Dynamic Reactive Modules *

J. Fisher[1], T.A. Henzinger[2], D. Nickovic[2], N. Piterman[3], A.V. Singh[2], and M.Y. Vardi[4]

[1] Microsoft Research, Cambridge, UK
[2] IST Austria, Klosterneuburg, Austria
[3] University of Leicester, UK
[4] Rice University, Houston, TX, USA

Abstract. State-transition systems communicating by shared variables have been the underlying model of choice for applications of model checking. Such formalisms, however, have difficulty with modeling process creation or death and communication reconfigurability. Here, we introduce "dynamic reactive modules" (DRM), a state-transition modeling formalism that supports dynamic reconfiguration and creation/death of processes. The resulting formalism supports two types of variables, data variables and reference variables. Reference variables enable changing the connectivity between processes and referring to instances of processes. We show how this new formalism supports parallel composition and refinement through trace containment. DRM provide a natural language for modeling (and ultimately reasoning about) biological systems and multiple threads communicating through shared variables.

1 Introduction

State-transition systems provide a natural formalism in many areas of computer science. They provide a convenient framework for understanding programming languages (cf. [21]), provide a natural executable modeling framework for reactive and concurrent systems (cf., [11]), provide the most intuitive semantics for the application of model checking (cf. [4]), and even proved to be useful to the development of biological models [7, 10, 8, 9], where the straightforward semantics make these formalisms natural and attractive for cell biologists. State-transition systems capture elegantly the concept of a system with variables that change their values over time. The state-transition approach to modeling concurrent systems can be fairly described as enormously successful, combining executability, explorability, and analyzability. In the state-transition approach communication is typically modeled via shared variables, while in the complementary approach of process calculi communication is modeled via message passing [18].

In recent years, new application domains that stress mobility and dynamic reconfigurability gained importance. In mobile and ad-hoc networks, network elements come and go, changing communication configuration according to their position. The state-transition approach, however, does not model naturally reconfigurable systems. Similarly, it has difficulty with dynamics features of biological systems, such as cell movement, division, and death.

* This work was supported by the ERC Advanced Grant QUAREM, the FWF NFN Grant S11402-N23 (RiSE), and the EU NOE Grant ArtistDesign

In the process-calculi approach, the π-*calculus* has become the de facto standard in modeling mobility and reconfigurability for applications with message-based communication [19, 20]. The power of the π-calculus comes from its ability to transmit processes as messages, a mathematically natural and powerful construct. This idea immediately allows the encoding of dynamic aspects and has been widely accepted by the research community (cf. [17]). No analogous widely acceptable extension exists for the state-transition approach, enabling the modeling of mobility and reconfigurability.

In this paper, we propose a state-transition formalism that supports reconfiguration of communication and dynamic creation of new processes. We accomplish this by adapting to the state-transition approach three fundamental language mechanisms of modern programming languages: *encapsulation*, *composition*, and *reference*. Encapsulation is a language mechanism for bundling together related data and methods, while restricting access to some of those. Composition is a language mechanism for composing such bundles of data and methods. Finally, reference is a language mechanism for creating such bundles dynamically. While the mechanisms of encapsulation and composition have been used in state-transition formalisms, for example, in *reactive modules* [2], which are the basis for our work here, it is striking that the concept of a reference is missing in all state-based modeling formalisms, while it is present in every reasonable imperative programming language. We show how this well known and widely used concept in programming offers a powerful modeling concept in the state-transition context.

In the reactive-modules formalism, modules define behavior of a bundled set of variables. Behavior of a module is defined through that of its variables, partitioned to internal, interface, and external variables. The module controls its internal and interface variables and reads the external variables from other modules. To allow executability, an update round is partitioned to subrounds. Variables that are co-updated in the same round are not allowed to depend on one another. Thus, the module mechanism essentially supports encapsulation. Then, composition is supported by the ability to compose modules in parallel, and the ability to make multiple copies of modules.

Modern imperative object-oriented programming languages combine our guiding principles: encapsulation, composition, and reference. A *class* is a schema of encapsulated behavior. It has a well defined interface that cleanly supports composition. An *object* has to be *instantiated*, returning a reference through which it can be accessed. It then executes according to its prescribed behavior. Different instantiations of the same class behave differently according to their individual histories, which are stored in their own variables. References, in addition to enabling us to create multiple instances of the same class, allow us to dynamically change the configuration of instances in memory. Classes and references together allow us to organize the program in multiple levels of abstraction and manage (to some extent) the complexity of software.

Here, we adapt these concepts to the world of state-transition modeling. In this context, the instantiation of an object also assigns "dynamic computation power" to it: every newly instantiated variable includes with it a recipe for behavior as a function of the values of some other variables. Our "objects" are independent processes, each controlling a set of variables. We impose encapsulation by assigning ownership to variables. Each process has its own variables, which it and it alone can change; the update may depend on the values of variables that it does not own. Thus, our variables are

single-write multiple-read variables. These variables can be accessed either, traditionally, by direct static sharing, or, via references, by dynamic sharing, enabling dynamic communication configurations. In addition, we use a specialized creation command to model the appearance of new processes that come to the awareness of the system at some point of computation. This is analogous to allocating new memory from the heap. Here, again, references are invaluable, as they allow communication in both directions: for a newly instantiated process, this enables initial knowledge about its environment; for an instantiating process this enables access to some of the newly created variables.

There have been few attempts to handle dynamicity in state-transition formalisms. Dynamic I/O automata [3] are an extension of *I/O automata* [16]. In order to change communication configuration, explicit state-based modeling of the reconfiguration is needed (through changing alphabet signatures from state to state). Alur and Grosu extend reactive modules by creation through the usage of unbounded arrays [1]. Global information regarding arrays and their length is required, as indeed exhibited in the "reconfiguration controller" that controls the entire system. Updates are done via λ-expressions on entire arrays and not locally. This makes it impossible to apply multiple levels of abstraction, one of the main strengths of programming languages. This is akin to viewing the heap as a linear sequence of memory locations and using integers as pointers into the array. This gives a low level implementation of the heap, depriving the programmer of the ability to abstract. Lucid-Synchrone is an extension of *Lustre* that supports creation but restricts to a fixed topology [5]. There have been attempts to add object-orientation to statecharts, an important state-transition formalism. For example, in [13, 12], the semantics of Rhapsody, an object-orientation extension of StateMate, is described in terms of the underlying programming languages. Thus, they bypass the need to reason about dynamic creation of processes. Damm et al. [6] give a specialized semantics for UML Statecharts. Their formalism is cumbered by the need to support directly many specialized features of Statecharts and does not offer a general solution.

