NEW INSIGHTS ON ARCHITECTURAL CONNECTORS*

Roberto Bruni [1], José Luiz Fiadeiro [2], Ivan Lanese [1], Antónia Lopes [3] and Ugo Montanari [1]

[1] *Computer Science Department, University of Pisa, Italy.*
{ bruni,lanese,ugo } @di.unipi.it

[2] *Department of Computer Science, University of Leicester, UK.*
jose@fiadeiro.org

[3] *Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal.*
mal@di.fc.ul.pt

Abstract This work is a first step toward the reconciliation of the two main approaches to composition in system modeling, namely the categorical one and the algebraic one. In particular, we present a mapping from CommUnity, which uses the categorical approach based on colimits, into the Tile Model, which uses algebraic operators for composition. Our results include a standard decomposition for CommUnity programs. We also establish a strong link between the colimit computation of the categorical approach and the abstract semantics of configurations in the algebraic approach by proving that the encoding of a CommUnity diagram is behaviorally equivalent to the encoding of its colimit.

Introduction

In this paper, we report on new insights on architectural connectors raised by the analysis of mobility aspects within software architectures for *Global Computing*. Since there is no single formalism that can best address these aspects we study the relationships between two different approaches: CommUnity [6] and the Tile Model [7]. The former is a prototype architectural description language that was developed to formalize the conceptual distinction between *computation* and *coordination* in communicating distributed systems. The latter is an operational model designed for concurrent systems. It is suited for behavioral semantics that deal uniformly with closed and open systems.

*Research supported by the FET-GC Project IST-2001-32747 AGILE.

These two frameworks can be seen as "canonical" representatives of two general approaches to the study of complex system structures: the *categorical approach* (for CommUnity) and the *algebraic approach* (for the Tile Model).

The *Categorical Approach*, which can be traced back to [8], is based on the definition of a category whose objects model system components and whose morphisms represent how systems are superposed, simulated, refined, etc. Complex systems can be modeled as diagrams in the category. Composition is achieved via universal constructions like taking the colimit, which encapsulates components and interactions in a single object. Different diagrams have the same colimit, which thus defines some sort of denotational semantics. The categorical approach is best suited for modeling systems based on shared resources (e.g., memory, channels, actions), sharing being expressed through morphisms. The main contribution of CommUnity has been to show how the categorical approach can be applied to program designs, formalizing architectural aspects.

The *Algebraic Approach*, initiated in [9, 13], is based on signatures whose constants are the basic processes and whose operations capture composition. The initial algebra of the signature defines the class of admissible systems. Typically, systems are equipped with an operational semantics based on labeled transition systems in the SOS style [14]. Abstract semantics can then be obtained by collapsing systems that are equivalent w.r.t. some observational semantics. The algebraic approach is best suited for message passing calculi. The main contribution of the Tile Model has been the support for two different dimensions of composition to co-exist, namely Computation and Distribution.

On the one hand, we are interested in capturing the "physiological" structure of architectural connectors, i.e. what they are made of and what mechanisms they put in place to coordinate required interactions, which is addressed through CommUnity. This is essential, for instance, to provide support for more abstract levels of modeling. On the other hand, we are interested in the "social" structures in which these connectors live, i.e. the laws that regulate the way they can be composed and superposed to interconnect components, as can be captured in the Tile Model. This step is essential for supporting the transition between the declarative and operational aspects of architectural configurations.

We are even more interested in relating the two approaches. In particular, the technical contribution of this paper is three-fold:

1 we define a standard decomposition for CommUnity diagrams in terms of elementary programs;

2 we define a translation from CommUnity diagrams into the Tile Model;

3 we establish a strong link between the denotational semantics of the categorical approach and the abstract semantics yielded by the algebraic approach by proving that the encoding of a CommUnity diagram is behaviorally equivalent to the encoding of its colimit.

design P **is**		
in		in(V)
out		out(V)
do	$[]_{g\in\Gamma}$	g: G(g) $\rightarrow$ R(g)

Figure 1. CommUnity designs.

design	counter **is**
in	x:nat
out	y:int
do	inc: $true \rightarrow y := y + x$
[]	dec: $y > MIN \rightarrow y := y - x$

Figure 2. The "counter" design.

This work has been developed in the context of FET-FP5 Project AGILE [2] on "Architectures for Mobility", that brings together different approaches to the modeling of architectural aspects of systems with the aim of complementing each other and of extending them to cope with *Global Computing*.

Structure of the paper. In § 1 we survey CommUnity and the Tile Model. In § 2 we define the standard decomposition of CommUnity diagrams and prove colimit preservation. In § 3 we define the tile system associated to CommUnity while in § 4 we sketch the encoding and the bisimilarity result (Theorem 9). Conclusions and directions for future work are given in § 5.

