Taming Hierarchical Connectors

José Proença and Alexandre Madeira

HASLab/INESC TEC, Universidade do Minho, Portugal
CIDMA, Universidade de Aveiro

Abstract. Building and maintaining complex systems requires good software engineering practices, including code modularity and reuse. The same applies in the context of coordination of complex component-based systems. This paper investigates how to verify properties of complex coordination patterns built hierarchically, i.e., built from composing blocks that are in turn built from smaller blocks. Most existing approaches to verify properties *flatten* these hierarchical models before the verification process, losing the hierarchical structure. We propose an approach to verify hierarchical models using *containers* as actions; more concretely, containers interacting with their neighbours. We present a dynamic modal logic tailored for hierarchical connectors, using Reo and Petri Nets to illustrate our approach. We realise our approach via a prototype implementation available online to verify hierarchical Reo connectors, encoding connectors and formulas into mCRL2 specifications and formulas.

1 Introduction

Coordination languages describe how to combine the behaviour of independently executing components, oblivious to each other. As the complexity of systems and their coordination increases, so does the need to structure these systems into reusable blocks of manageable size. In the context of coordination languages, we argue that a complex connector or protocol should be built using a hierarchy of reusable blocks, each in turn built by compositing more refined blocks.

This section motivates this notion of hierarchical construction using Reo [1] to describe a switcher connector (Fig. 1) that routes data from a source end a to either a sink end b or a sink end c. A second source end sw switches between the two possible data flows, i.e., initially all data flows from a to b, but after signalled by sw data will flow from a to c. Observe that the hierarchical construction can also be used with other connector models, such as process algebra communicating over shared channels [4], or with Petri Nets [5].

Connectors interact with components and with other connectors via their interfaces, depicted as '`o`' in Fig. 1. Informally, the `xor` connector sends data atomically from its left port to either its top-right or bottom-right port. In turn, the `alternator` uses this connector alternate between sending data from its left port to its top-right and to its bottom-right ports. The `gateOpen` connector controls the passage of data from the left port to the right port: initially data can flow; upon receiving a signal from its bottom port the data flow stops until

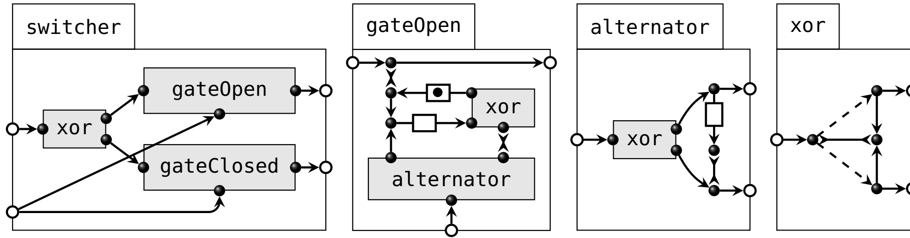

Fig. 1. Hierarchical construction of a `switcher` connector.

a new signal is received from its bottom port. Finally, the `switcher` connector routes data to either the top or to the bottom right port, alternating whenever the left-bottom port is triggered.

Existing approaches to model-check Reo connectors consider only the flattened connector (see Fig. 2 for the flattened `switcher`). This paper addresses how to model-check hierarchical connectors, exploiting the hierarchical structure. E.g., allowing one to verify if, after ignoring the internals of the `alternator`, the `switcher` can output two consecutive values on its bottom-right port.

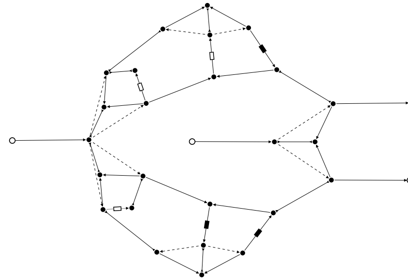

Fig. 2. Flattened `switcher` connector.

We present a *model* and a *modal logic* to specify hierarchical connectors, not restricted to the realm of Reo, whose alphabet of actions are the reusable *containers*. Containers can be either a primitive connector (e.g., —□→) or a connector built with other containers (e.g., `gateOpen` in Fig. 1). *Performing* a container c, from our perspective, means performing an action where c interacts with its exterior.

Summarising, the key contribution of this paper is a model (Section 2) and a logic (Section 3) to specify and model-check hierarchical connectors, using Reo as an example to specify connectors. A prototype implementation generates mCRL2 specifications and logical formulas of our proposed model and logic (Section 4).

2 Modelling Hierarchical Connectors

We define hierarchical models for connectors for which a compositional semantics exist. This semantics may be given, for example, by constraint automata [2] (in the case of Reo), by a process algebra [4], or by Petri nets (PN) [5]. The toy examples in Fig. 3, using Reo and PN, will be used to illustrate the concepts.