Our main contribution is a new state-transition formalism, based on widely used object-oriented programming paradigms, that supports communication via shared variables and dynamic reconfiguration and creation. We show that our formalism supports a straightforward trace semantics, where refinement corresponds to trace containment over appropriate projections and composition corresponds to a specialized form of intersection. In addition, we allow partial specifications that translate to nondeterminism, just like in standard state-transition settings, and their refinement, through a replacement operator. We provide a rich modeling formalism that suggests many future directions.

2 High-Level Description of Dynamic Reactive Modules

We generalize *reactive modules* [2] to *dynamic reactive modules* by including reference variables and the ability to create new modules. This is similar to modern object-oriented languages, where a reference variable refers to an instance of a class. A class definition describes the way to update multiple included variables and an instantiation leads to allocation of memory. While in standard objects the data is updated only via explicit method invocation; in dynamic reactive modules variables continually update their values according to update rules. Thus, instantiating a module leads to allocation of new variables that are updated simultaneously in all instantiated modules. In this sec-

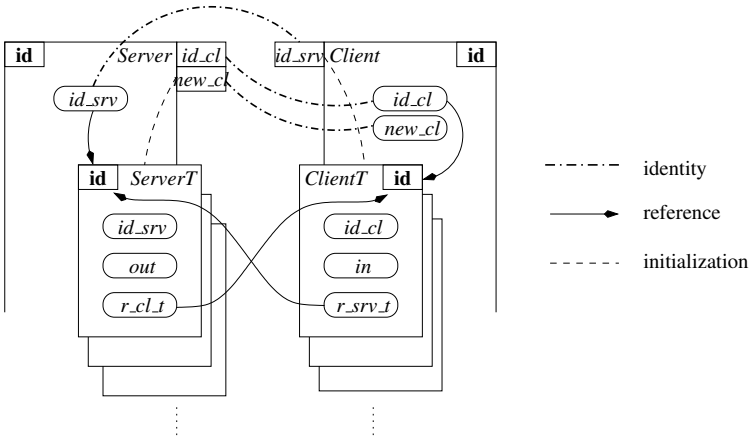

Fig. 1. A server/client system.

tion, we further motivate the need for- and introduce dynamic reactive modules through an example of a simple server/client model. Definitions are made formal in Section 4.

Consider the diagram in Figure 1. It includes a *server* and a *client*. The client generates new *client thread*s at arbitrary times. The server detects that a new client thread has been generated and allocates a new *server thread* dedicated to serve the respective client thread's request. The server and the client need to produce a pair of threads and connect them so that the server thread reads the client thread's input *in* and the client thread reads the server thread's output *out*. For that, the server thread initializes its reference variable *r_cl_t* to refer to *in* and the client thread initializes its reference variable *r_srv_t* to refer to *out*. Once a pair of server thread and client thread have been connected the server and the client can forget about them and create (and mutually initialize) a new pair. Every newly instantiated module gets a unique identifier and its own reference to itself, through the special **id** variable (akin to the *this* construct in Java). The mutual references between server thread and client thread variables are exchanged between the server and the client through reading each other's variables (static communication) and passed to the corresponding thread as parameter. This exchange of references is done by mutually accessing external variables *id_cl* and *id_srv* that hold references to newly created client thread and server thread, respectively. In addition, the server accesses client's variable *new_cl* that signals when a new client thread is created. We generalize the notion of reactive module to that of a *dynamic module*. We distinguish between (a) a dynamic module *class*, which defines the module, its variables, and how to update them and (b) a dynamic module, which is the actual instantiation. A *dynamic system* defines a collection of dynamic-module classes.

In Figure 2, we include the code for the *ServerClient* dynamic reactive system that models the above example. It consists of four modules depicted in Figure 2, together with the (initial) module *Server* || *Client* that denotes the composition of *Server* and *Client* modules. Every module in the system consists of a declaration, that defines the variables owned by the module, and a body that specifies initialization and update rules for these variables. The module body has a finite set of typed variables that are partitioned into *controlled* and *external* variables and either range over finite domains or

```
system ServerClient =
        ⟨{Server, ServerT, Client, ClientT, Server || Client}, Server || Client⟩

class Server
external id_cl : R, new_cl : B
control id_srv : R

atom id_srv
  init
    [] true → id_srv' := 0;
  update
    [] new_cl → id_srv' :=
          new ServerT(id_cl') ;

class ServerT
param id_cl : R
control out : B, r_cl_t : R

atom r_cl_t
  init
    [] true → r_cl_t' := id_cl';
  update
    [] true →
atom out
  initupdate
    [] r_cl_t' ≠ 0 → out' :=
                f(r_cl_t'.in) ;

class Client
external id_srv : R
control id_cl : R, new_cl : B

atom new_cl
  initupdate
    [] true → new_cl' := true;
    [] true → new_cl' := false;

atom id_cl
  init
    [] true → id_cl' := 0;
  update
    [] new_cl → id_cl' :=
          new Client(id_srv') ;
class ClientT
param id_srv : R
control in : B, r_srv_t : R

atom r_srv_t
  init
    [] true → r_srv_t' := id_srv';
  update
    [] true →
```

Fig. 2. Server/client system modeled with dynamic reactive modules.

are reference variables. Additionally, a module has a set of *parameters* and a special variable **id**, which holds the identifier of an instance of the module. Parameter variables are used for initialization of the module according to some information from its environment.

Reference variables establish dynamic communication between module instances. When a module is instantiated, its variable **id** is assigned a unique identifier. For example, $\mathbf{id}.m$ and $\mathbf{id}.n$ use the variable **id** to indirectly access many variables of the same module. We add the two basic functionalities of references. First, the ability to take the address of a variable through $\mathsf{ref}(x)$, which returns a reference to x. Second, the ability to dereference a variable and access the value of the variable that it references.

