}\langle V \rangle\} \ \cup \ \sigma' \ \mathsf{then}\ \texttt{seval}(\sigma')\ \mathsf{else}\ \sigma
\end{array}
$$

If C is a hypercell with $r \in C.\texttt{roles}$, such that there is a single cell $L \in C$ with $r \in L.\texttt{roles}$, then we note $(\!|C|\!)^s_r$ the hypercell $(s\ r) \cdot C$. For $\rho = \widehat{r} : \neg a$, we define $\rho.\texttt{r} = r$. We say that hypercell C, in environment Γ, satisfies the priority constraint $\rho = \widehat{r} : \neg a$, noted $C \models_\Gamma \rho$, if the following conditions hold:

$$
\begin{array}{l}
\Sigma(C) \subseteq \Gamma.\texttt{graph} \ \wedge \ r \in C.\texttt{unbound} \\
\neg(\exists D \in \mathbb{H}, V \in \mathbb{V}, \Lambda \in \mathbb{\Lambda}, \ \Gamma \vdash C \xrightarrow{\Lambda} D \ \wedge \ \overline{\widehat{r}} : \widehat{a}\langle V \rangle \in \Lambda.\texttt{sync})
\end{array}
$$

The predicates $\texttt{Cond}_P$ and $\texttt{Ind}_P$ in the premises of the rules COMP and CTX are defined as follows:

$$
\begin{array}{rl}
\texttt{Cond}_P(s, \pi, \pi_1, \pi_2, \Gamma, C_1, C_2) \triangleq & s \# \Gamma.\texttt{names} \\
& \wedge \ \pi = \{\rho \in \pi_1 \cup \pi_2 \mid \rho.\texttt{r} \in (C_1 \parallel C_2).\texttt{unbound}\} \\
& \wedge \bigwedge_{\rho \in \pi_1 \setminus \pi} (\!|C_1|\!)^s_{\rho.\texttt{r}} \parallel C_2 \models_\Gamma \rho \ \wedge \bigwedge_{\rho \in \pi_2 \setminus \pi} C_1 \parallel (\!|C_2|\!)^s_{\rho.\texttt{r}} \models_\Gamma \rho \\
\texttt{Ind}_P(s, \pi, \varpi, \Gamma, C, E) \triangleq & s \# \Gamma.\texttt{names} \\
& \wedge \ \pi = \{\rho \in \varpi \mid \rho.\texttt{r} \in (C \parallel E).\texttt{unbound}\} \\
& \wedge \bigwedge_{\rho \in \varpi \setminus \pi} (\!|C|\!)^s_{\rho.\texttt{r}} \parallel E \models_\Gamma \rho
\end{array}
$$

The predicate $\texttt{Cond}_P$ expresses the fact that priorities that appear on roles that bind the hypercells C_1 and C_2 together must be verified. Priorities on roles that bind hypercells C_1 and C_2 are exactly those constraints ρ in the set $(\pi_1 \setminus \pi) \cup (\pi_2 \setminus \pi)$, where π is the set of priorities that appear on roles not bound in $C_1 \parallel C_2$ (since priorities that appear in a transition of a hypercell C are expected to adorn unbound roles in C). To check whether a priority ρ is satisfied, one considers a variant of configuration $C_1 \parallel C_2$ where the role $\rho.\texttt{r}$, in the hypercell from which the priority originates, is replaced by a fresh role. In effect, this replacement amounts to severing the binding $\rho.\texttt{r}$ between C_1 and C_2.

Remark 1. The definition of satisfaction for a priority by a hypercell is not entirely trivial because of cycles of constraints that may occur. As a sanity check, consider the two following examples, depicted in Figure 3.

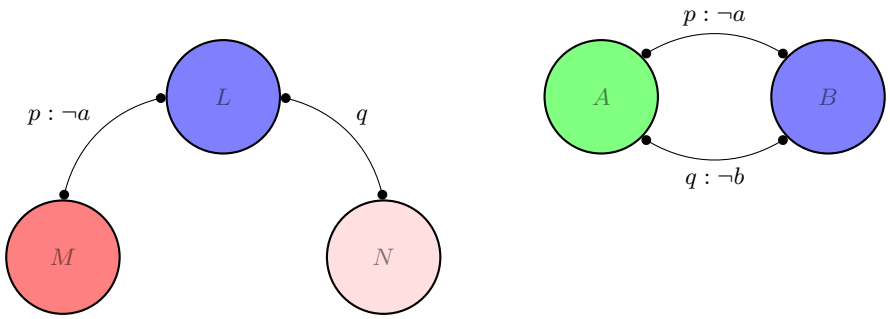

Fig. 3. Two hypercells with priorities

On the left, we are considering a hypercell $M \parallel L \parallel N$, and a transition from L of the form $\Gamma \vdash L \xrightarrow{\langle p:\neg a \cdot q:b\rangle} C$, where L can interact with N on channel b, provided M is not able to interact on a. Verifying the satisfaction of the priority on role p consists in checking whether $M \parallel (\!|L|\!)^s_p \parallel N$, where s is fresh, can interact on channel a on role p, which amounts to check that M can interact on channel a on role p.

