A Hypergraph-based Formalization of Hierarchical Reactive Modules and a Compositional Verification Method

Daisuke Ishii

Japan Advanced Institute of Science and Technology, Ishikawa, Japan
dsksh@jaist.ac.jp

Abstract. The compositional approach is important for reasoning about large and complex systems. In this work, we address synchronous systems with hierarchical structures, which are often used to model cyber-physical systems. We revisit the theory of reactive modules and reformulate it based on hypergraphs to clarify the parallel composition and the hierarchical description of modules. Then, we propose an automatic verification method for hierarchical systems. Given a system description annotated with assume-guarantee contracts, the proposed method divides the system into modules and verifies them separately to show that the top-level system satisfies its contract. Our method allows an input to be a circular system in which submodules mutually depend on each other. Experimental result shows our method can be effectively implemented using an SMT-based model checker.

1 Introduction

Synchronous reactive systems are a basic model used in the design of *cyber-physical systems (CPSs)* [3], which are typically described as a feedback loop model with plant and digital controller modules. Although it is important for industrial CPS products to formally verify the safety of their models, the effort has not been sufficient. The scale and complexity of the models are hampered by the expertise and computational complexity required by formal method tools.

A divide-and-conquer approach could be the cure for scalability. The theory of reactive modules [6, 3] provides a foundation of the compositional reasoning [16]. It formalizes a system as a set of *modules* (or agents or components) that behave synchronously on a sequence of rounds and enables to verify the *implementation* (or refinement) relation between composite modules. In the verification, the *assume-guarantee rule* [6] is crucial to reason about circular systems in which submodules depend on each other. Although it has been studied for decades, the theory is still underutilized in practice due to its discrepancy from the actual CPS descriptions and the lack of automated verification methods.

In this paper, we consider the verification of synchronous system models that are composed of reactive modules $M_1, \ldots, M_n$. Assuming that each module M_j is given a contract $(M_{a.j}, M_{g.j})$, consisting of assume (a) and guarantee (g)

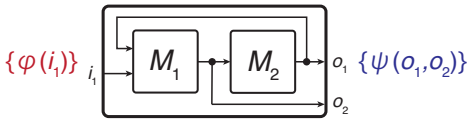


Fig. 1. Example system.

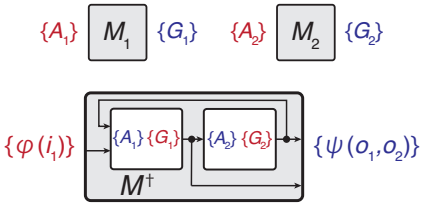


Fig. 2. Compositional verification.

properties, and satisfies it (we denote the fact by $M_j||M_{a.j} \preceq M_{g.j}$), we verify that the top-level system $M_1||\cdots||M_n$ satisfies its contract (M_a, M_g). As is the case in [2, 8, 14], verification based on the assume-guarantee contracts requires an interactive proof process. An automatic method [12, 11] has been proposed that abstracts submodules using their contracts and efficiently performs model checking. However, because of the abstraction, the method has the disadvantage of finding spurious counterexamples, especially when dealing with circular systems.

The objective of this research is to bridge the gap between the theory of reactive modules and the hierarchical design of practical systems, and to propose an automated compositional verification method based on the theory. Our contributions are summarized as follows:

1. We formalize the hierarchical structure commonly used in practical modeling languages, e.g. Lustre and Simulink, as a composition of reactive modules. Our formulation features the use of hypergraphs to describe hierarchical structures. We extend the definition of modules for hierarchization and show how it differs from parallel composition.
2. We propose a compositional verification method that transforms a hierarchical module $M[M_1, \ldots, M_n]$ into a form of a composition $M_1||\cdots||M_n||M^\dagger$ and then checks that it satisfies a given contract (M_a, M_g). We show how to validate the implementation relation $M[M_1, \ldots, M_n]||M_a \preceq M_g$ automatically. The effectiveness of the method is confirmed by applying our implementation to several examples.

Example. Hierarchical modules can be illustrated as a flow diagram in Fig. 1 in which rectangles represent modules e.g. M_1; outer rectangle represents a hierarchical module $M[M_1, M_2]$. Each module is equipped with input, output, and hidden state variables, and is interpreted as a reaction relation between their values. We consider to verify that the module satisfies a contract $(\varphi(i_1), \psi(o_1, o_2))$, assuming the submodules are given sub-contracts (A_1, G_1) and (A_2, G_2). A compositional verification can be done in three ways:

– The method based on the reactive module theory regards $M[M_1, M_2]$ as a parallel composition $M_1||M_2||\cdots$ (if other modules are used, they should be composed together). Then, we deduce that the system satisfies the top-level contract by applying the inference rules to the assumptions, along with

the composition structure. As the number of modules increases, the proof becomes more complex.
- The method based on abstraction [11] regards each submodule M_j as a reaction relation, e.g. $\mathsf{H}\, A_j \Rightarrow G_j$ described with a past fragment of LTL, and verifies $M[M_1, M_2]$ as a whole. Since there is a circular wiring in Fig. 1, this method may result in a spurious counterexample.
- The proposed method decomposes $M[M_1, M_2]$ as $M_1||M_2||M^\dagger$ (Fig. 2) where the module $M^\dagger$ represents the top-level description content equipped with interface variables with M_1 and M_2. We formalize $M^\dagger$ using hypergraphs and propose how to properly give sub-contracts to it. Then, we show that if the verification for the submodules M_1, M_2 and $M^\dagger$ succeeds, then the verification goal for $M[M_1, M_2]$ is also valid.

Paper organization. Sect. 2 introduces the basics of the theory of reactive modules, which is reformulated using hypergraphs. Sect. 3 describes a formalization of hierarchical modules. Sect. 4 presents a proposed method that transforms hierarchical modules into decomposed forms. Sect. 5 describes a prototype implementation of the method and Sect. 6 reports an experimental result. Sect. 7 describes the related work.

Preliminaries. We assume a basic knowledge of *directed hypergraphs* (V, E) where V and E are sets of vertices and *(hyper)edges* (or hyperarcs), respectively. Each hyperedge consists of two lists of vertices called the *source* (or head) and the *target* (or tail), respectively. Given $e \in E$, we denote by $\mathrm{src}(e)$ and $\mathrm{tgt}(e)$ the sets of the vertices in the source and target. We call a vertex v *initial* if $\forall e \in E, v \notin \mathrm{tgt}(e)$ and *terminal* if $\forall e \in E, v \notin \mathrm{src}(e)$. For details of the hypergraph theory, see e.g. [9, 19].

2 Reactive Modules

This section is a run-through introduction to the basics of the reactive module theory [6, 3], which is modified for our purpose.