1. Background

1.1 CommUnity

In this section we give a brief account of CommUnity, while referring the interested reader to [5] for full details.

CommUnity is a parallel program design language in the style of Unity [3] but based on action sharing. It was initially proposed in [6] to show how programs fit into Goguen's categorical approach to General Systems Theory [8]. Since then, it has evolved into an architectural description language, capitalizing on the fact that CommUnity takes to an extreme the separation between "computation" and "coordination" concerns.

The individual components of a system can be defined in terms of *channels* and *actions* organized in *designs*. In this paper, we consider a special class of CommUnity designs (see Figure 1), called *programs*, which are particular instances of the more general form in [5].

Channels. A design P is based on a set of (input and output) *channels* V. Input channels are read-only and are controlled by the environment while output channels are controlled locally by the component. Each channel v is typed with a sort $sort(v)$ which is part of a fixed many-sorted data algebra.

Actions. A design P exploits a pointed set of actions $\Gamma_\perp$. Actions represent possible interactions between the component and the environment. For each action name g, $G(g)$ is the enabling condition of g (a predicate on V), and $R(g)$

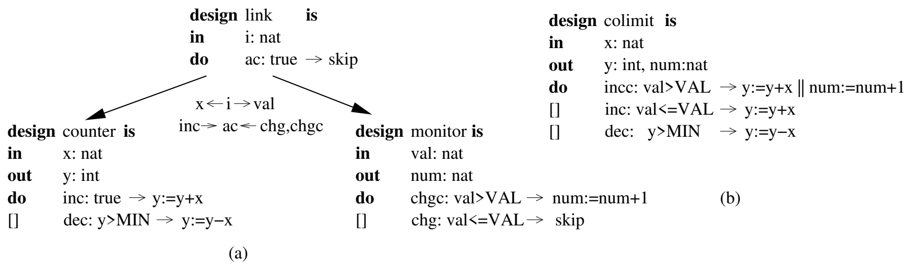

Figure 3. The "counter with monitor" diagram and its colimit.

is a multiple assignment, assigning to output channels expressions on input and output channels. The empty assignment is denoted by *skip*.

As an example, consider the design in Figure 2. It models a component that calculates and stores an integer value. It can decrease or increment this value of x units through the execution of actions *inc* and *dec*, but decreasing is only allowed when a minimum value *MIN* has not been reached.

A program with a non-empty set of input channels is *open* in the sense that its execution is only meaningful in a configuration in which these inputs have been instantiated with channels controlled by other components. A closed program behaves as follows: at each step, one of the actions whose enabling condition holds is selected, and its assignments are executed atomically.

Diagrams. The interaction between programs is based on action synchronization and interconnection of input and output channels. Name bindings are established through diagrams of the form in Figure 3(a) in a category **c-DSGN** with morphisms as follows. We may consider that the design in the middle is a program whose actions are all $true \to skip$, called a *cable* [5].

DEFINITION 1 *A morphism of designs $\sigma : P_1 \to P_2$ consists of a total function $\sigma_{var} : V_1 \to V_2$ that preserves sorts and never maps an output channel to an input channel together with a pointed mapping $\sigma_{ac} : \Gamma_{2_\perp} \to \Gamma_{1_\perp}$ that maps a conditional multiple assignment $G(g) \to R(g)$ to another one with stronger condition and a superset of the assignments (up-to renamings of channels).*

EXAMPLE 2 *The diagram in Figure 3(a) defines a system with two components: a* counter *and a* monitor. *The* monitor *counts the executions of an action when channel* val *has a value greater than a fixed value* VAL. *In that case action* chgc *takes place, action* chg *is executed in the other cases. In this configuration,* chgc *and* chg *monitor the execution of action* inc *and the value of channel* x*, as shown by the interconnection of channels* x *of* counter *and* val *of* monitor *and the synchronization of action* inc *with both* chg *or* chgc. *Here* dec *is (implicitly) mapped to the $\perp$ action of* link *and thus not synchronized.*

$$\begin{tikzcd} & \textbf{\textit{Glue}} & \\ \textbf{\textit{C}}_1 \arrow[ur] \arrow[d] & \textbf{\textit{C}}_i \arrow[u] \arrow[d] & \textbf{\textit{C}}_n \arrow[ul] \arrow[d] \\ \textbf{\textit{Role}}_1 & \textbf{\textit{Role}}_i & \textbf{\textit{Role}}_n \end{tikzcd}$$

Figure 4. Star-shaped configurations.

initial input interface *initial output interface*

$$\begin{tikzcd} x \arrow[r, "s"] \arrow[d, "a"'] \arrow[dr, phantom, "\alpha"] & y \arrow[d, "b"] \\ z \arrow[r, "t"'] & w \end{tikzcd}$$

final input interface *final output interface*

Figure 5. A generic tile α.