The variable *id_srv* (*id_cl*) holds a reference to a server (client) thread, it is controlled by *Server* (*Client*) and is external to *Client* (*Server*). In addition, *Client* controls *new_cl* (external to *Server*) that signals the instantiation of a new client in the system. *Client* and *Server* communicate statically over these three variables, and mutually exchange references between newly created client and server threads. The communication between the server thread and the client thread has to be dynamic (via reference variables) as the two are instantiated independently. For that, server (client) thread holds reference *r_cl_t* (*r_srv_t*) to the client (server) thread's identifier. The server (client) thread's identifier is passed to the client (server) thread through the parameter *id_srv* (*id_cl*) upon its

instantiation. We use indirect access to *in* through the reference to the client thread to update the value of *out* of the server thread, through the expression $f(r_cl_t'.in)$.

The module body consists of a set of *atoms* that group rules for setting values to variables owned by the module. Atoms of the module control precisely its controlled variables, and every controlled variable declared in the module is controlled by exactly one atom. We distinguish between the *current* value of a variable, denoted x, and its *next* value x'. Atoms contain initialization and update rules, or commands, that define the value of x' based on current and next values of variables declared in the module. When a module is instantiated, its variables do not have current values. Thus, initial commands may use only next values of variables in the same module, or the values of parameters passed to it. Update commands may refer to both current and next values of variables and can either define an instantiation of a new instance of a module, or a classic update of a variable as a function of current and next values of other variables.

The atom that controls *new_cl* in the *Client* sets the next value new_cl' to either *true* or *false*, nondeterministically. An instantiation is a special type of update, using the command new . It can update the reference variables of the instantiating module to refer to the newly instantiated module and uses parameters to pass information to the instantiated module for proper initialization. For example, the update in the atom that controls *id_cl* either takes no action (if *new_cl* is *false*) or instantiates a new *ClientT*. The instantiation updates the reference variable id_cl' to refer to the newly instantiated *ClientT*. When the new client thread is created it receives the value of the identifier of the *ServerT* instance in variable id_srv' through the parameter *id_cl*. Passing the identifier to an instance of a module enables access to all variables of that instance. For example, in the initial command in the atom *r_srv_t*, the value of parameter *id_srv* is stored in *r_srv_t*. Thus, if *id_srv* is null (**0**) then *r_srv_t* is updated to null as well. Otherwise, *id_srv* holds the reference of a server thread and *r_srv_t* is updated to the refer to the same server thread. Overall, the co-instantiation of a *ClientT* and *ServerT* modules will initialize the value of r_srv_t' of the client thread to refer to the server thread and the value of r_cl_t' of the server thread to refer to the client thread. To avoid infinite instantaneous creation, we disallow instantiation of new modules in initial commands.

A state of a dynamic reactive system carries the unique identifiers and variable valuations of instantiates modules. In the initial state, the only instantiated module is the initial one. In every subsequent round, the state variables are updated according to the specified commands, which may, in addition, instantiate new modules. Initialization of instantiated modules depends on transferred parameter values.

3 Dynamic Discrete Systems

Dynamic reactive modules is a modelling language. In this section, we introduce a semantic model, which is interesting in its own right, to give a formal semantics to dynamic reactive modules. We extend fair discrete systems (FDS) [14], which are "bare bones" transition systems including a set of variables and prescribed initial states and transition relations by logical formulas. The simplicity of FDS and their resemblance to BDDs, have made them a convenient tool for defining symbolic transition systems. FDSs support composition but not encapsulation and here we extend them with dynamicity. We then use this new model to define the semantics of dynamic reactive modules.

Our template for creating a process is a *simple dynamic discrete system* (SDDS) and the collection of SDDSs is a *dynamic discrete system* (DDS). An SDDS defines a process, its variables, their initializations, and their updates. To create multiple instances of an SDDS, each instantiation has a unique *identifier*. Accordingly, when instantiating an SDDS we allocate all its variables with the same identifier. As mentioned, DDS do not support encapsulation. Thus, the model has a set of variables coming from multiple SDDS and possibly multiple instantiations of the same SDDS. We prefix the variables of the SDDS with the identifier of its instantiation, thus making the variables unique. For that we will use *identified* variables. For example, if the definition includes the variable n, the instantiation with identifier i uses the variable $i.n$.

Let $\mathcal{N}$ be the universal set of variables such that $\mathbf{id} \in \mathcal{N}$. The variables in $\mathcal{N}$ are going to be used in the definitions of SDDSs. Let $\mathcal{I}$ be the universal set of identifiers. The identifiers in $\mathcal{I}$ are going to be used to identify instances of SDDSs. Apart from the universal set of variables, all sets of variables $N \subset \mathcal{N}$ are *finite*. For example, in Figure 2, the set $\{\mathit{id_cl}, \mathit{new_cl}, \mathit{id_srv}\}$ is the set of variables for the server. When an SDDS is instantiated, all its variables are going to be prefixed with an identifier i. For that, given a set of variables X, let $i.X$ denote $\{i.n \mid n \in X\}$. When a server thread, from Figure 2, is instantiated with identifier i, the set of identified variables for that instance is $\{i.\mathit{out}, i.\mathit{r_cl_t}\}$. So when there are multiple active instances of server thread, e.g., with identifiers i and j, their variables can be distinguished, e.g., as $i.\mathit{out}$ and $j.\mathit{out}$. Variables range either over some finite domain (for the sake of concreteness we use *Booleans* denoted $\mathbb{B}$) or over the set $\mathcal{R} = \mathcal{I} \cup \{\mathbf{0}\}$ of *references*. A reference is either the identifier of an instantiated SDDS ($i \in \mathcal{I}$) or null ($\mathbf{0}$). We denote by $\mathsf{type}(x)$ the *type* of a variable x. For a variable $x \in X$, we denote by x' its *primed* copy, and naturally extend this notation for a set X.