We consider *variables* typed as `unit`, `bool`, `int`, etc., referring to the *domains* $\mathcal{D}(\texttt{unit}) = \{()\}$, $\mathcal{D}(\texttt{bool}) = \{\top, \bot\}$, $\mathcal{D}(\texttt{int}) = \mathbb{Z}$, etc. Given a variable v of type t, we denote its *evaluation* by $[\![v]\!] \in \mathcal{D}(t)$. Given a set of variables $V = \{v_1, \ldots, v_n\}$ and a family of types $\{t_v\}_{v \in V}$, we denote by $\mathcal{D}(V)$ the *domain*,

$$\prod_{v \in V} \mathcal{D}(t_v) \quad \text{if } V \neq \emptyset, \qquad \{()\} \quad \text{if } V = \emptyset,$$

i.e. a Cartesian product of the family $\{\mathcal{D}(t_v)\}_{v \in V}$. For variable sets V ($\neq \emptyset$) and $W \subseteq V$, and a subdomain $D = \Pi_{v \in V} D_v \subseteq \mathcal{D}(V)$, $\pi_W(D)$ denotes the projection onto W i.e. $\Pi_{v \in W} D_v$.

2.1 Task hypergraphs

As a unit to describe a composite system, we consider stateless tasks that are non-blocking and can be nondeterministic.

Definition 1. *Let R and W be finite and mutually disjoint sets of variables. A* task *e with a* read set *R and a* write set *W is represented by a total relation in $\mathcal{D}(R) \times \mathcal{D}(W)$ such that $\forall r \in \mathcal{D}(R), \exists w \in \mathcal{D}(W), (r, w) \in e$. We also denote a task by $e(R, W)$ to clarify its read and write sets.*

For example, a task $e(\emptyset, W)$ represents a task that outputs a constant or a nondeterministically chosen value in $\mathcal{D}(W)$. Given $e : \mathcal{D}(R) \to \mathcal{D}(W)$, it can represent a task that applies the function to the value of R and writes it to W.

In this paper, we propose to formalize a composite task description as a hypergraph representing a network of tasks connected via read and write variables. It aims to be an extension of task graphs in [3] to depict relations between both tasks and variables.

Definition 2. *A* task (hyper)graph *(TG) (V, E) is a directed hypergraph whose vertices represent variables and hyperedges represent tasks. We assume that* (i) *(V, E) is acyclic,* (ii) *no vertex is isolated,* (iii) *each vertex has at most one incoming edge,* (iv) *for a task $e \in E$ such that $e \in \mathcal{D}(R) \times \mathcal{D}(W)$ and a variable $v \in V$, $v \in \mathrm{src}(e)$ iff $v \in R$ and $v \in \mathrm{tgt}(e)$ iff $v \in W$.*

If it is clear from the context, we do not distinguish between vertices and variables, or hyperedges and tasks (relations in the variable domains), respectively. *Precedence relation* between tasks (denoted by $e \prec e'$ in [3]) and *await dependency* between variables ($v \succ v'$ in [6, 3]) are represented by the existence of a path between the two in the graph. The condition (iii) prevents conflicts between writes to a variable by multiple tasks.

Example 3. Fig. 4 illustrates an example TG. Each dot (with or without circle) represents a vertex in $\{i_1, i_2, o_1, o_2, s_1, s'_1, l_1\}$ and each set of directed lines mediated by a numbered circle represents a hyperedge in $\{e_1, e_2, e_3\}$. For instance, $\mathrm{src}(e_2) = \{i_2, s_1\}$ and $\mathrm{tgt}(e_2) = \{l_1\}$. The hypergraph in Fig. 9b is not a TG since it contains a cycle.

TGs can be regarded as total relations i.e. tasks.

Definition 4. *Let (V, E) be a TG, R the set of initial vertices, and W the set of vertices such that $W \subseteq V \setminus R$. We consider a relation*

$$\exists v_1 \in \mathcal{D}(t_1), \cdots \exists v_m \in \mathcal{D}(t_m),\ e_1(R_1, W_1) \wedge \cdots \wedge e_n(R_n, W_n), \tag{1}$$

where $\{v_1, \ldots, v_m\} = V \setminus (R \cup W)$, $t_1, \ldots, t_m$ are their types, and $\{e_1, \ldots, e_n\} = E$. We denote the set of all such relations represented with a TG by $\mathcal{T}(R, W)$.

Note that variables in R may always be the initial in the TG, but those in W are not necessarily the terminal (they are shown as circled dots in the figures). Every variable in R_j or W_j $(j = 1, \ldots, n)$ not included in $R \cup W$ are bound by a quantifier in Eq. (1).

Lemma 5. *Every relation in $\mathcal{T}(R, W)$ is total.*

Proof. The vertices in a TG can be partially ordered by the lengths of the longest paths from any initial vertex; we group the vertices according to the ordering. The initial vertices in R belong to the first group. The vertices written by tasks with the empty read set belong to the second group. Then, we check $\forall u_r \in \mathcal{D}(R), \exists u_w \in \mathcal{D}(W), \mathcal{R}(u_r, u_w)$ holds where $\mathcal{R}$ represents the relation (1). We check by induction that a value exists for vertices in every group to satisfy the relation. The first group is universally quantified, so any values can be assigned to the vertices. Assuming that the previous groups has been assigned values, the values for the next group are determined by the incoming tasks. □

Although tasks in [3] are stateful, we formulate them as stateless. Modules defined later specify the state variables among the read/write set of tasks and properly manage the states. *Atoms* are used instead of tasks in [6] to represent the initialization of the state of a module when executed and the reactions in each round. We embed the initial conditions in modules and represent only the reactions with TGs.

2.2 Modules, implementation relation, and parallel composition

Reactive and synchronous systems are formalized as compositional modules (called *components* in [3]) executed in a series of rounds.

Definition 6 ([6, 3]). *A* module *is a tuple* $(I, O, S, \mathcal{I}, \mathcal{R})$ *where* I*,* O *and* S *are mutually disjoint sets of* input*,* output *and* state *variables.* $\mathcal{I} \subseteq \mathcal{D}(S)$ *is an* initial condition*, and* $\mathcal{R}$ *is a* reaction relation *that is a TG in* $\mathcal{T}(S \cup I, O \cup S')$*, where* S' *represents a set of variables renamed from* S*. An* execution *of a module is a sequence of reactions*

$$s(-1) \xrightarrow{i(0)/o(0)} s(0) \xrightarrow{i(1)/o(1)} s(1) \cdots \quad = \\ \{(s(j-1), i(j), o(j), s(j)) \in \mathcal{R} \mid j \in \{0\} \cup \mathbb{N}, s(-1) \in \mathcal{I}\}.$$

A trace *of an execution is a sequence of values* $(i(0), o(0))\,(i(1), o(1)) \cdots$*, i.e. the projection onto* $I \cup O$*.*

Note that $\mathcal{R}$ is interpreted as a set of quadruples of values, each of which is a family of values indexed by S, I, O or S'. We assume the state variables in S and S' always be the initial and terminal vertices of $\mathcal{R}$, respectively. Differently from the formalization in [3], which embeds state variables within tasks, modules must designate the vertices representing state variables from among the TG's initial and terminal vertices. Hereafter, for a module M_i with an identifier i, we denote its elements by e.g. I_i and $\mathcal{R}_i$.