The colimit construction internalizes the interactions described in a diagram **dia** and returns a program $\mathsf{colim}(\mathbf{dia})$ for the system as a whole. Colimits in **c-DSGN** capture a generalized notion of parallel composition in which interconnections are explicit. The colimit of the diagram in Figure 3(a) returns, up to isomorphism, the program in Figure 3(b). Only diagrams where no output channels are connected make sense. These are called *configuration diagrams*.

To conclude this overview of CommUnity, we mention star-shaped configurations (see Figure 4) which play an important role in the process of structuring systems. They can be used to represent architectural connectors as defined in [1]: the program in the center is the *glue* and the programs in the vertices are the *roles*. Each role $Role_i$ is connected to the glue by one *cable* C_i. The glue of the connector defines how the activities of the role instances are coordinated.

1.2 Tile Model

The tile model [7] relies on rewrite rules with side effects, called *basic tiles*, which are reminiscent of SOS rules and *context systems* [10], collecting ideas from *structured transition systems* [4] and *rewriting logic* [12].

A tile $\alpha : s \xrightarrow[b]{a} t$ has the graphical representation in Figure 5, stating that the *initial configuration* s can evolve to the *final configuration* t via α, producing the *effect* b; but the step is allowed only if the 'arguments' of s can contribute by producing a, which acts as *trigger*. Triggers and effects are called *observations*. Configurations and observations are represented by arrows to show that they can be composed via their interfaces.

DEFINITION 3 *A* tile system *is a tuple* $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ *where* $\mathcal{H}$ *and* $\mathcal{V}$ *are monoidal categories with the same set of objects* $O_{\mathcal{H}} = O_{\mathcal{V}}$*,* N *is a set of rule names and* $R\colon N \to \mathcal{H} \times \mathcal{V} \times \mathcal{V} \times \mathcal{H}$ *is a function such that for all* $\alpha \in N$*, if* $R(\alpha) = \langle s, a, b, t \rangle$*, then the arrows* s, a, b, t *can form a tile like in Figure 5.*

The Tile Model is designed for systems that are *compositional in space* and in *time*: tiles can be composed horizontally, in parallel, and vertically to generate larger steps. Horizontal composition coordinates the evolution of the initial configuration of α with the evolution of the environment yielding the 'synchronization' of the two rewrites. The parallel composition builds concurrent steps.

$$\frac{R(\alpha) = \langle s, a, b, t\rangle}{s \xrightarrow[b]{a} t}\ (\texttt{bas}) \qquad \frac{s \xrightarrow[b]{a} t \quad h \xrightarrow[c]{b} f}{s;h \xrightarrow[c]{a} t;f}\ (\texttt{hor}) \qquad \frac{t\colon x \to y \in \mathcal{H}}{t \xrightarrow[id_y]{id_x} t}\ (\texttt{vid})$$

$$\frac{s \xrightarrow[b]{a} t \quad h \xrightarrow[d]{c} f}{s \otimes h \xrightarrow[b\otimes d]{a\otimes c} t \otimes f}\ (\texttt{par}) \qquad \frac{s \xrightarrow[b]{a} t \quad t \xrightarrow[d]{c} h}{s \xrightarrow[b;d]{a;c} h}\ (\texttt{ver}) \qquad \frac{a\colon x \to z \in \mathcal{V}}{id_x \xrightarrow[a]{a} id_z}\ (\texttt{hid})$$

Figure 6. Inference rules for tile logic.

Vertical composition is sequential composition of computations. Moreover, we always have the horizontal and vertical identities as auxiliary tiles (vertical identities model idle components, while horizontal identities propagate effects through identity substitutions). All this is defined in Figure 6.

Depending on the chosen tile format, $\mathcal{H}$ and $\mathcal{V}$ must satisfy certain constraints and some other auxiliary tiles are added and composed with basic tiles and identities. The set of resulting tiles defines the *tile logic* associated with $\mathcal{R}$ and we write $\mathcal{R} \vdash s \xrightarrow[b]{a} t$ if the tile is derivable.

By taking $\langle \text{trigger}, \text{effect}\rangle$ pairs as labels one can see tiles as a labeled transition system. The resulting notion of bisimilarity is called *tile bisimilarity*.