Using the constraint automata semantics [2] without data constraints [6] for Reo and the standard Petri Net semantics we derive their semantics in Table 1

Table 1. Semantics of the containers in Fig. 3 (x and y) and container abstractions (∂).

	x	y	$\partial_{\mathsf{x},\mathsf{y}}(\mathsf{x})$	$\partial_{\mathsf{x}}(\mathsf{x})$
Reo	a,c; b,c; d,e; d	c; d,e; d	x,y; x,y; τ	x; x; τ
PN	a; b; c; e; f; e; f; a; b	c; e; f	x; x,y; τ; x,y; x,y; τ; x	x; x; τ; x; x; τ; x

(columns x and y). The two right columns exemplify container abstractions, replacing actions by the container names that use these to interact. The Reo and PN semantics are omitted because these are orthogonal to our approach.

A *hierarchical connector* (HiCon) is as a set of nested containers, each mapped to a labelled transition system whose labels consist of sets of actions. In turn, these actions are mapped to the set of their parent containers. As an example, the PN of Fig. 3 has containers x and y, whereas y is in x. The transition a belongs to x, c belongs to y, f does not belong to any container, and e belongs to both x and y. This notion of action belonging to containers is then used to formalise the container abstraction ∂ up to a given set of containers.

Formally, a **HiCon** is a tuple $H = (C, A, rt, \sigma, \rho)$ such that: C is a set of containers; A is a set of actions performed by containers; $rt \in C$ is the root container; σ is a function mapping each container c to a labelled transition system (LTS) $(Q_c, q_{0,c}, A_c, \rightarrow_c)$, with states Q_c, initial state $q_{0,c}$, actions $A_c \subseteq A$, and transition relation $\rightarrow \subseteq Q_c \times 2^{A_c} \times Q_c$; and $\rho = (\rho_C, \rho_A)$ is a pair of functions $\rho_C : C \rightarrow C$ and $\rho_A : A \rightarrow 2^C$, where ρ_C induces a total partial order with upper bound rt, and ρ_A maps actions to their parents such that $c \in \rho_A(a)$ implies $a \in A_c$. For example, the Reo connector depicted in Fig. 3 is formalised as $(C, A, \mathsf{x}, \sigma, \rho)$ where $C = \{\mathsf{x}, \mathsf{y}\}$, $A = \{a, b, c, d, e\}$, $\sigma(\mathsf{x})$ and $\sigma(\mathsf{y})$ are depicted in Table 1, $\rho_C = \{\mathsf{y} \mapsto \mathsf{x}, \mathsf{x} \mapsto \mathsf{x}\}$ and $\rho_A = \{a \mapsto \{\mathsf{x}\}, b \mapsto \{\mathsf{x}\}, c \mapsto \{\mathsf{x}, \mathsf{y}\}, d \mapsto \varnothing, e \mapsto \{\mathsf{x}, \mathsf{y}\}\}$.

Given a HiCon H, a container c, and a subset C_H of its containers, we define the **container abstraction** of c up to the containers in C_H, written $\partial_{C_H}(c)$, as the LTS $(Q, q_0, C', \rightarrow')$ where $(Q, q_0, A, \rightarrow) = \sigma(c)$, $\rightarrow' = \{(q, \bigcup \rho(as), q') \mid (q, as,$

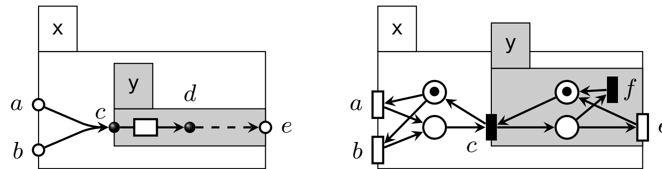

Fig. 3. Similar examples of hierarchical connectors, using Reo (left) and PN (right).

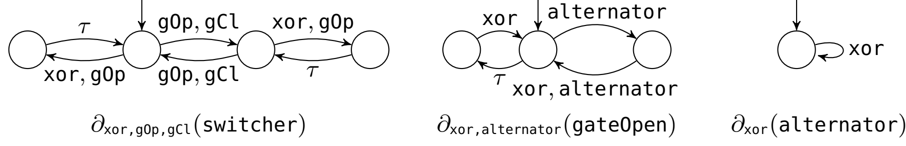

Fig. 4. Examples of container abstractions of the switcher connector from Fig. 3.