Let $\mathfrak{X} = \mathcal{I} \times \mathcal{N}$ be the universal set of *identified variables*. A *state* s is a valuation function $s : \mathfrak{X} \to \mathcal{R} \cup \mathbb{B} \cup \{\bot\}$ such that for every $i \in \mathcal{I}$ we have $s(i.\mathbf{id}) \in \{\bot, i\}$. That is, a state interprets *all* variables as either Booleans, identifiers, identified variables, or $\bot$. The $\mathbf{id}$ of i is either i or $\bot$. The value $\bot$ is used for two purposes. First, if $s(i.n) = \bot$, then $i.n$ is not allocated in s. Second, $\bot$ is used as a third value in 3-valued propositional logic. This allows to formally represent impossible dereferencing. The type of a variable x in state s is denoted as $\mathsf{type}_s(x)$. A variable x such that $s(x) \in \mathbb{B}$ is said to be Boolean, denoted $\mathsf{type}_s(x) = \mathbb{B}$. A variable x such that $s(x) \in \mathcal{R}$ is said to be a reference, denoted $\mathsf{type}_s(x) = \mathbb{R}$. Let $s_\bot$ denote the state such that $s(x) = \bot$ for every $x \in \mathfrak{X}$. An identified variable $x \in \mathfrak{X}$ is *inactive* in state s if $s(x) = \bot$ and *active* otherwise. If $i.\mathbf{id}$ is inactive in state s then for every variable n we have $i.n$ is inactive in s. An identifier i is *inactive* in state s if $i.\mathbf{id}$ is inactive in s and *active* otherwise.

Through a reference variable that holds an identifier of an SDDS, we need to be able to access the variables of this SDDS. Given a set of variables $X \subset \mathcal{N}$, we define *direct accesses* (π), *indirect accesses* (τ) and *expressions* (φ) over X as follows.

$$\begin{aligned} \pi &::= x \in X \cup X \\ \tau &::= \pi \mid \tau.m \text{ for } m \in \mathcal{N} \\ \varphi &::= \tau \mid \tau = \tau \mid \tau = \mathbf{0} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \end{aligned} \tag{1}$$

We give values to both x and x' by interpreting direct accesses, indirect accesses, and expressions over *pairs* of states, which stand for current and next values.

$$(s,t)(x)=\begin{cases} s(x) \text{ if } x \in \mathfrak{X} \\ t(x) \text{ if } x \in \mathfrak{X}' \end{cases}$$

$$(s,t)(x.m)=\begin{cases} \bot & \text{if } (s,t)(x) \notin \mathcal{I} \\ (s,t)((s,t)(x).m) & \text{if } (s,t)(x) \in \mathcal{I} \text{ and } x \in \mathfrak{X} \\ (s,t)((s,t)(x).m') & \text{if } (s,t)(x) \in \mathcal{I} \text{ and } x \in \mathfrak{X}' \end{cases}$$

Fig. 3. Evaluation of indirect accesses on a state pair (s,t).

In Figure 2, the indirect access *r_cl_t'.in* in *ServerT*'s atom *out*, updates the value of *out* based on the value of the variable *in* of the associated *ClientT*.

An expression that does not use primed variables is *current*. An expression that does not use unprimed variables is *next*. Thus, expressions are logical characterization of possible assignments to variables. As usual, using two copies of a variable x and x' we can use expressions to define the relations between current and next assignments. We assume familiarity Kleene's strongest regular 3-valued propositional logic over the set $\mathbf{3} = \{\mathtt{t}, \bot, \mathtt{f}\}$ [15]. For example, $\mathtt{f} \wedge \bot = \mathtt{f}$, $\bot \vee \mathit{true} = \mathit{true}$, and $\neg\bot = \bot$.

Given two states s and t, we denote by (s,t) the mapping $(s,t) : \mathfrak{X} \cup \mathfrak{X}' \to \mathcal{R} \cup \mathbb{B} \cup \{\bot\}$ such that for every $x \in \mathfrak{X}$ we have $(s,t)(x) = s(x)$ and $(s,t)(x') = t(x)$. The definition of $\mathsf{type}_{(s,t)}(x)$ is extended as expected.

The value of an indirect access τ in pair (s,t) is defined in Figure 3. For example, consider the value of the indirect access $x.m$. We start by evaluating $(s,t)(x)$. If $(s,t)(x)$ is not an identifier, then clearly we cannot access its m variable and return $\bot$. Otherwise, $(s,t)(x)$ is an identifier i. If x is unprimed, then we access the value of $i.m$ in s. Otherwise, we access the value of $i.m$ in t. Other evaluations of the indirect access are similar. The value of an expression φ in pair (s,t) denoted $(s,t)(\varphi)$, is defined as follows. For an indirect access τ we have already defined $(s,t)(\tau)$. We define $(s,t)(\tau_1 = \tau_2)$ to be $\mathtt{t}$ if $(s,t)(\tau_1) = (s,t)(\tau_2)$ and $\mathtt{f}$ otherwise. Similarly, $(s,t)(\tau = \mathbf{0})$ is $\mathtt{t}$ iff $(s,t)(\tau) = \mathbf{0}$. The definition of $(s,t)(\varphi)$ for expressions using the Boolean connectives $\wedge$, $\vee$, and $\neg$ is as expected, where every $\alpha \in \mathcal{R}$ is treated like $\bot$. Finally, a pair (s,t) satisfies an expression φ if $(s,t)(\varphi) = \mathtt{t}$. Note that the definitions in Figure 3 takes into account both the current and next versions of variables. Thus, it is defined over a pair of states (s,t). The definition for current expressions and single state is a specialization, where we care only about the state s on the left.

3.1 Definitions

A DDS is $\mathcal{K} = \langle \mathbb{D}, \mathcal{D}_0 \rangle$, where $\mathbb{D}$ is a finite set of SDDS and $\mathcal{D}_0 \in \mathbb{D}$ is an initial SDDS. An SDDS is a tuple $\mathcal{D} = \langle X, Y, \Theta, \rho \rangle$ consisting of the following components.