On the right, we are considering a hypercell $A \parallel B$, with the following transitions:

$$t_A = \Gamma \vdash A \xrightarrow{\langle q:\neg b \cdot \{p:c,q:c\}\rangle} A' \qquad t'_A = \Gamma \vdash A \xrightarrow{\langle \epsilon \cdot p:a\rangle} A'$$

$$t_B = \Gamma \vdash B \xrightarrow{\langle p:\neg a \cdot \{p:c,q:c\}\rangle} B' \qquad t'_B = \Gamma \vdash B \xrightarrow{\langle \epsilon \cdot q:b\rangle} B'$$

In other terms, A can interact on channel c on roles p and q, provided B cannot interact on channel b on role q, and B can interact on channel c on roles p and q, provided A cannot interact on a on role p. To verify the satisfaction of the priority from A on role q, we have to check whether the graph $(\!|A|\!)^s_q \parallel B$, where s is fresh, can interact on channel b on role q, which amounts to check that B can interact on channel b on role q. This is the case because of transition t'_B. The priority from A on role q is thus not verified, and transition t_A cannot fire in this configuration. Likewise, the priority from B on role p is not satisfied and transition t_B cannot fire in this configuration.

In both examples, our rules give results that match the intuition: in the first case, we expect the priority on p to be satisfied merely if M cannot interact on channel a on role p, and in the second case we expect the hypercell $A \parallel B$ to deadlock.

Rule Comp stipulates that a hypercell $G_1 \parallel G_2$ can evolve by combining a transition from G_1 and a transition from G_2. The combination involves synchronizing interactions on roles that bind G_1 and G_2 (condition $\texttt{Cond}_I$) and verifying priorities on the roles that bind G_1 and G_2 (condition $\texttt{Cond}_P$). Rule Ctx stipulates that in a hypercell $G \parallel E$, hypercell G can evolve independently of E, provided G's interactions and priorities do not involve roles from E (conditions $\texttt{Ind}_I$, $\texttt{Ind}_P$). Notice that both rules Comp and Ctx require the results of the transitions in their conclusion ($G'_1 \parallel G'_2$ and $G' \parallel E$) to be hypercells. Note that both rules are stratified by the number of bound roles in a hypercell: the number of bound roles in $C_1 \parallel C_2$ is one less than in $(\!|C_1|\!)^s_{\rho.\mathtt{r}} \parallel C_2$.

Notice that, in contrast to other process calculi frameworks such as the ψ-calculus [5] and SHR systems [19], hypercell models do not have a restriction operator à la π-calculus. In hypercells, events taking place at a role binding two cells are not visible outside of the two cells. This hiding provided by bound roles is actually enough to encode restriction as in the π-calculus. Our handling of name creation via environments is also unusual, again compared to the use of a restriction operator à la π-calculus. It is related to the nominal presentation of the π-calculus in [13], but relies on name insensitivity instead of α-conversion. This is no way a limitation on the expressive power of the hypercell framework for the restriction operator, as well as any other composition operator definable by means of GSOS rules, i.e. structured operational semantics rules obeying the general format defined in [7]. More generally we can prove that any GSOS language (as defined in [7]) can be encoded as a hypercell model:

Proposition 1. *For any GSOS language $\mathcal{L}$, there exists a hypercell model and an encoding $[\![\cdot]\!] : \mathcal{L} \times \mathbb{R} \to \mathcal{P}(\mathbb{H})$, such that for any $P, Q \in \mathcal{L}, a \in \mathcal{A}, u \in \mathbb{R}$, we have $P \xrightarrow{a} Q$ if and only if there exist $\Delta \subseteq \mathbb{A}$, $\Lambda \in \mathbb{\Lambda}$ with $u : a \in \Lambda.\mathtt{sync}$, $C \in [\![P]\!]_u$, and $D \in [\![Q]\!]_u$, such that $\Delta \vdash C \xrightarrow{\Lambda} D$.*

Similarly, we can prove that the π-calculus can be encoded as a hypercell model.