If I, O or S of a module is empty, there may be an execution that involves the value $()$. For example, the module $M_\top$ defined by $(\emptyset, \emptyset, \emptyset, \{()\}, \{((), (), (), ())\})$ has an execution represented by a sequence of $((), (), (), ())$.

As a graphical language to describe modules (e.g. Simulink), we consider *signal flow diagrams* (SFDs). An SFD consists of rectangles and directed lines

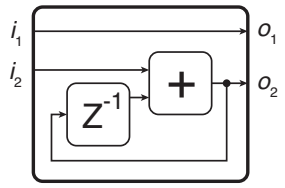


Fig. 3. A signal flow diagram describing the module $M_{\text{Ex.7}}$.

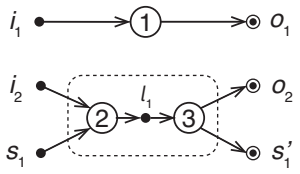


Fig. 4. The task hypergraph $\mathcal{R}_{\text{Ex.7}}$.

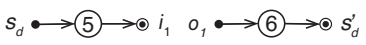


Fig. 6. $\mathcal{R}_{\text{delay}}$.

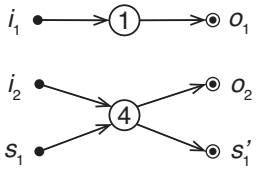


Fig. 5. Another TG for Ex. 7.

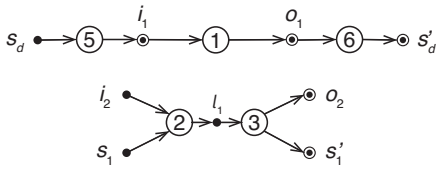


Fig. 7. The TG of $M_{\text{Ex.7}}||M_{\text{delay}}$.

annotated with variable names and other labels. The rectangles represent modules that can be stateful and the lines represent synchronous communication between the processes. Input signals are connected to the left side and output signals are extracted from the right side of rectangles.

Example 7. The SFD of a counter constructed with an addition module and a delay module is shown in Fig. 3, which consists of $I_{\text{Ex.7}} = \{i_1, i_2\}$, $O_{\text{Ex.7}} = \{o_1, o_2\}$, $S_{\text{Ex.7}} = \{s_1\}$, $\mathcal{I}_{\text{Ex.7}}(s_1) \equiv (\llbracket s_1 \rrbracket = 0)$, and $\mathcal{R}_{\text{Ex.7}}$ represents a function $(\llbracket i_1 \rrbracket, \llbracket i_2 \rrbracket, \llbracket s_1 \rrbracket) \mapsto (\llbracket i_1 \rrbracket, \llbracket i_2 \rrbracket + \llbracket s_1 \rrbracket, \llbracket i_2 \rrbracket + \llbracket s_1 \rrbracket)$. Fig. 4 shows $\mathcal{R}_{\text{Ex.7}}$ as a TG. The vertices in $O_{\text{Ex.7}}$ are shown as circled dots and the vertex l_1 represents a local variable that does not belong to $I_{\text{Ex.7}}$, $O_{\text{Ex.7}}$ or $S_{\text{Ex.7}}$. The hyperedges e_1, e_2 and e_3 represent the identity, addition and copy functions. An execution can be

$$0 \xrightarrow{(\top,1)/(\top,1)} 1 \xrightarrow{(\bot,0)/(\bot,1)} 1 \xrightarrow{(\top,2)/(\top,3)} 3 \cdots .$$

Example 8. There can be multiple TGs representing a reaction relation. Fig. 5 shows another TG for $\mathcal{R}_{\text{Ex.7}}$ in which e_4 outputs two copies of the sum.

We also represent a safety property of a coexisting module as a module. A safety property $\varphi(v)$ involving a list of variables $v = (v_1, \ldots, v_n)$ is a module $M_{\varphi(v)}$ with $O_{\varphi(v)} = \{v_1, \ldots, v_n\}$, which nondeterministically outputs a signal satisfying $\varphi(v)$.

Example 9. We can consider an invariance property $\mathsf{G}\, o_2 \geq 0$ of $M_{\text{Ex.7}}$ where G is the "always/globally" operator of LTL. Output signals represented by variable o_2 of the module $M_{\text{Ex.7}}$ must be mimicked by $M_{\mathsf{G}\, o_2 \geq 0}$.

Next, we introduce the implementation (or refinement) relation and the composition mechanism for the modules.

Definition 10 ([6]). *Let M_1 and M_2 be modules. We say M_1* implements *M_2, denoted by $M_1 \preceq M_2$, if* (i) $O_2 \subseteq O_1$, (ii) $I_2 \subseteq I_1 \cup O_1$, (iii) *the await dependency of $y \in O_2$ on $x \in I_2 \cup O_2$ in $\mathcal{R}_2$ (i.e. $y \succ x$) is preserved in $\mathcal{R}_1$, and* (iv) *for every trace t of M_1, the projection of t onto $I_2 \cup O_2$ is a trace of M_2. We denote $M_1 \preceq M_2 \wedge M_2 \preceq M_1$ by $M_1 \cong M_2$.*

Lemma 11 ([6]). *The implementation relation is a preorder.*

Definition 12 ([6, 3]). *We say modules M_1 and M_2 are* compatible *if* (i) $O_1 \cap O_2 = S_1 \cap S_2 = \emptyset$ *and* (ii) $\mathcal{R}_1 \cup \mathcal{R}_2$ *(the union as graphs) is acyclic. The* parallel composition (PC) *$M_1||M_2$ is a module $(I, O, S, \mathcal{I}, \mathcal{R})$, where $I = (I_1 \cup I_2) \setminus O$, $O = O_1 \cup O_2$, $S = S_1 \cup S_2$, $\mathcal{I} = \mathcal{I}_1 \times \mathcal{I}_2$ and $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$.*

Example 13. Consider composing the module in Ex. 7 with a module M_{delay} that represents a delay task whose TG is Fig. 6. The TG of PC $M_{\text{Ex.7}}||M_{\text{delay}}$ is shown in Fig. 7.

Lemma 14 ([6]). *The PC operation is associative, transitive and symmetric.*

To deal with extra output variables added by the PC operation, we introduce the hiding operator.