$q') \in \rightarrow\}$, and $C' = \{c' \mid ((q, cs, q') \in \rightarrow) \wedge c' \in cs\}$. This definition matches $\partial_{\mathsf{x,y}}(\mathsf{x})$ and $\partial_{\mathsf{x}}(\mathsf{x})$ depicted in Table 1. The more complex example in Fig. 1, using Reo's Port Automata semantics, yields the container abstractions in Fig. 4.

3 Container logic

This section introduces a logic to express and verify properties over hierarchical connectors. For that, let us consider the following syntax:

$$\begin{aligned}
\psi &:= \mathsf{true} \mid \mathsf{false} \mid \langle\phi\rangle\,\psi \mid [\phi]\,\psi \mid @_c\,\psi \mid \partial\,\psi && \text{(state formula)}\\
\phi &:= \varphi \mid \phi^* \mid \phi + \phi \mid \phi\,.\,\phi && \text{(regular formula)}\\
\varphi &:= c \mid \tau \mid \mathsf{all} \mid \mathsf{none} \mid \overline{\varphi} \mid \varphi + \varphi \mid \varphi \,\&\, \varphi && \text{(action formula)}
\end{aligned}$$

This logic, based on a Hennessy-Milner with regular modalities (e.g. as the one adopted in the mCRL2 toolset [4]), is intended to express and verify properties of container abstractions $\partial_C(c)$. *Action formulas* build sets of actions over basic containers and abstract transitions τ (that abstracts actions not belonging to containers C). *Regular formulas* represent regular expressions over these sets. Finally, *state formulas* enrich standard dynamic (modal) formulas with two extra operators, aiming to navigate over the hierarchy of containers. Intuitively, @ operator allows to move down in the hierarchy by 'looking within' the view being analysed; conversely, ∂ operator goes up by 'looking outside' of the current view.

We start by formalising the *interpretation of regular formulas* in our semantic structures. A regular formula ϕ is inductively interpreted in a container abstraction $M = \partial_C(c) = (Q, q_0, C, \rightarrow)$ by the relation $M_\phi \subseteq Q \times Q$ defined below.

$$\begin{array}{lll}
M_{\phi^*} = (M_\phi)^* & M_{\phi+\phi'} = M_\phi \cup M_{\phi'} & M_{\varphi\&\varphi'} = M_\varphi \cap M_{\varphi'}\\
M_{\phi.\phi'} = \{(q,q') \mid \exists s \in Q \cdot (q,s) \in M_\phi \wedge (s,q') \in M_{\phi'}\} & & M_{\overline{\varphi}} = Q \times Q \setminus M_\varphi\\
M_{\mathsf{all}} = \{(q,q') \mid (q,c,q') \in \rightarrow, c \in C\} & M_\tau = \{(q,q') \mid (q,c,q') \in \rightarrow, c \notin C\} &\\
M_{\mathsf{none}} = Q \times Q \setminus M_{\mathsf{all}} & M_c = \{(q,q') \mid (q,c,q') \in \rightarrow, c \in C\} &
\end{array}$$

Let $H = (C, A, rt, \sigma, \rho)$ be a hierarchical connector, $c \in C$ a container, and $C_H \subseteq C$ a set of containers. The **satisfaction of a formula** ψ in a state $q \in Q$ of a container abstraction $M = \partial_C(c) = (Q, q_0, C, \rightarrow)$ is defined as follows.

$$\begin{array}{ll}
H, c, q \models \mathsf{true} \text{ is always true} & H, c, q \models @_{c'}\,\psi \text{ if } H, c', q \models \psi\\
H, c, q \models \mathsf{false} \text{ is always false} & H, c, q \models \partial\,\psi \text{ if } H, \rho_C(c), q \models \psi\\
H, c, q \models \langle\phi\rangle\,\psi \text{ if } \exists q' \in Q \cdot (q,q') \in M_\phi \wedge H, c, q' \models \psi &\\
H, c, q \models [\phi]\,\psi \text{ if } \forall q' \in Q \cdot (q,q') \in M_\phi \Rightarrow H, c, q' \models \psi &
\end{array}$$

Consider, for example, the formulas $\phi_1 = [\mathsf{all}^* \,.\, \texttt{gateOpen} \,\&\, \texttt{gateClose}]\,\mathsf{false}$ and $\phi_2 = \langle \mathsf{all}^* \,.\, \texttt{gateOpen}\rangle\, @_{\texttt{gateOpen}}\, \langle \overline{\texttt{alternator}}^* \rangle\, \partial\, \langle \texttt{gateOpen} \rangle\, \mathsf{true}$. The first states that `gateOpen` and `gateClose` cannot fire at the same time, and the second states that `gateOpen` can fire twice in a row without its `alternator` firing. Both properties hold for the `switcher` connector; more specifically, it holds that $\partial_{\texttt{gOp,gCl}}(\texttt{switcher}), \texttt{switcher}, q \models \psi_1$ and $\partial_{\texttt{gOp,alt}}(\texttt{switcher}), \texttt{switcher}, q \models \psi_2$, where q is the initial state of `switcher`.