DEFINITION 4 *Let $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ be a tile system. A symmetric relation $\sim$ on configurations is called* tile bisimulation *if whenever $s \sim t$ and $\mathcal{R} \vdash s \xrightarrow[b]{a} s'$, then t' exists such that $\mathcal{R} \vdash t \xrightarrow[b]{a} t'$ and $s' \sim t'$. The maximal tile bisimulation is denoted by $\simeq$, and two configurations s and t are* tile bisimilar *iff $s \simeq t$.*

We focus on tile systems where $\mathcal{H}$ and $\mathcal{V}$ are *categories of substitutions*. Substitutions over a signature Σ and their composition $\_;\_$ form a cartesian category for which there is an alternative presentation given by Lawvere's *algebraic theories* [11]. In Lawvere's theories cartesianity is expressed by a symmetric monoidal structure $(\otimes, \underline{0}, \gamma)$ enriched with two natural transformations, a duplicator $\nabla = \{\nabla_n\colon \underline{n} \to \underline{2n}\}_n$ and a discharger $! = \{!_n\colon \underline{n} \to \underline{0}\}_n$. The result is a freely generated cartesian category $\mathbf{Th}[\Sigma]$ whose objects are underlined natural numbers and whose arrows from $\underline{m}$ to $\underline{n}$ are in a one-to-one correspondence with n-tuples of terms of the free Σ-algebra over m variables. In particular, arrows from $\underline{0}$ to $\underline{1}$ are in bijective correspondence with the closed terms over Σ. We assume the standard naming $x_1, \ldots, x_m$ of the m input variables. For example, $f \in \Sigma_2$ defines an arrow $f(x_1, x_2)\colon \underline{2} \to \underline{1}$ in $\mathbf{Th}[\Sigma]$. We denote the identity arrow $\langle x_1, \ldots, x_n\rangle$ for the object $\underline{n}$ as $id_{\underline{n}}$ and the empty substitution as $id_{\underline{0}}$.

In this work we deal with substitutions on multi-sorted terms, thus instead of natural numbers we have monoids on the set of sorts. For instance, if a and b are sorts we have $\gamma_{a,b} : a \otimes b \to b \otimes a$ and $\nabla_{a\otimes b} : a \otimes b \to a \otimes b \otimes a \otimes b$.

2. Standard decomposition of CommUnity programs

In this section we present an original decomposition for CommUnity programs, which is the first step towards the definition of the mapping from CommUnity to the Tile Model. This decomposition transforms a complex program in a star-shaped configuration with simpler components. Given a CommUnity program $\mathcal{D}$ we decompose it in a diagram with four kinds of components:

- a *glue*, which has as many actions as the number of actions in $\mathcal{D}$, but each action in the glue has the form $true \rightarrow skip$. The glue has one input channel for each input/output channel of $\mathcal{D}$;
- one *channel manager* for each output channel in $\mathcal{D}$. The channel manager for channel x has exactly one action for each action in $\mathcal{D}$, with *true* as guard and as body the assignment (if any) in the action that assigns x. The channel manager has exactly one output channel x and all the input channels needed by the assignments to x;
- one *guard manager* for each action in $\mathcal{D}$. The guard manager has exactly one action in the form $p \rightarrow skip$ where p is the guard of the corresponding action in $\mathcal{D}$. The guard manager has exactly the channels needed for evaluating its guard, all as input channels;
- *cables* to connect each channel manager and each guard manager to the glue. Each cable has one action/input channel for each action/channel of the corresponding role, mapped to that action/channel and to the corresponding action/channel in the glue.

Roughly, letting n and m be respectively the number of output channels and of actions in $\mathcal{D}$, then the standard decomposition $\mathsf{DS}(\mathcal{D})$ of $\mathcal{D}$ is a diagram with n channel managers, m guard managers, $n+m$ cables and one glue.

Figure 7 shows a sample decomposition. In the figure we have not explicitly represented the details of morphisms, but we have just used the same name in different programs for corresponding actions and corresponding channels.

We can also define the standard decomposition of a diagram **dia**, which is a diagram obtained by substituting each role with its standard decomposition. The morphisms entering a program become morphisms entering the glue of its standard decomposition.

The correctness of the decomposition is given by the following theorem, where $\cong$ denotes the isomorphism relation in **c-DSGN**.

THEOREM 5 *For each design $\mathcal{D}$ we have* $\mathsf{colim}(\mathsf{DS}(\mathcal{D})) \cong \mathcal{D}$. *Moreover, for each diagram* **dia** *we have* $\mathsf{colim}(\mathsf{DS}(\mathbf{dia})) \cong \mathsf{colim}(\mathbf{dia})$.