Definition 15 ([6, 3]). *Given a module M and a subset of output variables $O' \subseteq O$,* hiding *of O' in M, denoted $M \setminus O'$, is a module consisting of the same elements as M but excluding O' from O.*

2.3 Compositional verification

We consider to verify that a module M fulfils an *assume-guarantee contract* (M_a, M_g), i.e. a pair of modules. For that purpose, we can show $M||M_a \preceq M_g$ holds. In our experiment in Sect. 6, we consider contracts consisting of safety properties.

Example 16. $M_{\text{Ex.7}}||M_{\mathsf{G}i_2 \geq 0} \preceq M_{\mathsf{G}o_2 \geq 0}$ holds.

In this paper, we consider the verification of systems composed of n submodules. Here, we assume that each submodule satisfies a given contract, and aim at efficiently verifying the fact that the entire system fulfils the top-level contract by utilizing the assumptions.

Definition 17. *A* compositional verification problem *consists of n modules $M_1, \ldots, M_n$, a top-level contract (M_a, M_g) and n sub-contracts $(M_{a.j}, M_{g.j})$ where $j = 1, \ldots, n$; we assume $M_j||M_{a.j} \preceq M_{g.j}$ for every j. The* goal *is a condition $M_1|| \cdots ||M_n||M_a \preceq M_g$.*

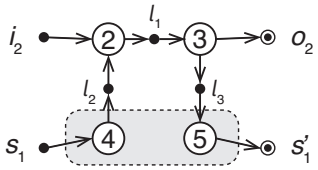


Fig. 8. A detailed TG for Ex. 7.

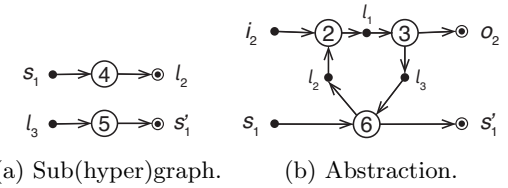


Fig. 9. Separation of a hypergraph.

The module $M_\top$ can be used to omit some elements of contracts. For arbitrary M, $M \preceq M_\top$ and $M||M_\top \cong M$ hold.

The following lemma provides two basic inference rules for the compositional reasoning on modules. The second rule allows a PC $M_1||M_2$ to be *circular* i.e. $I_1 \cap O_2$ and $I_2 \cap O_1$ are nonempty.

Lemma 18 ([6]). *Let M_1, M_2, M_3 and M_4 be modules, where M_1 and M_2, and M_3 and M_4 are respectively compatible, and $I_3 \cup I_4 \subseteq I_1 \cup I_2 \cup O_1 \cup O_2$.* (i) *$M_1||M_2 \preceq M_1$.* (ii) *If $M_1||M_3 \preceq M_4$ and $M_2||M_4 \preceq M_3$, then $M_1||M_2 \preceq M_3||M_4$.*

Example 19. We consider a compositional verification problem of the system $M_{\text{Ex.7}}||M_{\text{delay}}$ in Ex. 13, which consists of

- submodules $M_{\text{Ex.7}}$ and M_{delay},
- top-level contract $(M_\top, M_{\mathsf{G}o_2 \geq 0})$, and
- sub-contracts $(M_{\mathsf{G}i_2 \geq 0}, M_{\mathsf{G}o_2 \geq 0})$ and $(M_{\mathsf{G}o_2 \geq 0}, M_{\mathsf{G}i_2 \geq 0})$.

The goal $M_{\text{Ex.7}}||M_{\text{delay}}||M_\top \preceq M_{\mathsf{G}o_2 \geq 0}$ is provable. First, we can deduce as follows.

$$\dfrac{\dfrac{\begin{array}{c} M_{\text{Ex.7}}||M_{\mathsf{G}i_2 \geq 0} \preceq M_{\mathsf{G}o_2 \geq 0} \\ M_{\text{delay}}||M_{\mathsf{G}o_2 \geq 0} \preceq M_{\mathsf{G}i_2 \geq 0} \end{array}}{M_{\text{Ex.7}}||M_{\text{delay}} \preceq M_{\mathsf{G}i_2 \geq 0}||M_{\mathsf{G}o_2 \geq 0}}\,\text{Lem.18(ii)} \qquad \dfrac{}{M_{\mathsf{G}i_2 \geq 0}||M_{\mathsf{G}o_2 \geq 0} \preceq M_{\mathsf{G}o_2 \geq 0}}\,\text{Lem.18(i)}}{M_{\text{Ex.7}}||M_{\text{delay}} \preceq M_{\mathsf{G}o_2 \geq 0}}\,\text{Trans.}$$

Then, the goal follows from $M_{\text{Ex.7}}||M_{\text{delay}}||M_\top \preceq M_{\text{Ex.7}}||M_{\text{delay}}$.

3 Hierarchical Reactive Modules

Practical languages for describing modules (e.g. Lustre and Simulink) tend to support hierarchical structures. We formalize such structures by allowing a TG to be hierarchical, separating a subhypergraph of the TG as a submodule.

Definition 20. *For a TG (V, E), its* sub(hyper)graph *is a TG such that $V' \subseteq V$ and $E' \subseteq E$. Assume a subgraph (V', E') of (V, E) is in $\mathcal{T}(R, W)$. Then, the* abstraction of (V', E') in (V, E) *is a hypergraph (V'', E'') where V'' is a set such that $V' \cup V'' = V$ and $E'' = (E \setminus E') \cup \{e\}$ where e is a fresh hyperedge such that $\mathrm{src}(e) = R$ and $\mathrm{tgt}(e) = W$.*

According to [9], subgraphs defined above are partial subhypergraphs without isolated vertex. Note that an abstraction of a TG may contain a cycle as shown in the following example.

Example 21. Consider a hypergraph in Fig. 8 that describes yet another TG (lower part) for Ex. 7. The graph in Fig. 9a is its subgraph that consists of the separated edges e_4 and e_5. The abstraction of the subgraph in Fig. 8 is shown in Fig. 9b, in which e_6 is associated with the subgraph.

Now, we consider the TGs of modules to be hierarchical. Intuitively, a hierarchical module is a module that separates subgraphs as submodules. To ensure the consistency among decomposed descriptions, and because of the proposed method, we assume several conditions.

Definition 22. *Let $M_1, \ldots, M_n$ be modules. A* hierarchical *module $M[M_1, \ldots, M_n]$ (also denoted by M or $M[M_1..M_n]$) is a module that satisfies the following conditions for each submodule M_j:* (i) $I \cap I_j = O \cap O_j = \emptyset$, (ii) $S_j \subseteq S$, (iii) $\pi_{S_j}(\mathcal{I}) \equiv \mathcal{I}_j$, *and* (iv) $\mathcal{R}_j$ *is a subgraph of* $\mathcal{R}$.