3.2 Behavioral equivalence for hypercell models

We define in this section a strong notion of behavioral equivalence for hypercell models, in the form of a bisimilarity relation.

Definition 1 (Environment equivalence). *Two environments Γ, Γ' are said to be equivalent, noted $\Gamma \backsimeq \Gamma'$, if for all $\Upsilon \in \mathbb{E}$ such that $\Gamma \cup \Upsilon \in \mathbb{E}$ and $\Gamma' \cup \Upsilon \in \mathbb{E}$, for all $C \in \mathbb{C}$, $\Lambda \in \mathbb{\Lambda}$, $G \in \mathbb{H}$, we have $\mathtt{Auth}(\Gamma \cup \Upsilon, C, \Lambda, G) = \mathtt{Auth}(\Gamma' \cup \Upsilon, C, \Lambda, G)$. Two hypercells G and F are said to be environmentally equivalent, also noted $G \backsimeq F$, if $\mathtt{supp}(G) \cdot \Sigma(G) \backsimeq \mathtt{supp}(F) \cdot \Sigma(F)$.*

Definition 2 (Strong simulation). *A name insensitive binary relation on hypercells $\mathcal{R} \subseteq \mathbb{H} \times \mathbb{H}$ is a strong simulation if, for all $\langle G, F \rangle \in \mathcal{R}, G'' \in \mathbb{H}$, $\Lambda \in \mathbb{\Lambda}$, the following properties hold:*

1. *$G \backsimeq F$ and the unbound provided (resp. required) roles of G and F coincide.*
2. *For all $\Gamma \in \mathbb{E}$ such that $\Gamma \cup \Sigma(G) \in \mathbb{E}$, $\Gamma \cup \Sigma(F) \in \mathbb{E}$, if $\Gamma \cup \Sigma(G) \vdash G \xrightarrow{\Lambda} G'$, then there exists $F' \in \mathbb{H}$ such that $\Gamma \cup \Sigma(F) \vdash D \xrightarrow{\Lambda} F'$ with $\langle G', F' \rangle \in \mathcal{R}$.*

The main difference compared to the usual notion of strong simulation on labelled transition systems is the quantification on environments, which is necessary to take into account the effect of authorization functions. Note also that we require that a transition be simulated by a transition with the exact same label. This is a strong requirement but which can only be relaxed if one knows more about actions hypercells can take on values (e.g. if processes can only be exchanged and run – placed in a cell –, one may require only that they be similar, as in higher-order simulations).

Definition 3 (Strong bisimulation and bisimilarity). *A binary relation* $\mathcal{R} \subseteq \mathbb{H}^2$ *is a strong bisimulation if both it and its inverse relation* $\mathcal{R}^{-1}$ *are strong simulations.*

Strong bisimilarity, noted $\sim$*, is defined by* $\sim \triangleq \bigcup_{\mathcal{R} \in \mathbb{S}} \mathcal{R}$*, where* $\mathbb{S} \subseteq \mathcal{P}(\mathbb{H}^2)$ *is the set of all strong bisimulations.*

Crucially, in any hypercell model strong bisimilarity is a congruence (meaning our notion of bisimilarity is a reasonable notion of behavioral equivalence for hypercells):

Theorem 1. *In any hypercell model, for all* $G, F \in \mathbb{H}$*, if* $G \sim F$*, then for all* $E \in \mathbb{H}$ *such that* $G \parallel E \in \mathbb{H}$ *and* $F \parallel E \in \mathbb{H}$*, we have* $G \parallel E \sim F \parallel E$*.*

The proof of this is left out for lack of space but it proceeds by showing, by induction on the maximum number of bound names in $G \parallel E$ and $F \parallel E$, that the relation $\mathcal{R} = \{(G \parallel E, F \parallel E \mid G \sim F)\}$ is a strong bisimulation.

4 Encapsulation policies

We show in this section how to enforce different encapsulation policies in hypercell models. Specifically, we present a form of strict encapsulation, inspired by *owner-as-dominator* policies studied in ownership types [14], and a weaker variant that allows software architectures with overlapping encapsulation scopes as in Figure 1. The challenge is of course to enforce these policies in the highly dynamic and concurrent setting of hypercell evolutions. Some notations first. For $L, M \in \mathbb{C} \cup \mathbb{C}_s$, we write $L \frown M$ to mean L and M are bound, i.e. $(L.\texttt{prov} \cap M.\texttt{req}) \cup (L.\texttt{req} \cap M.\texttt{prov}) \neq \emptyset$. For $F, G \in \mathbb{H} \cup \mathbb{H}_s$, we write $F \shortparallel G$ to mean $F.\texttt{roles} \cap G.\texttt{roles} = \emptyset$.