- $X \subseteq \mathcal{X}$ is the finite set of variables of $\mathcal{D}$ and Y is the finite set of its parameters.
- Θ: The initial condition is a next expression over $X \cup Y$ characterizing all states in which $\mathcal{D}$ can be created. These are the initial states of $\mathcal{D}$ at the time of creation.
- ρ: The transition relation. We extend expressions in Equation (1) to *creation expressions*. Given $\mathcal{D}_i = \langle X_i, Y_i, \Theta_i, \rho_i \rangle$, for $i \in \{1,2\}$, a *creation* of $\mathcal{D}_2$ by $\mathcal{D}_1$ is $n_1' = \mathsf{new}\ \mathcal{D}_2(\tau_1, \ldots, \tau_l)$ where $n_1 \in X_1$, $\{y_1, \ldots, y_l\} = Y_2$, and $\tau_1, \ldots, \tau_l$ are terms over X_1. Intuitively, the new expression returns the identifier i of the newly created module. Thus, the new command stores the identifier of the newly created SDDS in n_1'. The parameters of $\mathcal{D}_2$ are initialized with the values of the expressions

τ_j passed by $\mathcal{D}_1$. Let $C(\mathbb{D}, \mathcal{D})$ be the set of all possible creations of SDDS $\mathcal{D}'$ by $\mathcal{D}$ such that $\mathcal{D}' \in \mathbb{D}$. Let φ denote the set of expressions over X, then creation expressions by $\mathcal{D}$ in the context of $\mathbb{D}$ are:

$$\varphi_c \ ::= \ \varphi \mid c \in C(\mathbb{D}, \mathcal{D}) \mid \varphi_c \wedge \varphi_c \mid \varphi_c \vee \varphi_c,$$

The transition relation ρ is a creation expression by $\mathcal{D}$ in the context of $\mathbb{D}$.

We now define the possible traces of an SDDS. This is a sequence of states such that every pair of adjacent states satisfy the transition of the SDDS. However, as creation is involved, the transition relation needs to be augmented with the rules that govern the newly created variables. For that, we add to traces the maps of creations that are performed along them and the update of the transition that governs these new creations.

Given a transition relation ρ, let $\mathsf{subnew}(\rho)$ be the subformulas of ρ that are creations. A *creation-map* m for ρ is a partial one-to-one function $m : \mathsf{subnew}(\rho) \to \mathcal{I}$. A creation map tells us which creations are actually invoked (those for which m is defined) and what is the identifier of the instantiated process.

A pair of states (s, t) satisfies a transition ρ with creation map m and producing transition $\tilde{\rho}$, denoted $(s, t) \models (\rho, m, \tilde{\rho})$ if all the following conditions hold.

1. For every creation $c \in \mathsf{subnew}(\rho)$ of $\mathcal{D}_1$, if $m(c) = i$ then i is *inactive* in s and for every $n \in X_1$ we have $i.n$ is *active* in t. That is, instantiated SDDS are activated.
2. For every $i.n$ such that i is *active* in s we have $i.n$ is *active* in s iff it is *active* in t and $\mathsf{type}_s(i.n) = \mathsf{type}_t(i.n)$. That is, existing instantiations do not change.
3. For every creation $c \in \mathsf{subnew}(\rho)$ of $\mathcal{D}_1$, where c is
$$i.n_1 = \mathsf{new}\ \mathcal{D}_1(\tau_1, \ldots, \tau_l),$$
if $m(c) = j$ then all the following hold:
   (a) $(s, t) \models \Theta_1[j][\tau_1/y_1, \ldots, \tau_l/y_l]$, where $\Theta_1[j][\tau_1/y_1, \ldots, \tau_l/y_l]$ is obtained from Θ_1 by replacing every mention of n by $j.n$ and every input y'_b by τ_b.
   (b) $(s, t)(i.n'_1) = j$,
   That is, the pair (s, t) satisfies the initialization of the instantiated SDDS using the inputs sent by the creating SDDS. Furthermore, the reference n_1 of the creating SDDS now references the newly created SDDS.
4. $(s, t) \models \overline{\rho}$, where $\overline{\rho}$ is obtained from ρ by replacing the creation sub-formulas $c \in \mathsf{subnew}(\rho)$ such that $m(c) = i$ by `t`, and $c \in \mathsf{subnew}(\rho)$ such that $m(c)$ is undefined by `f`.
   That is, the pair (s, t) satisfies the transition relation. We ensure that enough SDDS were instantiated by evaluating those that were not instantiated as `f`.
5. $\tilde{\rho} = \rho \wedge \bigwedge_{\{c \mid m(c)=i\}} \rho_c[m(c)]$, where $\rho_c[m(c)]$ is the transition relation of the SDDS created by c with every mention of n replaced by $m(c).n$.
   That is, we update the transition relation with the rules that govern the updates of the newly created SDDSs.

We are now ready to define traces of DDS. Traces are going to include the states, transition relations, and creation maps that match them. Consider a finite or infinite sequence $\sigma = s_0, \rho_0, m_0, s_1, \rho_1, m_1 \ldots$, where for every $j \geq 0$ we have s_j is a state, ρ_j is a creation expression, and m_j is a creation map for ρ_j. If σ is infinite we write $|\sigma| = \omega$. If σ is finite it ends in an expression ρ_{n-1} and we write $|\sigma| = n$. A sequence σ is a

creation trace for an SDDS $\mathcal{D} = \langle X, Y, \Theta, \rho \rangle$ with identifier i at time $0 \leq t < |\sigma|$ and valuations $v_1, \ldots, v_l$ for $\{y_1, \ldots, y_l\} = Y$ if all the following hold.

1. For every $t' < t$ we have $\rho_{t'} = \mathtt{t}$ and $m_{t'}$ is the empty map. Furthermore, $\rho_t = \rho[i]$.
2. If $s_{-1} = s_\perp$ then $(s_{t-1}, s_t) \models \Theta[i][v_1/y_1, \ldots, v_k/y_k]$.
3. For every $0 \leq t' < |\sigma| - 1$ we have $(s_{t'}, s_{t'+1}) \models (\rho_{t'}, m_{t'}, \rho_{t'+1})$.
4. The identifier i is *inactive* in s_{t-1} and for every $n \in X$, $i.n$ is *active* in s_t.

We write in short $(\sigma, i, t, v_1, \ldots, v_k)$ is a CT of $\mathcal{D}$.