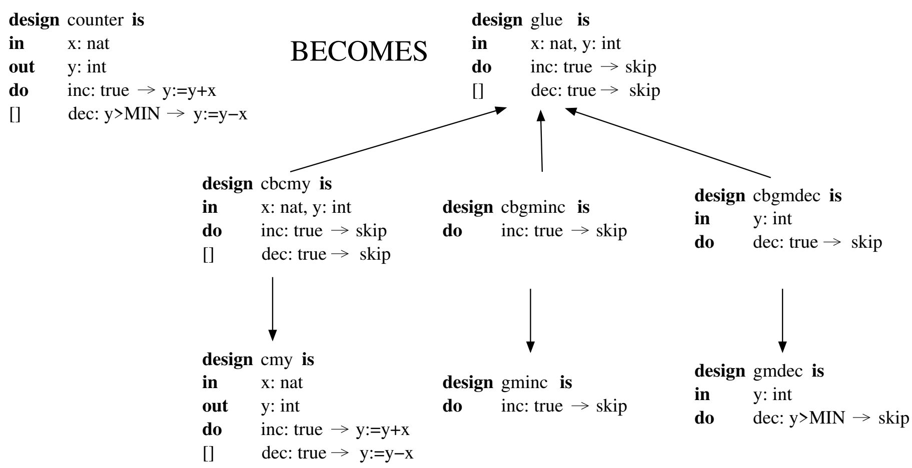

Figure 7. Standard decomposition of a CommUnity program.

3. Mapping CommUnity into the Tile Model

In this section we define the operational and abstract semantics of CommUnity by exploiting the Tile Model. The encoding maps a diagram into a tile system together with a fixed initial configuration. We consider both *anchored* systems (systems with state) and *unanchored* systems.

In order to have a clear separation between functionalities and state, each configuration is the composition of two parts: one that corresponds to the state, and the other one that corresponds to the unanchored system.

Tile objects. The typed interfaces of the tile system are tuples that contain the following elements:

- channels: these are specified by a type (boolean, integer, ...) and a modality (input or output) exactly as in CommUnity;
- a special boolean object b that is attached to the evaluation of guards;
- placeholders for actions, which play the role of synchronization objects.

We denote tuples of channels and special boolean objects with *chs* with an optional subscript to denote their cardinality and/or types, furthermore we write *ins* to specify that all channels in the tuple are input channels (or possibly special boolean objects) and similarly *outs* for outputs. We denote a synchronization object with 1 and a tuple of n synchronization objects with n.

Tile configurations. We take as horizontal category the symmetric strict monoidal category freely generated by the basic arrows below. Note that symmetries allow for rearranging the order of the objects in the interfaces.

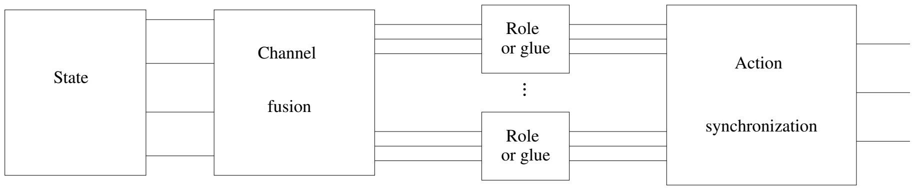

Figure 8. Initial configuration for a CommUnity program.

- $state[val : typ] : 0 \to chs$ models a state where val is a tuple of values of types typ and chs a tuple of channels with these types; arrows of this form model the actual states of anchored configurations;
- $cm[\langle f_i \rangle_{i=1\ldots n}] : out \otimes ins \otimes b \to n$ models a channel manager where f_is are functions on channels in $out \otimes ins$ to the output channel out;
- $gm[p] : ins \otimes b \to 1$ models a guard manager with predicate p that uses channels in ins;
- $\nabla_{chs,ins\otimes chs} : chs \to ins \otimes chs$ which are data synchronization connectors (where ins and chs have the same number of elements and the same type);
- $!_{ins} : ins \to 0$; which are hiding connectors;
- $\nabla_n : n \to 2n$ and $!_n : n \to 0$ which are mutual exclusion and hiding connectors respectively;
- $\nabla_n : n \to 2n$ and $\Delta_n : 2n \to n$ which are synchronization connectors;
- $\bar{\mathbf{1}} : 0 \to 1$ which is a connector that forces some actions to be performed.

The structure of the anchored configuration obtained as the translation of a generic CommUnity diagram is shown in Figure 8, using the wire-and-box notation, where arrows are represented as boxes and their composition as wiring between their interfaces.