4.1 Strict encapsulation

In this form of encapsulation, cells come in three disjoint categories: *owner cells*, *owned cells*, and *free cells*. Owner cells can be understood as composite components. The cells they own – their owned cells – are their subcomponents. Free cells are neither owner nor owned. The encapsulation policy we consider here takes the form of a structural invariant which ensures an owned cell cannot directly interact with cells which do not belong to its owner's group - made by this owner cell and all its owned cells. For simplicity, we have only a single level of ownership (owner cells cannot be owned). It is relatively straightforward to extend this policy to allow multiple levels of ownerhsip.

To capture this, we consider a hypercell model (actually a class of models) with sorts that take the form of 4-tuples $\langle k, \mathbf{p}, \mathbf{o}, \mathbf{r} \rangle$, where $k \in \{\top, \bot\}$ is a flag and $\mathbf{p}, \mathbf{o}, \mathbf{r} \in \mathcal{P}_{\texttt{fin}}(\mathbb{R})$. We set: $\mathfrak{s}.\texttt{flag} = k$, $\mathfrak{s}.\texttt{fprov} = \mathbf{p}$, $\mathfrak{s}.\texttt{owned} = \mathbf{o}$, $\mathfrak{s}.\texttt{freq} = \mathbf{r}$, and write $C.\texttt{fprov}$ for $C.\texttt{sort.fprov}$, $C.\texttt{owned}$ for $C.\texttt{sort.owned}$, $C.\texttt{freq}$ for $C.\texttt{sort.freq}$, $C.\texttt{flag}$ for $C.\texttt{sort.flag}$. Flags in sorts are used to avoid race conditions in the parallel evolution of owner and owned cells in a

owner group, which would break the global structural invariant (for instance two owned cells being bound, while their owner is splitting itself in two).

A cell C in this model is assumed to maintain the following invariant:

$$C.\texttt{prov} = C.\texttt{fprov} \wedge C.\texttt{req} = C.\texttt{owned} \cup C.\texttt{freq} \wedge C.\texttt{owned} \cap C.\texttt{freq} = \emptyset \quad (1)$$

We also require to identify in a transition label Λ the roles that are sent by the initial cell in the transition. We note $\Lambda.\texttt{sent}$ the set of sent roles in Λ.

We define the following (these definitions apply to skeletons as well). For $L, M \in \mathbb{C}$, we write $L \multimap M$ for $L.\texttt{owned} \cap M.\texttt{prov} \neq \emptyset$ (intuitively, L owns M), and $M.\texttt{up}_L$ for $L.\texttt{owned} \cap M.\texttt{prov}$ when $L \multimap M$. If $G \in \mathbb{H}$, we write $L \multimap G$ to mean that, for all $M \in G$, $L \multimap M$. For $G \in \mathbb{H}, L \in \mathbb{C}$, we define $\texttt{scope}_G(L) = \{M \in G \mid L \multimap M\}$ (the set of cells owned by L), and $\texttt{group}_G(L) = \{L\} \cup \texttt{scope}_G(L)$. We write $\texttt{scope}_\Gamma(L)$ for $\texttt{scope}_{\Gamma.\texttt{graph}}(L)$. We drop the subscript G to write $\texttt{scope}(L)$ and $\texttt{group}(L)$ when the hypercell or skeleton context G is clear. An *owner* is a cell L such that $L.\texttt{owned} \neq \emptyset$ and we write $L\,\texttt{owner}$. We define: $G.\texttt{owners} \triangleq \{M \in G \mid M.\texttt{owned} \neq \emptyset\}$. An *owned* cell L in a hypercell G is a cell such that there exists $M \in G$ with $M \multimap L$. A *free* cell in a hypercell G is a cell which is neither owner nor owned.

The structural properties we expect are defined as follows. For any $G \in \mathbb{H} \cup \mathbb{H}_s$:

$$\forall L, M \in G,\ L \neq M \implies \texttt{scope}_G(L) \Vdash \texttt{scope}_G(M) \wedge L.\texttt{owned} \cap M.\texttt{owned} = \emptyset \quad (2)$$

$$\forall L, M \in G,\ L \multimap M \implies M.\texttt{owned} = \emptyset \quad (3)$$

$$\forall L, M, N \in G,\ L \multimap M \wedge M \frown N \implies L \multimap M \vee L = N \quad (4)$$

Property (2) states that the encapsulation scopes of two owners L and M in the same hypercell are necessarily distinct and they are bound by no role. Property (3) states that there is only a single level of ownership: an owner cannot be owned. Property (4) states that cells in the encapsulation scope of an owner can only be bound to cells in the same scope or to the owner itself. We write $\texttt{Inv}(G)$ if properties (2), (3), and (4) hold for G (hypercell or skeleton).