The condition (i) forces a submodule to handle its own input and output variables to facilitate the separation of the submodules in Sect. 4. The input and output variables must be copied to local variables before communicating with submodules. The conditions (ii) and (iii) make the state variables of a submodule shared with the parent and are maintained properly. A hierarchical module is interpreted as a *flattened* module whose TG embeds the submodules' TGs. In order for a hierarchical module $M[M_1..M_n]$ and its submodules $M_1, \ldots, M_n$ to be interpreted as modules properly, the TGs of the submodules must avoid a write conflict (Def. 2) and their states must be managed with separate variables, i.e., $\bigcap_{j=1,\ldots,n} O_j = \bigcap_{j=1,\ldots,n} S_j = \emptyset$ must hold.

Example 23. We consider a hierarchical module $M_{\text{Ex.23}}[M_1, M_2]$ that has two counter modules as submodules (Fig. 10). We assume they are the same as in Ex. 7, except for the variable names. Its TG is shown in Fig. 11 where the dashed frames enclose the hyperedges in the subgraphs. The state variables of the submodules are inherited to $M_{\text{Ex.23}}$ ($S_{\text{Ex.23}} = \{s_1, s_{1.1}, s_{2.1}\}$) and $\mathcal{I}_{\text{Ex.23}}$ is set as $\{\top\} \times \mathcal{I}_1 \times \mathcal{I}_2$. $M_{\text{Ex.23}} || M_{\mathsf{G}i_1 \geq 0} \preceq M_{\mathsf{G}o_1 \geq 0}$ holds.

Abstraction of hierarchical modules. The hierarchization of modules can be viewed as an abstraction as shown in Ex. 21 (Fig. 9b). Naively, we can replace a fragment of the TG that belongs to a submodule with a hyperedge, which is then regarded as a complete graph between input and output vertices. Then, it may help to efficiently search for a counterexample that can be executed by the parent module. The Kind2 model checker [11] abstracts each submodule M_j with a hyperedge representing a property $\mathsf{H}\,\varphi(i_j) \Rightarrow \psi(o_j)$, where H is the pLTL "historically" modality, to exploit the contract given as a pair of safety properties $(\varphi(i_j), \psi(o_j))$ (where i_j/o_j is a variable list in I_j/O_j). Separately, the fact $M_j || M_{\varphi(i_j)} \preceq M_{\psi(o_j)}$ has to be verified to check that the submodule M_j satisfies the contract.

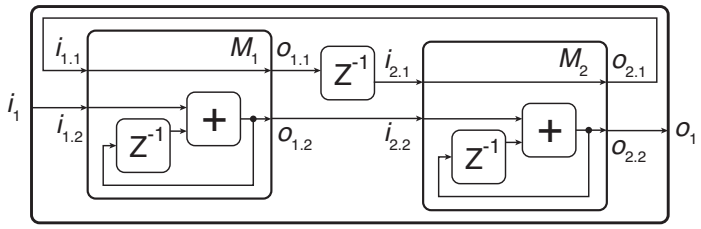


Fig. 10. A hierarchical module $M_{\text{Ex.23}}$.

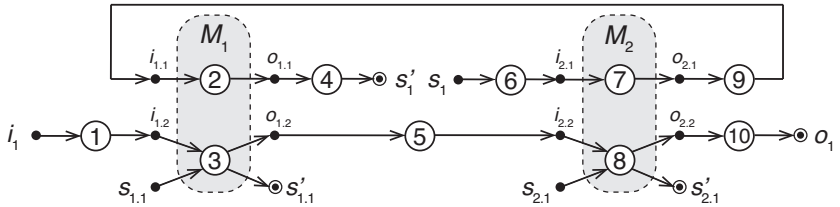


Fig. 11. The TG of $M_{\text{Ex.23}}$.

Example 24. Let $\varphi(x) \equiv \mathsf{G}\, x \geq 0$. For the submodules M_j of $M_{\text{Ex.23}}$ ($j = 1, 2$), $M_j || M_{\varphi(i_{j.2})} \preceq M_{\varphi(o_{j.2})}$ holds. Then, the fact $M_{\text{Ex.23}} || M_{\varphi(i_1)} \preceq M_{\varphi(o_1)}$ can be verified with an abstraction that replaces each submodule M_j with $M_{\mathsf{H}\, \varphi(i_{j.2}) \Rightarrow \varphi(o_{j.2})}$.

While the abstraction methods enable a verification process that leverages the sub-contracts, they cannot properly handle circular systems whose submodules assume the existence of other submodules. In such cases, the abstraction approach might not work; the process may detect spurious counterexamples.

Example 25. Consider the verification of $\varphi(x) \equiv \mathsf{G}\, x \geq 0$ on $M_{\text{Ex.23}}$ again. Let $\psi(x_1, x_2) \equiv \mathsf{G}(x_1 \wedge x_2 \geq 0)$. For the submodules M_j ($j = 1, 2$), $M_j || M_{\psi(i_{j.1}, i_{j.2})} \preceq M_{\psi(o_{j.1}, o_{j.2})}$ holds. So, we can abstract the subgraph for M_j with $\mathsf{H}\, \psi(i_{j.1}, i_{j.2}) \Rightarrow \psi(o_{j.1}, o_{j.2})$ in the TG of $M_{\text{Ex.23}}$. Then, the verification of $M_{\text{Ex.23}} || M_{\varphi(i_1)} \preceq M_{\varphi(o_1)}$ will result in a false counterexample such that $[\![i_{1.1}]\!] = [\![i_{2.1}]\!] = \bot$.

4 Compositional Verification of Hierarchical Modules

In this section, we propose a compositional verification method that can handle circular hierarchical modules. The method validates that a module $M[M_1..M_n]$ satisfies a contract (M_a, M_g). We describe how to validate the goal under the assumption that every submodule M_j ($j \in \{1, \ldots, n\}$) satisfies its contract $(M_{a.j}, M_{g.j})$. The key idea is to prepare a module $M^\dagger$ called *adapter* by extracting only the top-level part of the hierarchical TG. The subgraphs are separated from the top-level TG and the variables that correspond to the boundary vertices

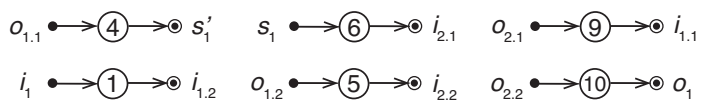


Fig. 12. The TG of $M^{\dagger}_{\mathrm{Ex.23}}$.

between the top-level part and the subgraphs are set as input and output variables of the adapter module. The adapter $M^{\dagger}$ is prepared in a way $M[M_1..M_n]$ and $M_1||\cdots||M_n||M^{\dagger}$ be isomorphic, and thus yields a compositional verification problem (Def. 17).