Intuitively, the SDDS $\mathcal{D}$ is created at time t by initializing its inputs to v_1, …, v_l. All the variables of $\mathcal{D}$ (and possibly more) identified by i become active in t; And the (mutable according to the creation maps) transition of $\mathcal{D}$ holds on the entire sequence. Prior to the creation of $\mathcal{D}$ the transitions are $\mathtt{t}$ and accordingly creation maps are empty.

A finite CT σ ends in a *deadlock* if it cannot be extended to a longer CT. Intuitively, there can be two reasons for deadlocks. First, a contradiction in the transition, such as requiring that $x = y$ and $y = \neg x$. Obviously, this can be made more interesting by accessing x and y through their references. Second, the option to indirectly access through null, a Boolean variable, or trying to access a wrong name.

3.2 Properties

Here we define parallel composition, refinement, and replacement. Parallel composition allows to create models of increasing complexity from smaller parts. It enables static communication through external variables. Refinement says when one DDS is more general than another. Then, replacement is the action of replacing creation of abstract SDDS by SDDS that refine it. Composition corresponds to intersection of traces and refinement to inclusion of traces (both with appropriate adjustments).

We start with parallel composition, which essentially allows to "run" two SDDS side by side. Consider a set of SDDS $\mathbb{D}$ and two SDDS $\mathcal{D}_i \in \mathbb{D}$, where $\mathcal{D}_i = \langle X_i, Y_i, \Theta_i, \rho_i \rangle$ for $i \in \{1, 2\}$. Then, $\mathcal{D}_{1\|2}$ is the SDDS $\langle X_{1\|2}, Y_{1\|2}, \Theta_{1\|2}, \rho_{1\|2} \rangle$ where $X_{1\|2} = X_1 \cup X_2$, $Y_{1\|2} = Y_1 \cup Y_2$, $\Theta_{1\|2} = \Theta_1 \wedge \Theta_2$, and $\rho_{1\|2} = \rho_1 \wedge \rho_2$.

Consider a CT $\mu = (\sigma, i, t, v_1, \ldots, v_k)$, where $\sigma = s_0, \rho_0, m_1, \ldots$. We say that CTs $(\sigma^1, i, t, v_{j_1}, \ldots, v_{j_{l_1}})$ and $(\sigma^2, i, t, v_{p_1}, \ldots, v_{p_{l_2}})$ *partition* μ if $\{v_{j_1}, \ldots, v_{j_{l_1}}\} \cup \{v_{p_1}, \ldots, v_{p_{l_2}}\} = \{v_1, \ldots, v_k\}$ and for every $t \geq 0$ we have $s_t = s_t^1 = s_t^2$, $\rho_t = \rho_t^1 \wedge \rho_t^2$, and m_t is the disjoint union of m_t^1 and m_t^2.

Theorem 1. *A creation trace μ is a creation trace of $\mathcal{D}_{1\|2}$ iff there exist μ_1 and μ_2 that partition μ such that μ_i is a creation trace of $\mathcal{D}_i$, for $i \in \{1, 2\}$.*

We define refinement as having the same set of traces with specialized creations. Consider a set of SDDS $\mathbb{D}$ and two SDDS $\mathcal{D}_i \in \mathbb{D}$, where $\mathcal{D}_i = \langle X_i, Y_i, \Theta_i, \rho_i \rangle$ for $i \in \{1, 2\}$. Consider two CTs $\mu_1 = (\sigma_1, i, t, v_{j_1}, \ldots, v_{j_{l_1}})$ and $\mu_2 = (\sigma_2, i, t, v_1, \ldots, v_{l_2})$, where $\sigma_i = s_0^i, \rho_0^i, m_0^i, \ldots$, for $i \in \{1, 2\}$. We say that μ_2 *specializes* μ_1 if for every $t' \geq 0$ we have $\rho_{t'}^2 = \rho_{t'}^1 \wedge \rho_{t'}^*$ and $m_{t'}^2$ is the disjoint union of $m_{t'}^1$ and $m_{t'}^*$ for some $\rho_{t'}^*$ and $m_{t'}^*$. We say that $\mathcal{D}_2$ *refines* $\mathcal{D}_1$, denoted $\mathcal{D}_2 \preceq \mathcal{D}_1$, if $X_2 \supseteq X_1$, $Y_2 \supseteq Y_1$, and every creation trace μ_2 of $\mathcal{D}_2$ is a specialization of some CT μ_1 of $\mathcal{D}_1$.

Theorem 2. *The refinement relation $\preceq$ is a preorder.*

In order to replace the creation of $\mathcal{D}_2$ by $\mathcal{D}_3$ we have to ensure that $\mathcal{D}_3$ does "more" than $\mathcal{D}_2$. Consider a set of SDDS $\mathbb{D}$. We say that transition relation ρ_2 *refines* transition relation ρ_1 if $\rho_2 = \overline{\rho}_1 \wedge \rho^*$ for some ρ^*, where $\overline{\rho}_1$ is obtained from ρ_1 by replacing every creation $n_1 = \mathsf{new}\ \mathcal{D}_3(\ldots, \tau_l)$ in ρ_1 by creation $n_1 = \mathsf{new}\ \mathcal{D}_4(\ldots, \tau_l, \tau_{l+1}, \ldots, \tau_{l+b})$, where $\mathcal{D}_4$ refines $\mathcal{D}_3$. As for every SDDS $\mathcal{D} \preceq \mathcal{D}$, some creations can remain unchanged.

Theorem 3. *For all* SDDS $\mathcal{D}_1$ *and* $\mathcal{D}_2$*, if the initial condition* Θ_2 *refines the initial condition* Θ_1 *and the transition relation* ρ_2 *refines the transition relation* ρ_1*, then* $\mathcal{D}_2 \preceq \mathcal{D}_1$.

Theorem 4. *For all* SDDS $\mathcal{D}_1$*,* $\mathcal{D}_2$*, and* $\mathcal{D}_3$*, we have* $\mathcal{D}_{1\|2} \preceq \mathcal{D}_1$ *and* $\mathcal{D}_{(1\|2)\|3} = \mathcal{D}_{1\|(2\|3)}$.