We assume the existence of a predicate $\texttt{New} \subseteq \mathcal{P}(\mathbb{A}) \times \mathcal{P}_{\texttt{fin}}(\mathbb{A}) \times \mathcal{P}_{\texttt{fin}}(\mathbb{A})$ with the following properties ($\texttt{New}$ can be defined constructively but we eschew this definition here for lack of space):

$$\texttt{New}(\Delta, A, B) \implies B \cap (\Delta \cup A) = \emptyset$$
$$\texttt{New}(\Delta, A, B) \wedge \texttt{New}(\Delta, A', B') \wedge A \neq A' \implies B \cap B' = \emptyset$$

We define the following predicates, which are used in the definition of the authorization predicate. Predicate $\texttt{Safe} \subseteq \mathbb{E} \times \mathbb{C} \times \mathbb{H}$ is such that $\texttt{Safe}(\Gamma, M, G)$ holds if roles of G are new roles or are roles already used in the scope of M in the context Γ. It is defined as follows:

$$\texttt{Safe}(\Gamma, M, G) \triangleq \exists B \in \mathcal{P}_{\texttt{fin}}(\mathbb{R}), \texttt{New}(\Gamma.\texttt{names}, H.\texttt{roles}, B) \wedge G.\texttt{roles} \subseteq B \cup \texttt{scope}_\Gamma(M).\texttt{roles}$$

Predicate $\texttt{Incl} \subseteq \mathbb{H}_s \times \mathbb{C}_s$ is such that $\texttt{Incl}(G, M)$ holds if M shares a role with a skeleton cell in G. It is defined as follows:

$$\texttt{Incl}(G, M) \triangleq M.\texttt{prov} \cap G.\texttt{prov} \neq \emptyset \ \vee \ M.\texttt{req} \cap G.\texttt{req} \neq \emptyset$$

We now define the authorization predicate for our class of hypercell models with strict encapsulation. Predicate $\texttt{Auth}$ is defined as follows. $\texttt{Auth}(\Gamma, L, \Lambda, G)$ is $\texttt{true}$ exactly in the cases below:

1. If $\exists M \in \Gamma,\ M \multimap \Sigma(L) \wedge M.\texttt{flag} = \top$, $G.\texttt{owned} = \emptyset \wedge M \multimap \Sigma(G) \wedge \texttt{Safe}(\Gamma, M, G)$, and $\Lambda.\texttt{sent} \subseteq \texttt{supp}(L)$. If the flag of its owner is up, an owned cell can reconfigure into an hypercell G provided all the cells in G remain owned by the same owner, and the roles of G are either existing roles of cells in the owner scope, or brand new ones. If a cell is owned, the only constraint on its transitions labels is that sent roles in a label be roles already known by L (i.e. new roles created during a transition cannot be immediately sent).
2. If $L\,\texttt{owner} \ \wedge\ L.\texttt{flag} = \bot$, $\texttt{Inv}(\texttt{H}(\Gamma, L, G)) \ \wedge\ \texttt{Osafe}(\Gamma, L, G)$ with:

$$\texttt{H}(\Gamma, L, G) \triangleq \Sigma(G) \cup (\texttt{scope}_{\Gamma}(L) \setminus \{M \in \texttt{scope}_{\Gamma}(L) \mid \texttt{Incl}(\Sigma(G), M)\})$$
$$\texttt{Osafe}(\Gamma, L, G) \triangleq \bigwedge_{K \in G.\texttt{owners}} \texttt{Safe}(\Gamma, L, \texttt{scope}_{\texttt{H}(\Gamma, L, G)}(K))$$