Definition 26. *For sets $V_1, \ldots, V_n$, we denote their union by $V_{\{1..n\}}$. The* adapter *$M^{\dagger}$ of a hierarchical module $M[M_1..M_n]$ is a module with the components $I^{\dagger} = I \cup O_{\{1..n\}}$, $O^{\dagger} = O \cup I_{\{1..n\}}$, $S^{\dagger} = S \setminus S_{\{1..n\}}$, $\mathcal{I}^{\dagger} = \pi_{S^{\dagger}}(\mathcal{I})$, and $\mathcal{R}^{\dagger}$ obtained from $\mathcal{R}$ by removing the hyperedges e_j and the vertices corresponding with the variables in S_j for every j.*

Example 27. The TG of $M^{\dagger}_{\mathrm{Ex.23}}$ is with $R = \{i_1, o_{1.1}, o_{1.2}, o_{2.1}, o_{2.2}, s_1\}$ and $W = \{o_1, i_{1.1}, i_{1.2}, i_{2.1}, i_{2.2}, s'_1\}$. It is illustrated as Fig. 12.

A hierarchical module can now be regarded as a PC of decomposed modules (up to $\cong$ in Def. 10). However, since extra output variables of the submodules are added by the PC, they must be hidden (Def. 15).

Lemma 28. *Let $M[M_1..M_n]$ be a hierarchical module and M' be $M_1||\cdots||M_n||M^{\dagger}$. Then, $M[M_1..M_n] \cong M' \setminus (I_{\{1..n\}} \cup O_{\{1..n\}})$.*

Proof. In Def. 26, the variable sets $I_{\{1..n\}}$ and $O_{\{1..n\}}$ are added to the adapter as initial and terminal vertices in $\mathcal{R}^{\dagger}$. Then, the PC of M_j and $M^{\dagger}$ merges $\mathcal{R}_j$ and $\mathcal{R}^{\dagger}$ by matching the vertices for the variables in $I_j \cup O_j$. Hence, the resulting TG is equivalent of replacing e_j with $\mathcal{R}_j$ in Def. 22 (ii). Each set of the variables is equal to that of $M[M_1..M_n]$ by the hiding operation. □

Once a hierarchical module is decomposed into a PC of submodules and an adapter, it is possible to perform a compositional verification of facts about the module based on the theory of reactive modules. To do so, we can make a proof using the inference rules in Sect. 2, which may require a manual effort. However, the following theorem shows that such verification always succeeds if the conditions on the submodules and the adapter are valid.

Theorem 29. *Consider a goal $M[M_1..M_n] \,||\, M_a \preceq M_g$. If* (a) *$M_j||M_{a.j} \preceq M_{g.j}$ for every j and* (b) *$M^{\dagger}||M_a||M_{g.1}||\cdots||M_{g.n} \preceq M_g||M_{a.1}||\cdots||M_{a.n}$, then the goal holds.*

Proof. Let $M_{[j..k]}$ represent $M_j||\cdots||M_k$ if $j \leq k$ and the empty module $M_{\top}$ if $j > k$ (we use the same notation for $M_{a.[j..k]}$ and $M_{g.[j..k]}$). We rewrite the condition (b) as follows (we assume $j = n$ initially):

$$M_{[(j+1)..n]}||M^{\dagger}||M_a||M_{g.[1..j]} \preceq M_g||M_{a.[1..j]}. \tag{2}$$

From (a), $(M_j||M_{a.j})||(M_{a.[1..(j-1)]}||M_g) \preceq M_j||M_{a.j}$ (Lem. 18 (i)), and the transitivity of $\preceq$, we have

$$M_j||M_g||M_{a.[1..j]} \preceq M_{g.j}. \tag{3}$$

From (2), (3) and Lem. 18 (ii), we have

$$M_{[j..n]}||M^\dagger||M_a||M_{g.[1..(j-1)]} \preceq M_g||M_{a.[1..j]}||M_{g.j}.$$

The rhs can be simplified to $M_g||M_{a.[1..(j-1)]}$ (by Lem. 18 (i) and the transitivity). By repeating the above for $j = n-1, \ldots, 1$, we will obtain the fact, which is equivalent to the goal due to Lem. 28. □

The proposed method can be regarded as transferring the proof process along with top-level's compositional structure to the verification process of the condition (b) of Th. 29. If the top-level can be viewed as a nested PC structure of submodules, then a proof of a hierarchical module may be obtained by applying the inference rules in Lem. 18 to the PCs. However, the proof process for a verification goal is non-trivial in general. Applying the inference rules requires a number of deductions, introduction of PCs of submodules and property modules, and adjusting the form of PC terms while checking module compatibility.

The proposed method provides an automated process for the compositional verification problem. As implemented in the next section, the separation of adapters can be automated, and the entire process can also be automated when combined with an automatic verifier for submodules.

When a system contains multiple hierarchies, we can simply repeat the process to check the two conditions in Th. 29 in a bottom-up fashion to verify the whole system. Each process for a hierarchical module decomposes it after deriving an adapter, then either verifies the contract for the PC of the submodules or composes them with other submodules of the parent.

5 Implementation

We have implemented the proposed method by extending the Kind2 tool.

5.1 Lustre, CoCoSpec, and Kind2

Kind2 (version 1.6.0)[11] is an SMT-based model checking tool (we used Z3 4.12.4 as an SMT solver). Its input language is *Lustre* [10], a textual language for describing hierarchical synchronous modules, and the modules can be annotated contracts with the *CoCoSpec* language [12].

Example 30. Fig. 13 shows an example described with Lustre and CoCoSpec. It is a module similar to Ex. 23 in which the counters are replaced with second-order digital filters. A Lustre node is defined by a section started with `node`, which is followed by

- the node name (e.g. `Filter` and `Toplevel`),

```
node Filter (in1 : bool; in2 : real)
returns (out1 : bool; out2 : real);
(*@contract
  assume    in1;  assume    -1.0 <= in2 and in2 <= 1.0;
  guarantee out1; guarantee -1.0 <= out2 and out2 <= 1.0;
*)
var sum, D1, D2: real;
let
  out1 = in1;
  sum = 0.0582*(if in1 then in2 else -in2) - (-1.49*D1) - 0.881*D2;
  D1 = 0.0 -> pre sum;      D2 = 0.0 -> pre D1;
  out2 = (sum - D2) / 1.25;
tel

node Toplevel (in : real) returns (out : real);
(*@contract
  assume    -1.0 <= in and in <= 1.0;
  guarantee -1.0 <= out and out <= 1.0;
*)
var b1, b2, pre_b2 : bool; s1 : real;
let
  b1, s1 = Filter(b2, in);  pre_b1 = true -> pre b1;
  b2, out = Filter(pre_b1, s1);
  --%MAIN;
tel
```

Fig. 13. An example Lustre program annotated with CoCoSpec.