4 Model-checking HiCon in practice

We propose a concrete approach to model-check HiCon, in the context of Reo connectors, by using mCRL2 model-checking tools. This work is built over the encoding of a calculus of Reo [8] into the process algebra used to describe mCRL2 specifications [3,7], here extended to hierarchical connectors, and over the μ modal logic used by mCRL2's model-checker [4].

This section introduces (1) the hierarchical calculus of Reo connectors, (2) its encoding into mCRL2, and (3) an informal encoding of our logic into the standard modal logic used in mCRL2. Our approach is realised by a public prototype tool that can be executed inside a web-browser (to run the encodings (2) and (3) using JavaScript) and locally (to directly invoke mCRL2 by a local server).[1]

Hierarchical calculus of Reo connectors. The core language of hierarchical connectors is given by the grammar below, based on the core by Proença and Clarke [8], where n is a number and $\mathcal{P}$ is a set of primitive connectors.

$$\begin{aligned} c &:= \; p \in \mathcal{P} \mid \texttt{id} \mid \texttt{swap} \mid c\,;c' \mid c * c' \mid \texttt{loop}(n)(c) \mid c\{\mathit{def}\} \\ \mathit{def} &:= \; s = c \mid \texttt{[hide]}s = c \mid \mathit{def}\,,\, \mathit{def}' \end{aligned}$$

The set $\mathcal{P}$ includes primitives `dupl` (to duplicate data), `merger` (to combine two inputs), `fifo`, `lossy`, and `drain`. In a nutshell, connectors are sequentially composed with '`;`' and composed in parallel with '$*$'. The connector `id` is the identity of '`;`', `swap` swaps the order of 2 inputs, and `loop(1)`(c) creates a feedback loop from the last output of c to its first input.

The example from Fig. 3 can be written as '`x {y=fifo;lossy, x=merger;y}`', meaning that the connector is the container `x`, defined as `merger;y`, and `y` is defined as `fifo;lossy`. We can define the container abstraction $\partial_{\texttt{merger,y}}(\texttt{x})$ by marking the specification of `y` with the prefix `[hide]`.

Encoding into mCLR2 specifications. We encode hierarchical Reo connectors into mCRL2 specifications based on a previous encoding of a calculus of flatten Reo connectors [3]. The hierarchy allows (1) the hiding of actions of containers marked as hidden, and (2) the inclusion of the names of the parent containers in the actions. We describe this encoding using as example the encoding of the connector '`x {[hide]y=fifo;lossy, x=merger;y}`' from Fig. 3:

[1] `https://reolanguage.github.io/ReoLive/snapshot/` – see Appendix A

```
proc
Merger1 = (x_merger_1i1|x_merger_1m3 + x_merger_1i2|x_merger_1m3) . Merger1;
Fifo2 = x_y_2m4 . x_y_2m5 . Fifo2;
Lossy3 = (x_y_3m6 + x_y_3m6|x_y_3o1) . Lossy3;
Init1 = hide({x_y_2m5_x_y_3m6},
        block({x_y_2m5, x_y_3m6},
        comm({x_y_2m5|x_y_3m6 -> x_y_2m5_x_y_3m6}, Lossy3 || Fifo2 )));
Init2 = block({x_merger_1m3, x_y_2m4},
        comm({x_merger_1m3|x_y_2m4 -> x_merger_1m3_x_y_2m4}, Init1 || Merger1));
init Init2;
```

Actions in this specification are ports of the Reo connectors. E.g. `x_merger_1i2` denotes the 2nd input port of the `merger` in container `x` with unique identifier `1`. The main process denoting this connector is `Init2` (line 10), which is defined as the `Init1` and `Merger1` processes in parallel (line 9). In turn, `Init1` consists of the `Lossy3` and `Fifo2` processes. In both `Init` processes communication is enforced by the `block` and `comm` constructs, but they differ in that `Init1` also includes a `hide` construct (line 5) to hide communication between its `lossy` and `fifo`.