Given a well-formed connected diagram on which we have applied the standard decomposition we want to build such an arrow in a compositional way. In order to do that we must first fix a total ordering over the programs in the diagram and we translate each of them separately. Then we use as basic operation to build up the system the "parallel composition through one cable". Thanks to the hypothesis that the CommUnity diagram is well-formed, a sequence of operations of this kind allows to translate the whole diagram. At the end we may add the state. A main result ensures that the behavior of the resulting tile system is independent from the choice of the ordering (Proposition 8).

During the translation we have to remember which are the channels that correspond to each channel object and which is the action that corresponds to each synchronization object.

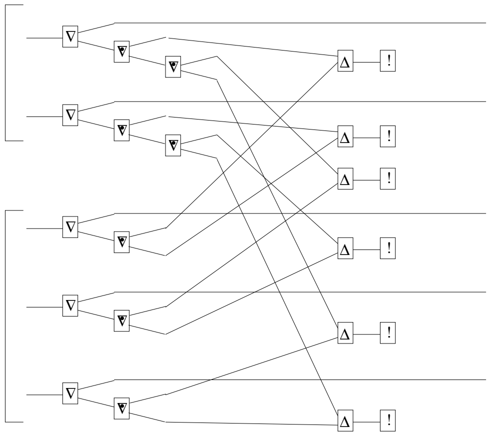

Figure 9. Connecting actions.

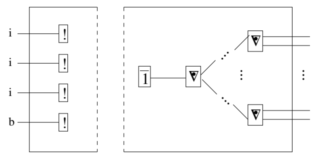

Figure 10. A glue as an arrow.

- The translation of a channel manager is a basic arrow $cm[\langle f_i \rangle_{i=1\ldots n}]$: $chs \otimes b \to n$ where *chs* contains the channels used by the channel manager and f_i is the function that is computed during the *i*-th action. The *i*-th action corresponds to the *i*-th synchronization object in the interface.

- The translation of a guard manager is a basic arrow $gm[p] : ins \otimes b \to 1$ where p is the guard of the only action of the guard manager.

- The translation of a glue with n actions is a tree composed by ∇ connectors with n leaves and with a $\overline{\mathbf{1}}$ connector as root together with the ! for all its channels (plus one for a special boolean object), see e. g. Figure 10.

Note that we have a bijective correspondence between channels in a program and channel objects in the left interface of its translation and between actions of a program and synchronization objects in the right interface.

We show now how the operation of "parallel composition through one cable" is performed. Suppose we have a cable with channels $x_1, \ldots, x_n$ and actions $a_1, \ldots, a_n$. Each channel is mapped through morphisms to two groups of channels, one for each of the diagrams to be composed. Each action is the image of zero or more actions from the diagrams to be composed.

The resulting arrow is obtained by taking the parallel composition of the translations of the two components. On the left we merge, using trees of ∇ connectors (and possibly some permutations), the two special boolean objects and all the channels that are mapped to the same channel in the cable.

On the right we have to synchronize tuples of actions that are mapped to the same action in the cable. In order to synchronize a tuple t_1 of n actions with a tuple t_2 of m actions we have to duplicate each of them. Then we create using

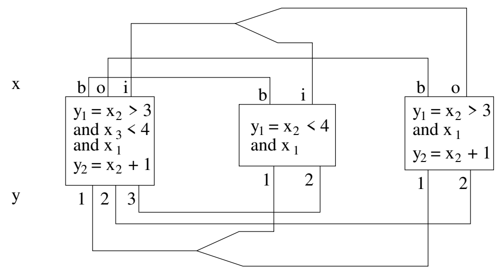

Figure 11. A sample tile for data handling.

∇ connectors n links to each object in t_2 and m links to each object in t_1. Then we merge using Δ connectors each action in the first group with each action in the second group and we close the resulting objects using ! connectors. See Figure 9 for an example (groups of two and three actions respectively).

The left interface of the resulting arrow has all the channels modulo equivalence while the right interface has all the actions in the components.

When the whole diagram has been mapped we also need to close all the synchronization objects in the output interface using ! connectors. If we want an anchored configuration, we can add the state to the left.

Tile observations. The observations of our tile system are of two kinds: in the action part we have tuples of $\mathsf{tick} : 1 \to 1$ and $\mathsf{untick} : 1 \to 1$ operators, which express that the action associated with the initial interface is either taking place or it is inhibited, respectively.

In the channel part we have as observations conditional multiple assignments where the condition is associated to the special boolean object and the assignments to the output channels (note that names are immaterial). This kind of observations can be formalized as arrows of $\mathbf{Th}[\Sigma]$ where Σ contains the data-signature, all predicate symbols, logical conjunction and also a $*$ unary operator standing for a guess on the update of input channels, on which components have no control. Graphically, observations are conveniently represented as boxes decorated with predicates and assignments over the variables in the initial and final interfaces, denoted by the x's and y's, respectively. Three sample observations are in Figure 11.