- the input variable list (in parentheses),
- the output variable list (with keyword `returns`),
- the contract annotation (described within a comment),
- the local variable list (with keyword `var`), and
- the body (enclosed in `let` and `tel`). The line "`--%MAIN;`" specifies that the node is a verification target.

We consider modules to be instances of Lustre nodes whose input and output variables are substituted by the arguments. The above program describes the verification conditions $M_{\mathtt{F0}}||M_{\mathtt{F0}.a} \preceq M_{\mathtt{F0}.g}$ and $M_{\mathtt{T0}}[M_{\mathtt{F1}}, M_{\mathtt{F2}}]||M_{a.\mathtt{T0}} \preceq M_{g.\mathtt{T0}}$ where $M_{\mathtt{N}i}$ represents the ith instance of the node `N` and $M_{a.\mathtt{N}i}$ and $M_{g.\mathtt{N}i}$ represent the annotated properties (we abbreviate `Filter` and `Toplevel` as `F` and `T`).

When an input describes a goal (verification condition for the target module) $M[M_1..M_n]||M_a \preceq M_g$ and conditions for submodules $M_j||M_{a.j} \preceq M_{g.j}$ where $j = 1, \ldots, n$, Kind2 is able to verify its validity in three modes:

- *Monolithic mode* that interprets the target $M[M_1..M_n]$ as a module (cf. Def. 22) and verifies only the goal.
- *Modular mode* that verifies only the conditions for submodules $M_1, \ldots, M_n$.

– *Compositional mode* that verifies the goal with an abstraction as described in Sect. 3.

We used Kind2 with the default setting, which runs several model checking algorithms e.g. BMC, k-induction and PDR, in parallel.

5.2 Implementation of the proposed method

We have implemented the proposed method in OCaml by modifying Kind2.[1] The function we have added translates a hierarchical Lustre program to a program in which the hierarchical modules are replaced with adapter modules (here, we also refer to Lustre *nodes* as modules). The input is Lustre programs annotated with CoCoSpec contracts. It generates a list of reactive modules by applying the following processes:

1. *Instantiation of module definitions.* Because a Lustre node may be invoked several times by the parent module, we instantiate a node definition with the real argument of each call.
2. *Modification of the top-level into an adapter.* For each hierarchical module, we remove the submodule invocation statements and modify the variable list as described in Def. 26.
3. *Pretty printing.* The intermediate data will be printed as a decomposed and properly annotated Lustre program.

By feeding the output Lustre program to Kind2 with the modular mode, the satisfaction of the assume-guarantee contract by each module will be checked. The success of the process implies the validity of the annotated top-level module in the original Lustre program.

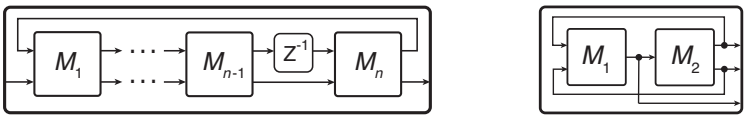
6 Experiment

To evaluate the effectiveness and the performance of the proposed method, we have conducted the verification of several examples using the implementation. The experiment was done with a MacBook Pro (10-core Apple M2 Pro chip and 32GB RAM).

We prepared several circular hierarchical modules for the experiment.

– *Feedback loop system containing n digital filters* (nFilters). This is an extended and parameterized version of Ex. 30. The `Toplevel` instance has n `Filter` instances, and they form a loop as illustrated in Fig. 14a. We gave the same assume-guarantee contracts to `Filter` and `Toplevel` as in Ex. 30 and verified that `Toplevel` satisfies the contract.

[1] The artifact is available at `https://doi.org/10.5281/zenodo.10559936` and the source code is available at `https://github.com/dsksh/kind2`.



(a) nFilters. (b) MCtrl.

Fig. 14. SFDs of the example modules.

Table 1. Execution result.

	Monolithic	Proposed method	
Example	Time	Time	#Guarantees
2Filters	204s	14.9s	7
3Filters	TO	14.9s	9
36Filters	TO	15.4s	75
MCtrl	TO	3.1s	24

– *DC motor control (MCtrl).* This is a more practical and typical example in which a motor and a controller are described as submodules M_2 and M_1 and form a feedback loop as illustrated in Fig. 14b. We annotated the top-level module with 8 safety (guarantee) properties, and submodules with 3 to 5 assume/guarantee properties.

We first verified the target hierarchical modules with the monolithic mode of Kind2. Second, we decomposed the target modules into the PC forms using the proposed method, and then verified that the submodules and the adapters satisfied their contracts (i.e. the conditions in Th. 18) using Kind2. Note that, if we verify any of the examples with the compositional mode of Kind2, it will result in a spurious counterexample as explained in Ex. 25.

Experimental result. The result is shown in Table 1. Each column shows wall clock time for the monolithic process or the process with the proposed method, and the total number of guarantee properties verified with the proposed method ("#Guarantees"). "TO" means the process did not terminate within 600s.

Discussions. Since the submodules i.e. the digital filters, motor and controller behave in a stateful manner, using Kind2 to check the safety properties requires analyzing the execution prefixes of certain lengths, which would be time-consuming. Therefore, the verification of the system at once resulted in timeouts except for 2Filters. When the proposed method verified the examples by dividing them into modules, the process was more efficient because the numbers of rounds analyzed were reduced.

In each experiment for nFilters, the proposed method verified only a `Filter` instance because n verification conditions for submodules were for the same Lustre node. Therefore, differences in the value of n should appear only in the verification of adapter module; the number of read/write variables of the module and the number of corresponding guarantee properties would increase. The

execution time increased slightly; although this was due to the simplicity of the top-level content, we consider the overhead of our method was small since increasing the number of variables did not have much effect. The effectiveness of our method was also confirmed in MCtrl.

7 Related Work

The hierarchy with nesting parallel compositions has been considered in an implementation [2] and extensions e.g. [4] of the reactive module theory. The hierarchization we consider in this paper is slightly different from the nesting of composition operations; ours corresponds to the embedding operation in dataflow diagrams. Alur et al. [5] address two kinds of hierarchies by agents and modules, and propose to perform reasoning on them separately; they do not consider a transformation between the two unlike this work.

More recently, there has been work on compositional verification of hierarchical Simulink models [8, 13, 14, 17]. As in this paper, these methods give contracts to subsystems and verify the models according to a hierarchical structure. Boström and Wiik [8] propose to convert both models and contracts to specific dataflow graphs, then to sequential programs, and perform program verification. Their method does not support models with algebraic loops, and it is not clear whether it can handle the circular systems we consider. Dragomir et al. [14, 21] introduce QLTL properties and dedicated refinement calculus to perform verification on hierarchical models interactively on Isabelle. Notably, they handle liveness properties which we do not. Their verification method requires human support unlike ours. Since they do not seem to provide any inference rules that explicitly deal with circular cases, it is unclear whether our method is applicable to their framework.