   and $\Lambda.\texttt{sent} \subseteq \texttt{supp}(L) \ \wedge\ \overline{\texttt{rmv}} \notin \Lambda.\texttt{sync.channels}$. If its flag is down, an owner L can reconfigure into an hypercell G, provided G and the cells in L's scope remaining after the transition (those such that $\texttt{incl}(\Sigma(G), M)$ have been removed) respect the global invariant $\texttt{Inv}$, and the roles in the scope of owners in G are either ones already in its scope, or brand new ones. If a cell is an owner, the same constraint as above on sent roles apply, but in addition it cannot be removed by any other cell.
3. If $L\,\texttt{owner} \wedge L.\texttt{flag} = \top$, $L.\texttt{owned} \subseteq G.\texttt{owned}$, $G \in \mathbb{C}$, and $\Lambda.\texttt{sent} \subseteq \texttt{supp}(L) \wedge \overline{\texttt{rmv}} \notin \Lambda.\texttt{sync.channels}$. If its flag is up, an owner can only change into a single owner cell, not losing any owned role, possibly adding some (e.g. to allow the reconfiguration of cells it owns).
4. If $L\,\texttt{free}$, $\texttt{Inv}(G) \wedge \texttt{Fsafe}(\Gamma, L, G)$ where:

$$\texttt{Fsafe}(\Gamma, G) \triangleq G.\texttt{roles} \cap \Gamma.\texttt{owned} = \emptyset \ \wedge \bigwedge_{K \in G.\texttt{owners}} \texttt{Safe}(\Gamma, L, \texttt{scope}_{G}(K))$$

   and $\Lambda.\texttt{sent} \subseteq \texttt{supp}(L)$. A free cell can reconfigure into a hypercell G provided it respects the global invariant $\texttt{Inv}$, it does not insert new cells in the scope of existing owners, and the roles of cells in the scope of new owners in G are safe. Also, since it is not an owner, new roles created during a transition cannot be immediately sent.

Note that, with the above definition of $\texttt{Auth}$, in an environment Γ where cell L is owned and the flag of its owner M is down, i.e. $\exists M \in \Gamma,\ M \multimap L \wedge M.\texttt{flag} = \bot$, then L cannot evolve in environment Γ.

The authorization predicate is quite permissive in the kinds of evolutions owner cells can perform. Notice in particular that owner cells may split or dissolve during execution, allowing e.g. for the transfer of owned cells from one owner to another. Likewise, owned cells can be freed by their owner and become owner cells later on. The dynamicity and concurrency in the class of hypercell models obeying strict encapsulation is much bigger than that allowed in the computational models underlying ownership types (either strictly sequential or actor like).

Predicate $\texttt{Inv}$ is indeed an invariant for the class of hypercell models equipped with these sorts and authorization functions:

Proposition 2 ($\texttt{Inv}$ is an invariant). *For all* $\Gamma \in \mathbb{E}, G, G' \in \mathbb{H}, \Lambda \in \mathbb{A}$, *if* $\texttt{Inv}(\Gamma)$, $\texttt{Inv}(G)$ *and* $\Gamma \vdash G \xrightarrow{\Lambda} G'$, *then* $\texttt{Inv}(G')$.

4.2 Selective encapsulation

We extend the strict encapsulation policy of the previous section with a notion of *weak ownership*. Briefly, owner scopes of strict encapsulation are now allowed to include weakly owned cells. A cell may belong to only one owner, as previously, but may also belong to several weak owners. A cell can be weakly owned only if it has identified specific provided roles for this purpose ($\texttt{wprov}$ roles below).

We extend sorts to 6-tuples $\langle k, \mathbf{p}, \mathbf{o}, \mathbf{r}, \mathbf{q}, \mathbf{w}\rangle$ with $\mathbf{q}, \mathbf{w} \in \mathcal{P}_{\texttt{fin}}(\mathbb{R})$. We set $\mathfrak{s}.\texttt{wowned} = \mathbf{w}$ and $\mathfrak{s}.\texttt{wprov} = \mathbf{q}$. We write

$C.\texttt{wowned}$ for $C.\texttt{sort}.\texttt{wowned}$, and $C.\texttt{wprov}$ for $C.\texttt{sort}.\texttt{wprov}$. $C.\texttt{wowned}$ are required roles for binding to a weakly owned cell. $C.\texttt{wprov}$ are provided roles for binding to a weak owner.

We adapt the invariant (1) as follows:

$$\begin{aligned} C.\texttt{prov} = C.\texttt{fprov} \cup C.\texttt{wprov} \;\wedge\; & C.\texttt{fprov} \cap C.\texttt{wprov} = \emptyset \\ \wedge\; & C.\texttt{req} = C.\texttt{owned} \cup C.\texttt{wowned} \cup C.\texttt{freq} \\ \wedge\; & C.\texttt{owned}, C.\texttt{wowned}, C.\texttt{freq} \text{ mutually disjoint} \end{aligned} \tag{5}$$

For $L, M \in \mathbb{C}$ (or $\mathbb{C}_s$), we write $L \rightharpoonup M$ for $L.\texttt{wowned} \cap M.\texttt{wprov} \neq \emptyset$.