Finally, when an SDDS refines another, we can replace creations of the second by creations of the first. Consider SDDS $\mathcal{D}_i = \langle X_i, Y_i, \Theta_i, \rho_i \rangle$, for $i \in \{1, 2, 3\}$, where $\mathcal{D}_3 \preceq \mathcal{D}_2$. The SDDS $\mathcal{D}_{1[3/2]}$ is given by $\langle X_1, Y_1, \Theta_1, \overline{\rho}_1 \rangle$ where $\overline{\rho}_1$ is obtained from ρ_1 by replacing every creation $n_1 = \mathsf{new}\ \mathcal{D}_2(\ldots, \tau_l)$ by a creation $n_1 = \mathsf{new}\ \mathcal{D}_3(\ldots, \tau_l, \tau_{l+1}, \ldots, \tau_{l+b})$.

Theorem 5. *For all* SDDS $\mathcal{D}_1$*,* $\mathcal{D}_2$ *and* $\mathcal{D}_3$ *where* $\mathcal{D}_3 \preceq \mathcal{D}_2$*, we have* $\mathcal{D}_{1[3/2]} \preceq \mathcal{D}_1$.

4 Formal Dynamic Reactive Modules

We give the formal definition of dynamic reactive modules. As mentioned, a module class is the recipe of behavior that may be instantiated multiple times. A dynamic reactive system is a collection of reactive-module classes, where one is identified as initial.

A *dynamic reactive system* $M = (\mathcal{S}, S_0)$ consists of a finite set of *module classes* $\mathcal{S}$ and an *initial* class $S_0 \in \mathcal{S}$. A class $S = (X, Y, \mathcal{A})$ consists of a finite set X of *typed* variables, a finite set Y of *typed parameters* and a finite set $\mathcal{A}$ of *atoms*. The set X is partitioned into two sets: (1) a set *ctr* of *controlled* variables and (3) a set *ext* of *external* variables. The set of atoms $\mathcal{A}$ partitions further the controlled variables, where each atom $A \in \mathcal{A}$ controls the initialization and the updates of a subset $\mathit{ctr}(A) \subseteq \mathit{ctr}$. Note that we allow $\mathcal{A}$ to be *empty*, in which case all the variables in X can have unconstrained behavior. If $\mathcal{A}$ is not empty, every atom $A \in \mathcal{A}$ consists of two finite sets $\mathit{Init}(A)$ and $\mathit{Update}(A)$ of *guarded commands* γ that define rules for initializing and updating variables in $\mathit{ctr}(A)$, respectively. We distinguish between *initial* and *update* guarded commands. A guarded command $\gamma \in A$ is a pair $(p_\gamma, \mathit{Act}_\gamma)$, where p_γ is a *guard*, i.e. a next expression φ over $X \cup Y$ if γ is initial, or an expression over X if γ is update, and Act_γ consists of a finite set of *actions* that can have the following form: (1) $n' := \varphi;$, where $n \in \mathit{ctr}(A)$ and φ is a future expression over $X \cup Y$ if γ is initial, or an expression over X if γ is update, or (2) $n'_1 := \mathsf{new}\ S'(\tau_1, \ldots, \tau_l);$, where $S' \in \mathcal{S}$, for all $i \in [1, k]$, $n_1 \in \mathit{ctr}(A)$, $\mathit{param}(S') = \{y_1, \ldots, y_l\}$, and for all $i \in [1, l]$, we have τ_i is a term over X. A guarded command γ is said to be *creation-free* if Act_γ contains no creation action. We require that for all classes $S \in \mathcal{S}$, all atoms $A \in \mathcal{A}(S)$, the set $\mathit{Init}(A)$ contains only creation-free guarded commands.

Renaming avoids conflicts when statically creating different instances of a class.

Definition 1 (Class Renaming). *For a class S with $X = \{n_1, \ldots, n_k\}$ then $S[m_1 = n_1, \ldots, m_k = n_k]$ is the class that results from S by replacing n_j by m_j for every j.*

The composition operation between two classes results in a single class whose behavior captures the interaction between two classes. Two classes S_1 and S_2 are composable if they do not share controlled variables. Parallel composition encodes the static and hard-coded input/output connections between S_1 and S_2. We naturally extend composition from classes to systems M_1 and M_2.

Definition 2 (Parallel Composition). *Let $S_1 = (X_1, \mathcal{A}_1)$ and $S_2 = (X_2, \mathcal{A}_2)$ be two classes. We say that S_1 and S_2 are* composable *if* $\text{ctr}(S_1) \cap \text{ctr}(S_2) = \emptyset$. *Given two composable classes S_1 and S_2, we denote by $S = S_1 \,||_S\, S_2$ the* parallel composition *of S_1 and S_2, where $S = (X, \mathcal{A})$, such that $X(S)$ is partitioned into* $\text{ctr}(S) = \text{ctr}(S_1) \cup \text{ctr}(S_2)$, $\text{ext}(S) = (\text{ext}(S_1) \cup \text{ext}(S_2)) \setminus \text{ctr}(S)$ *and* $\text{param}(S) = \text{param}(S_1) \cup \text{param}(S_2)$ *and $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$.*

Let $M_1 = (\mathcal{S}_1, S_1^0)$ and $M_2 = (\mathcal{S}_2, S_2^0)$ be two dynamic reactive systems. We say that M_1 and M_2 are composable *if $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$ and S_1^0 and S_2^0 are composable. Given two composable systems M_1 and M_2, we define their* parallel composition*, denoted by $M = M_1 \,||\, M_2$ as the system $M = (\mathcal{S}, S^0)$, such that $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{S^0\}$ and $S^0 = S_1^0 \,||_S\, S_2^0$.*

We define the *extending* operator between classes S_1 and S_2 to capture specialization at the syntactic level. Informally, a class S_1 extends S_2 if S_1 and S_2 have the same updates for the joint controlled variables, but S_1 is allowed to constrain more control variables than S_2 and can read more variables (external and input) from its environment.

Definition 3 (Extending Classes). *Let S_1 and S_2 be two classes. We say that S_1* extends *S_2, denoted by $S_1 \sqsubseteq S_2$ if* $\text{ctr}(S_2) \subseteq \text{ctr}(S_1)$, $\text{ext}(S_2) \subseteq \text{ext}(S_1)$, $Y(S_2) \subseteq Y(S_1)$ *and* $\mathcal{A}(S_2) \subseteq \mathcal{A}(S_1)$.