The Kind2 tool [11] supports the modular and compositional model checking of hierarchical Lustre programs as described in Sect. 3 and Sect. 5. It is limited in handling circular programs. Murugesan et al. [17] also propose a compositional model checking method for Simulink based on a similar abstraction method.

Dragomir et al. [13] propose to analyze hierarchical models with composite predicate transformers (CPTs) that use several composition operators, e.g. serial and parallel compositions and feedbacks. In comparison, we use only one composition operator and formalize the hierarchical structure in a fixed way. Tripakis et al. [22] formalize hierarchical dataflow diagrams and propose a profiling method to assure the modularity; although the subject is similar, their purpose is different from ours. Bakirtzis et al. [7] propose a general framework for various compositional CPS models, which includes concepts equivalent to modules and hierarchies and a formalization of contracts; however, they do not discuss either a transformation between the two concepts or verification methods. Fong and Spivak [15] formalize various hierarchical models of reactive systems, but do not consider synchronous behavior or assume-guarantee verification.

Automation of contract generation has been studied, e.g. [20, 1, 18]. We assume that contracts are given; the combination of our method and contract generation is a future issue.

8 Conclusion

We have formalized hierarchical synchronous systems based on the theory of reactive modules. We have then proposed a verification method that decomposes a hierarchical module into non-hierarchical modules and checks each module separately to show that the whole system satisfies the contract. As the experimental results show, the proposed method can effectively verify the systems with circular structures, which are suitable for describing plant control CPSs.

In general, compositional reasoning requires proof of the consistency among the verification results of each module, but this is not necessary when a top-level module is decomposed with our method. The proof task to analyze the system structure is delegated to the implementation relation on the adapter, and then it is efficiently discharged using a tool like Kind2.

Future work includes integrating the proposed method with other compositional methods such as for triggered modules and different-rate modules. Also, cooperation with automatic contract generation methods remains an issue.

Acknowledgements. Tien Duc Ngo contributed to the preliminary experiment of this work. This work was supported by JST, CREST Grant Number JPMJCR23M1 and JSPS, KAKENHI Grant Number 23K11969.

References

1. Abd Elkader, K., Grumberg, O., Păsăreanu, C.S., Shoham, S.: Automated circular assume-guarantee reasoning. Formal Aspects of Computing **30**(5), 571–595 (2018). https://doi.org/10.1007/s00165-017-0436-0
2. Alur, R., Hcnzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S.K., Tasiran, S.: MOCHA: Modularity in model checking. In: CAV. pp. 521–525. *LNCS* 1427 (1998). https://doi.org/10.1007/bfb0028774
3. Alur, R.: Synchronous Model. In: Principles of Cyber-Physical Systems, pp. 13–64. MIT Press (2015)
4. Alur, R., Grosu, R.: Modular Refinement of Hierarchic Reactive Machines. In: POPL. pp. 390–402 (2000). https://doi.org/10.1145/325694.325746
5. Alur, R., Grosu, R., Lee, I., Sokolsky, O.: Compositional modeling and refinement for hierarchical hybrid systems. The Journal of Logic and Algebraic Programming **68**(1-2), 105–128 (Jun 2006). https://doi.org/10.1016/j.jlap.2005.10.004
6. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods in System Design **15**(1), 7–48 (1999). https://doi.org/10.1023/A:1008739929481
7. Bakirtzis, G., Fleming, C.H., Vasilakopoulou, C.: Categorical Semantics of Cyber-Physical Systems Theory. ACM Transactions on Cyber-Physical Systems **5**(3), 1–32 (Jul 2021). https://doi.org/10.1145/3461669

8. Boström, P., Wiik, J.: Contract-based verification of discrete-time multi-rate Simulink models. Software & Systems Modeling **15**(4), 1141–1161 (Oct 2016). https://doi.org/10.1007/s10270-015-0477-x
9. Bretto, A.: Hypergraph Theory: An Introduction. Mathematical Engineering, Springer (2013). https://doi.org/10.1007/978-3-319-00080-0
10. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: A declarative language for programming synchronous systems. In: POPL. pp. 178–188 (1987)
11. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The KIND 2 Model Checker. In: CAV. pp. 510–517. *LNCS* 9780 (2016). https://doi.org/10.1007/978-3-319-41540-6˙29
12. Champion, A., Gurfinkel, A., Kahsai, T., Tinelli, C.: CoCoSpec: A mode-aware contract language for reactive systems. In: SEFM. pp. 347–366. *LNCS* 9763 (2016). https://doi.org/10.1007/978-3-319-54292-8
13. Dragomir, I., Preoteasa, V., Tripakis, S.: Compositional Semantics and Analysis of Hierarchical Block Diagrams. In: SPIN. pp. 38–56. *LNCS* 9641, Springer (2016). https://doi.org/10.1007/978-3-319-32582-8˙3
14. Dragomir, I., Preoteasa, V., Tripakis, S.: The Refinement Calculus of Reactive Systems Toolset. International Journal on Software Tools for Technology Transfer **22**(6), 689–708 (Dec 2020). https://doi.org/10.1007/s10009-020-00561-4
15. Fong, B., Spivak, D.I.: An Invitation to Applied Category Theory: Seven Sketches in Compositionality. Cambridge University Press (2019)
16. Giannakopoulou, D., Namjoshi, K.S., Pasareanu, C.S.: Compositional Reasoning. In: Handbook of Model Checking, pp. 345–383. Springer (2018)
17. Murugesan, A., Whalen, M.W., Rayadurgam, S., Heimdahl, M.P.: Compositional verification of a medical device system. In: ACM SIGAda Annual Conference on High Integrity Language Technology. pp. 51–64 (2013). https://doi.org/10.1145/2527269.2527272
18. Neele, T., Sammartino, M.: Compositional Automata Learning of Synchronous Systems. In: FASE. pp. 47–66. *LNCS* 13991, Springer (2023). https://doi.org/10.1007/978-3-031-30826-0˙3
19. Ouvrard, X.: Hypergraphs: an introduction and review. CoRR **arXiv:2002.05014** (2020)
20. Păsăreanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: Applying the L* algorithm to automate assume-guarantee reasoning. Formal Methods in System Design **32**(3), 175–205 (2008). https://doi.org/10.1007/s10703-008-0049-6
21. Preoteasa, V., Dragomir, I., Tripakis, S.: The refinement calculus of reactive systems. Information and Computation **285**, 104819 (May 2022). https://doi.org/10.1016/j.ic.2021.104819
22. Tripakis, S., Lublinerman, R.: Modular Code Generation from Synchronous Block Diagrams : Interfaces, Abstraction, Compositionality. In: Principles of Modeling, pp. 449–477. *LNCS* 10769, Springer (2018)