Tiles. The rules defining the behavior of the configurations in terms of allowed observations are the following. Since the structure of diagrams is fixed, we have tiles with equal initial and final configurations, except for the values in the state.

- $state[val : typ] \xrightarrow[Term]{id_0} state[val' : typ]$ where $Term$ is a conditional multiple assignment whose condition is satisfied by val such that val' is obtained evaluating the assignments on *val* (the arrow *Term* involves a guess $*$ attached to each input variable);

- there are several auxiliary tiles for value handling that guarantee the consistency of distributed assignments and assumptions on shared channels (an example of tile for value handling is in Figure 11);

- there are n possible tiles for each channel manager with n actions, of the form $cm[\langle f_i \rangle_{i=1\ldots n}] \xrightarrow[\mathsf{untick}^{i-1}\otimes\mathsf{tick}\otimes\mathsf{untick}^{n-i}]{true\to f_i} cm[\langle f_i \rangle_{i=1\ldots n}]$ where untick^{i-1} denotes the monoidal product of $i-1$ instances of untick, and the term $true \to f_i$ assigns f_i to the output channel and has *true* as condition;

- there are two possible tiles for each guard manager:

$$gm[p] \xrightarrow[\mathsf{tick}]{p\to skip} gm[p] \qquad gm[p] \xrightarrow[\mathsf{untick}]{true\to skip} gm[p]$$

- tiles for action connectors are as follows. They define the allowed combinations of tick and untick actions at the interfaces.

$$\nabla \xrightarrow[\mathsf{tick}\otimes\mathsf{untick}]{\mathsf{tick}} \nabla \qquad \nabla \xrightarrow[\mathsf{untick}\otimes\mathsf{tick}]{\mathsf{tick}} \nabla \qquad \nabla \xrightarrow[\mathsf{untick}\otimes\mathsf{untick}]{\mathsf{untick}} \nabla$$

$$\nabla_1 \xrightarrow[\mathsf{tick}\otimes\mathsf{tick}]{\mathsf{tick}} \nabla_1 \qquad \Delta_1 \xrightarrow[\mathsf{tick}]{\mathsf{tick}\otimes\mathsf{tick}} \Delta_1 \qquad !_1 \xrightarrow[id_0]{\mathsf{tick}} !_1 \qquad \bar{\mathbf{1}} \xrightarrow[\mathsf{tick}]{id_0} \bar{\mathbf{1}}$$

$$\nabla_1 \xrightarrow[\mathsf{untick}\otimes\mathsf{untick}]{\mathsf{untick}} \nabla_1 \qquad \Delta_1 \xrightarrow[\mathsf{untick}]{\mathsf{untick}\otimes\mathsf{untick}} \Delta_1 \qquad !_1 \xrightarrow[id_0]{\mathsf{untick}} !_1$$

4. The encoding and its properties

The tile system for CommUnity allows for many ill-formed configurations that have no correspondence with CommUnity diagrams, so we restrict our attention to configurations that are images of configuration diagrams as defined in § 1.1.

The operational semantics is then given by taking as transitions the tiles whose initial configuration is the image of such a diagram. The abstract semantics is given by tile bisimilarity.

Although the details of the encoding are omitted because of space limitations, the formal definition is given inductively on the size of the diagram. Let **dia** be a CommUnity configuration diagram, let $\mathsf{DS}(\mathbf{dia})$ be its standard decomposition, let $\leq$ be a total ordering on the programs in $\mathsf{DS}(\mathbf{dia})$, and let *val* denote the initial state; then we denote by $\mathsf{TS}(\mathbf{dia}, \leq, val)$ the resulting initial anchored configuration where synchronization objects have been closed using ! connectors. We denote by $\mathsf{TS}(\mathbf{dia}, \leq)$ the corresponding unanchored configuration.

Note that the configuration $\mathsf{TS}(\mathbf{dia}, \leq, val)$ has empty input and output interfaces, while the input interface of $\mathsf{TS}(\mathbf{dia}, \leq)$ has the form $chs \otimes b$, for chs the list of typed channels in $\mathsf{colim}(\mathbf{dia})$. When $\mathsf{DS}(\mathbf{dia})$ is a single program, the total order is fixed and we denote it with $\bullet$.

PROPOSITION 6 *If a tile has initial configuration* $\mathsf{TS}(\mathbf{dia}, \leq, val)$*, then there exists an assignment of values* val' *such that the final configuration takes the form* $\mathsf{TS}(\mathbf{dia}, \leq, val')$.

We state the correctness result of our encoding w.r.t. the intended behavior of the program obtained as colimit of the diagram.