Writing now $G.(\texttt{roles} - \texttt{wroles})$ for $G.\texttt{roles} \setminus (G.\texttt{wowned} \cup G.\texttt{wprov})$, and $F \diamond G$ for $F.(\texttt{roles} - \texttt{wroles}) \cap G.(\texttt{roles} - \texttt{wroles}) = \emptyset$ he global structural invariant is now the conjunction of the following properties:

$$\forall L, M \in G, L \neq M \implies \texttt{scope}_G(L) \diamond \texttt{scope}_G(M) \wedge L.\texttt{owned} \cap M.\texttt{owned} = \emptyset \tag{6}$$

$$\forall L, M \in G, L \multimap M \implies M.\texttt{owned} = \emptyset \tag{7}$$

$$\forall L, M, N \in G,\; L \multimap M \wedge M \frown N \implies L \multimap N \vee L = N \vee M \rightharpoonup N \vee N \rightharpoonup M \tag{8}$$

Notice how the invariant (8) changes from the strict encapsulation policy. Cells in an owner scope are now allowed to bind to cells they weakly own, i.e. the weak ownership relation allows cells to bind roles across group boundaries.

The authorization predicate for this new policy is defined as in the previous section, with just a change in the definition of the $\texttt{Safe}$ predicate. $\texttt{Safe}$ is now defined as follows:

$$\begin{aligned} \texttt{Safe}(\Gamma, M, G) \triangleq\; & \exists B \in \mathcal{P}_{\texttt{fin}}(\mathbb{R}), \texttt{New}(\Gamma.\texttt{names}, M.\texttt{roles}, B) \\ & \wedge\; G.(\texttt{roles} - \texttt{wroles}) \subseteq B \cup \texttt{scope}_\Gamma(M).(\texttt{roles} - \texttt{wroles}) \\ & \wedge\; G.\texttt{wowned} \subseteq \Gamma.\texttt{wprov} \end{aligned}$$

Using this policy, we can describe the architecture described in Figure 1 as a hypercell with cells C, C_1, C_2, L, L_1, L_2, where C and L are owners of cells C_1, C_2

and L_1, L_2, respectively, and where C_1 and C_2 are weak owners of L_1 and L_2, respectively.

As it is, the selective encapsulation policy just allows for specifically identified roles to break the encapsulation policy, and for weakly owned cells to act as shared internal means of communication between different owner scopes. It is possible, however, to enforce additional constraints on weak ownership to reflect different aggregation semantics. For instance, one could enforce a lifetime dependency between weak owner and weak owned cell, preventing the removal of a weak owner if its weakly owned cells are still in place, or, one could ensure a cell has a single weak owner. We do not present these examples here, but our two examples in this section should provide a good taste of the possibilities offered.

5 Conclusion

We have presented the Hypercell framework for defining software component models (hypercell models). The basic ontology of any hypercell model agrees with the classical elements of software component models [21, 16], but the combination of dynamicity, sharing and encapsulation an hypercell model can offer is, to the best of our knowledge, unique. The key points to retain are the following: (i) this combination is made possible by the use of contextual transitions, cell sorts and context dependent runtime checks that enforce encapsulation policies; (ii) a proper notion of equivalence between hypercells is obtained thanks to authorizations at the level of individual cells and a notion of bisimulation that decouples the effect of authorizations from the classical bisimulation game.

Our runtime approach to enforcing encapsulation policies seems more permissive, and able to express more forms of policies and aggregations semantics than possible with ownership types, as our examples suggest. However, we have at this time no formal proof of this. Also, how our approach compares with those combining static ownership discipline with dynamic ownership tests, as in the Mezzo permission-based language [1], remains to be seen. It is worth pointing out that in defining encapsulation policies in the Hypercell framework, we do have a choice between imposing static constraints on unconstrained transitions, and imposing dynamic constraints via authorization predicates. In this paper, we have opted in our examples for an approach that made maximal use of authorization, but other options are available that combine both. For expressivity, however, we believe some amount of run-time checking is inescapable.

A crucial question is of course whether our abstract Hypercell framework can be efficiently implemented and supported. An implementation of an abstract machine for object-based hypercells is currently under way, but is clear that enforcing encapsulation constraints via runtime checks is a viable option, as demonstrated by the work on Siaam [15]. This work showed, in the simpler context of the actor model, that the overhead of such checks can largely be mitigated by means of static analyses that can safely remove most unnecessary ones.