Encoding into mCRL2 formulas. The encoding into mCRL2 specifications shown above quickly becomes unreadable for humans. To verify Reo connectors we use our container logic (Section 3) over containers (including primitive connectors), encoded into the modal logic used by mCRL2 to verify the encoded mCRL2 specifications. For example, the property $\langle \mathsf{all}^* \rangle\, @_{\mathsf{x}}\, \langle \mathtt{merger} \rangle\, \mathtt{true}$ can be read as *"at any moment the container `merger` inside `x` can interact"*, and is translated into the modal formula `<true*> <x_merger_1i1|x_merger_1m3_x_y_2m4 || x_merger_1i2|x_merger_1m3_x_y_2m4> true`. Informally, this encoding collects all possible actions and communications by traversing the internal representation of the mCRL2 specification, and uses this to infer in which actions the `merger` container can have interactions with its neighbours. It then expands `merger` occurrence by a disjunctions of its possibilities. Note that all constructs of our container logic are mapped directly into their mCRL2 counterparts, with the exception of $@$ and ∂ that are used to pinpoint the desired containers.

5 Conclusions and future work

This paper presents an approach to reason about the behaviour of connectors built in a modular way. We empower the hierarchical structure of this construction, and propose a model that focuses on the containers of sub-connectors rather than on their interfaces. An action in the evolution of this model consists of a set of containers that interact with its neighbours at a given moment in time. We claim that this perspective over hierarchical connectors facilitates the writing and verification of properties of complex connectors.

In the future we plan to further explore the dedicated logic for hierarchical connectors. On one hand, we plan to exploit the existence of internal actions at different levels, leading to notions of weak and strong modalities and to new notions of behavioural equivalences. On the other hand, we plan to formalise the encodings described in Section 4, and to prove relevant results over our constructions, such as the preservation of behaviour during container abstractions.

Acknowledgements. This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within projects `POCI-01-0145-FEDER-029946` (first author) and `POCI-01-0145-FEDER-016692` (second author) and in the scope of the framework contract foreseen in the numbers 4, 5 and 6 of the article 23, of the Decree-Law 57/2016, of August 29, changed by Portuguese Law 57/2017, of July 19.

References

1. F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
2. C. Baier, M. Sirjani, F. Arbab, and J. J. M. M. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.
3. R. Cruz and J. Proença. Reolive: Analysing connectors in your browser. In J.-M. Jacquet and J. Soldani, editors, *FOCLASA*, LNCS, 2018. To appear.
4. J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany.
5. K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Publishing Company, Incorporated, 1st edition, 2009.
6. C. Koehler and D. Clarke. Decomposing port automata. In *Proc. SAC '09*, pages 1369–1373, New York, NY, USA, 2009. ACM.
7. N. Kokash, C. Krause, and E. P. de Vink. Reo + mCRL2: A framework for model-checking dataflow in service compositions. *FAC*, 24(2):187–216, 2012.
8. J. Proença and D. Clarke. Typed connector families and their semantics. *Sci. Comput. Program.*, 146:28–49, 2017.

A HiCon in the ReoLive framework

All examples presented in this paper, including the `switcher` connector, have been specified and analysed using the ReoLive framework, which we extended for hierarchical connectors. We invite the reader to open the website https://reolanguage.github.io/ReoLive/snapshot/, and find in the "Examples" window the `switcher` example. The output should match the screenshot below, with the specification and the logical formulas on the left side, and the visualisation and the generated mCRL2 specification on the right size. Using the button to load the formula will print the encoded mCRL2 formula, which can be used against the mCRL2 specification. By downloading ReoLive, it is possible to run a server locally that will also include the options to invoke mCRL2 tools directly from the browser to verify the formulas and to visualise the mCRL2 instance.

Input

```
1  switcher {
2    [hide] xor = ...,
3    [hide] alternator = ...,
4    [hide] gateOpen = ...,
5    [hide] gateClosed = ...,
6    switcher =
7      xor*dupl;
8      id*swap*id;
9      gateOpen * gateClosed
10 }
```

Type

Concrete instance

examples

Modal Logic

```
1 // sometimes gateOpen cannot fire
2 <all*>@switcher[gateOpen]false
```

Circuit of the instance

xor

gateClosed

gateOpen

Automaton of the instance

mCRL2 of the instance

```
act
switcher_xor_1i1, switcher_xor_1m2, switcher_xor_1m3,
switcher_xor_2m4, switcher_xor_2m5, switcher_xor_2m6,
switcher_xor_3m7, switcher_xor_3m8,
switcher_xor_1m2_switcher_xor_2m4,
switcher_xor_1m3_switcher_xor_3m7, switcher_xor_4m11,
switcher_xor_4m12, switcher_xor_5m13, switcher_xor_5m14,
switcher_xor_6m15, switcher_xor_6m16, switcher_xor_6m17,
switcher_xor_7m18, switcher_xor_7m19, switcher_xor_7m20
```