THEOREM 7 *We have a tile with* $\mathsf{TS}(\mathbf{dia}, \leq, val) \xrightarrow[id_0]{id_0} \mathsf{TS}(\mathbf{dia}, \leq, val')$ *iff there exists a sequence of (enabled) actions of* $\mathsf{colim}(\mathbf{dia})$ *starting from a state with values* val *to a state with values* val'.

It follows that the ordering considered in the encoding is immaterial.

PROPOSITION 8 *Let* **dia** *be a configuration diagram, let* $\leq$, $\leq'$ *be total orderings on the programs in* $\mathsf{DS}(\mathbf{dia})$*, and let* val *denote the initial state. Then,* $\mathsf{TS}(\mathbf{dia}, \leq, val) \simeq \mathsf{TS}(\mathbf{dia}, \leq', val)$. *Moreover, there exists a symmetry* ρ *such that* $\mathsf{TS}(\mathbf{dia}, \leq) \simeq (\rho \otimes id_b); \mathsf{TS}(\mathbf{dia}, \leq')$.

The symmetry ρ is needed to rearrange the input interface of $\mathsf{TS}(\mathbf{dia}, \leq')$ so that channel objects that are associated to the same channel in $\mathsf{colim}(\mathbf{dia})$ have the same position in $\mathsf{TS}(\mathbf{dia}, \leq)$ and $(\rho \otimes id_b); \mathsf{TS}(\mathbf{dia}, \leq')$

Proposition 8 is also instrumental in proving the main result below.

THEOREM 9 $\mathsf{TS}(\mathbf{dia}, \leq, val) \simeq \mathsf{TS}(\mathsf{colim}(\mathbf{dia}), \bullet, val)$. *Moreover, there exists a symmetry* ρ *such that* $\mathsf{TS}(\mathbf{dia}, \leq) \simeq (\rho \otimes id_b); \mathsf{TS}(\mathsf{colim}(\mathbf{dia}), \bullet)$.

5. Concluding remarks

In this paper, we have reported on our research aimed to establish connections between CommUnity and the Tile Model in the way they address architectural concerns in the development of distributed and mobile software systems.

The main results of our investigation are:

- we have identified a standard decomposition for CommUnity programs, which separates the key aspects involved: channel managers, guard managers, coordination;
- the encoding into tiles gives an operational semantics to CommUnity programs and an abstract semantics correct w.r.t. the colimit construction;

- the separation of concerns has been exported from CommUnity to the Tile Model by separating the state from functionalities in the latter.

As already mentioned, the two frameworks are being investigated as representatives of two more general approaches to the study of complex system structures in general, and of their architectural aspects in particular: the *categorical approach* and the *algebraic approach*. It is clear that both views allow for separating *components* from *coordinators*.

In CommUnity the elementary components are channel managers and guard managers, which are boxes in the Tile Model, while coordination is expressed by cables, glues and morphisms which are connectors in the Tile Model.

As future work, we plan to extend our investigation by taking into account locality and mobility aspects and dynamic diagram reconfigurations. Furthermore we want to find a suitable axiomatization of ours connectors such that the translation of a diagram and of its colimit are equal up-to the axioms.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connectors. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[2] L. Andrade *et al.* AGILE: Software architecture for mobility. *Proc. of WADT 2002*, *LNCS* 2755, pp. 1–33. Springer Verlag, 2003.

[3] K. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley, 1988.

[4] A. Corradini and U. Montanari. An algebraic semantics for structured transition systems and its application to logic programs. *Theoret. Comput. Sci.*, 103:51–106, 1992.

[5] J.L. Fiadeiro, A. Lopes, and M. Wermelinger. A mathematical semantics for architectural connectors. *Generic Programming*, *LNCS* 2793, pp. 190–234. Springer Verlag, 2003.

[6] J.L. Fiadeiro and T. Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28:111–138, 1997.

[7] F. Gadducci and U. Montanari. The tile model. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pp. 133–166. MIT Press, 2000.

[8] J. Goguen. Categorical foundations for general systems theory. *Advances in Cybernetics and Systems Research*, pp. 121–130. Transcripta Books, 1973.

[9] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[10] K.G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. *Proc. of ICALP'90*, *LNCS* 443, pp. 526–539. Springer Verlag, 1990.

[11] F.W. Lawvere. Functorial semantics of algebraic theories. *Proc. National Academy of Sciences*, 50:869–872, 1963.

[12] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoret. Comput. Sci.*, 96:73–155, 1992.

[13] R. Milner. A calculus of communicating systems. *LNCS* 92. Springer Verlag, 1989.

[14] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.