References

1. T. Balabonski, F. Pottier, and J. Protzenko. The design and formalization of Mezzo, a permission-based programming language. *ACM Trans. Program. Lang. Syst.*, 38(4), 2016.
2. F. Barbier, B. Henderson-Sellers, A. Le Parc, and J.M. Bruel. Formalization of the whole-part relationship in the unified modeling language. *IEEE Trans. Software Eng.*, 29(5), 2003.
3. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 3nd edition, 2013.
4. S. S. Bauer, A. David, R. Hennicker, K.G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Moving from specifications to contracts in component-based design. In *FASE 2012*, volume 7212 of *LNCS*, 2012.
5. J. Bengtson, M. Johansson, J. Parrow, and B. Victor. Psi-calculi: a framework for mobile processes with nominal data and logic. *Logical Methods in Computer Science*, 7(1), 2011.
6. S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *CONCUR*, volume 5201 of *LNCS*. Springer, 2008.
7. B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1), 1995.
8. E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12), 2006.
9. R. Bruni, U. Montanari, and M. Sammartino. Reconfigurable and software-defined networks of connectors and components. In *Software Engineering for Collective Autonomic Systems*, volume 8998 of *LNCS*. Springer, 2015.
10. M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: the calculus of boxed ambients. *ACM. Trans. Prog. Languages and Systems*, 26(1), 2004.
11. L. Cardelli and A. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1), 2000.
12. I. Castellani. Process algebras with localities. In *Handbook of Process Algebra, J. Bergstra, A. Ponse and S. Smolka (eds)*. Elsevier, 2001.
13. G. L. Cattani and P. Sewell. Models for name-passing processes: interleaving and causal. *Information and Computation*, 190(2), 2004.
14. D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*. Springer, 2013.
15. B. Claudel, Q. Sabah, and J.B. Stefani. Simple isolation for an actor abstract machine. In *35th IFIP Int. Conf. Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, volume 9039 of *LNCS*, 2015.
16. I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron. A classification framework for software component models. *IEEE Trans. Software Eng.*, 37(5), 2011.
17. R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *ACM Trans. on Autonomous and Adaptive Systems*, 9(2), 2014.
18. J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1), 2003.

19. G. L. Ferrari, D. Hirsch, I. Lanese, U. Montanari, and E. Tuosto. Synchronised hyperedge replacement as a model for service oriented computing. In *FMCO 2005*, volume 4111 of *LNCS*. Springer, 2005.
20. J. L. Fiadeiro and A. Lopes. A model for dynamic reconfiguration in service-oriented architectures. *Software and System Modeling*, 12(2), 2013.
21. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*. Cambridge U. Press, 2000.
22. M. Hennessy, J. Rathke, and N. Yoshida. Safedpi: a language for controlling mobile code. *Acta Inf.*, 42(4-5), 2005.
23. S. S. T. Q. Jongmans and F. Arbab. Overview of thirty semantic formalisms for Reo. *Sci. Ann. Comp. Sci.*, 22(1), 2012.
24. G. Leavens and M. Sitaraman, editors. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.
25. Zhiming Liu and He Jifeng, editors. *Mathematical Frameworks for Component Software - Models for Analysis and Synthesis*. World Scientific, 2006.
26. J. Magee and J. Kramer. Dynamic structure in software architectures. In *4th ACM Symp. on Foundations of Software Engineering (FSE-4)*. ACM, 1995.
27. P. Merle and J.B. Stefani. A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA, France, 2008.
28. R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.
29. R. De Nicola, G.L. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Trans. on Software Engineering*, 24(5), 1998.
30. A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, 2013.
31. J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 1998.
32. A. Sanchez, L. S. Barbosa, and D. Riesco. Bigraphical modelling of architectural patterns. In *FACS 2011*, volume 7253 of *LNCS*. Springer, 2012.
33. D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
34. A. Schmitt and J.-B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *LNCS*. Springer, 2005.
35. M. Sevegnani and M. Calder. Bigraphs with sharing. *Theoretical Computer Science*, 577, 2015.
36. J.B. Stefani. Components as location graphs. In *11th FACS Conference, Revised Papers*, volume 8997 of *LNCS*. Springer, 2015.
37. I. Tutu and J.L. Fiadeiro. Service-oriented logic programming. *Logical Methods in Computer Science*, 11(3), 2015.
38. R. J. van Glabbeek. The meaning of negative premises in transition system specifications II. *J. Log. Algebr. Program.*, 60-61, 2004.
39. J. Voigt. Access contracts: a dynamic approach to object-oriented access protection. Technical Report UCAM-CL-TR-880, U. of Cambridge, 2016.
40. M. Wermelinger and J. L. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Sci. Comput. Program.*, 44(2), 2002.