Definition 4 (Replacement of Classes). *Let S_1 and S_2 be two classes. We say that S_2 is* replaceable *by S_1 if $S_1 \sqsubseteq S_2$.*

Let S_1, S_2 and S_3 be three classes such that $S_3 \sqsubseteq S_2$. We denote by $S_1[S_3/S_2]$ the replacement *of S_2 by S_3 in S_1, that consists in replacing every occurrence of a creation $n_1' :=$ **new** $S_2(\ldots, \tau_l)$; in S_1 by a creation $n_1' :=$ **new** $S_3(\ldots, \tau_l, \tau_{l+1}, \ldots, \tau_{l+q})$;.*

We extend this operator to systems, and given two systems $M_A = (\mathcal{S}_A, S_A^0)$ and $M_B = (\mathcal{S}_B, S_B^0)$ and two classes S_2 and S_3 such that $S_2 \in \mathcal{S}_A$ and $S_3 \sqsubseteq S_2$, we say that M_B replaces *S_2 by S_3 in M_A, denoted by $M_B = M_A[S_3/S_2]$, if the following conditions hold: (1) $\mathcal{S}_B = \mathcal{S}_A \cup \{S_3\}$; (2) if $S_A^0 = S_2$, then $S_B^0 = S_3$, and $S_B^0 = S_A^0$ otherwise; and (3) every S in $\mathcal{S}(M_A)$ is replaced by $S[S_3/S_2]$ in M_B.*

We define the semantics of a reactive dynamic system M in terms of an associated dynamic discrete system $[\![M]\!]$. We now formalize the translation from M to $[\![M]\!]$.

Definition 5 (Semantics: from DRM to DDS). *Let $M = (\mathcal{S}, S^0)$ be a dynamic reactive system. Then, its associated DDS is $[\![M]\!] = \langle \mathbb{D}, \mathcal{D}^0 \rangle$, where $\mathbb{D} = \bigcup_{S \in \mathcal{S}} \mu(S)$ and $\mathcal{D}^0 = \mu(S^0)$ and for a given S, $\mu(S) = \langle X_S, Y_S, \theta_S, \rho_S \rangle$, such that*

- $X_S = X(S(M))$ *and* $Y_S = Y(S(M))$
- θ_S *is the expression* $\bigwedge_{A \in \mathcal{A}(S(M))} \bigvee_{\gamma \in \text{Init}(A)} (p_\gamma \rightarrow (\bigwedge_{\alpha \in \text{Act}_\gamma} e_\alpha))$*, where* e_α *is the expression* $n' = \varphi$ *obtained from the assignment action* $\alpha = (n' := \varphi)$
- ρ_S *is the expression* $\bigwedge_{A \in \mathcal{A}(S(M))} \bigvee_{\gamma \in \text{Update}(A)} (p_\gamma \rightarrow (\bigwedge_{\alpha \in \text{Act}_\gamma} e_\alpha))$*, where* e_α *is either the expression* $n' = e$ *if* $\alpha = (n' = e)$ *or the creation* $n'_1 = \textsf{new}\, \mathcal{D}_i(\tau_1, \ldots, \tau_l)$ *if* α *is the creation action* $n'_1 := \textsf{new}\, S_i(\tau_1, \ldots, \tau_l)$*.*

The following theorem establishes some derived properties from the operations on modules and the properties shown in Section 3.2

Theorem 6. *Given three classes* S_1*,* S_2*, and* S_3*, we have*
1. *if* $S_2 \sqsubseteq S_1$*, then* $\mu(S_2) \preceq \mu(S_1)$*;*
2. *if* S_1 *and* S_2 *are composable classes, then* $\mu(S_1 \,||_S\, S_2) \preceq \mu(S_1)$*;*
3. *if* $S_2 \sqsubseteq S_3$*, then* $\mu(S_1[S_3 \backslash S_2]) \preceq S_1$*.*

Biological Example: As another example, shown in Figure 4, we model a simple system *CellModule* of cells (*Cell* class) arranged in a row. Cells can divide at arbitrary times. This is modeled by creation of a new *Cell* instance by an existing one. A newly created cell always appears to the right of its parent. The parent then updates its right neighbor by updating its variable *right* to refer to its daughter cell. Similarly, the daughter cell updates its left neighbor by updating its variable *left* to refer to the parent cell's **id** (passed in the parameter *pid*). The cell to the right of the parent cell in the current round, updates its left neighbor in the next round by updating variable *left* to refer to the **id** of the new child cell that appeared on its left. The system runs creating new cells non-deterministically, updating the cell-cell communication pattern with each creation.

5 Conclusions and Future Work

We introduced here dynamic reactive modules, a formalism for modeling dynamic state-transition systems communicating via shared variables. Our formalism supports the three basic features of programming languages: composition, encapsulation, and dynamicity. Previous formalisms supported only the first two and by adding references and creation we achieve dynamicity. The resulting formalism supports instantiation of new "active" variables and reconfiguration of communication.

The resulting formalism is quite powerful and it is clear that many questions, such as deadlock freedom, reachability, and model checking, are going to be undecidable. As dynamicity has been generally missing from state-transition formalisms communicating via shared variables, we hope that this formalism will motivate further research into its modeling capacity and the availability of analysis techniques for it. We state a few obvious such directions. We are interested in techniques from software model checking that could be adapted for its analysis. Similarly, pointer analysis, techniques for understanding the structure of the heap, and static analysis in general, could be applied in this context as well. Another interesting direction is the identification of fragments for which such questions are "well behaved". One very important type of well behavedness is deadlock avoidance. We are searching for simple rules for deadlock avoidance through by combining (a) avoiding cyclic dependencies between variables and (b) reference safety through typing and access protection.

$$\textbf{system } \mathit{CellModule} = \langle \{\mathit{Cell}\}, \mathit{Cell} \rangle$$

class *Cell*
control $\mathit{create}, \mathit{left}, \mathit{prev_l}, \mathit{right} : \mathcal{R}$
param $\mathit{pid} : \mathcal{R}$
atom *create*
  init
    $[]\ \mathit{true} \rightarrow \mathit{create}' := \mathbf{0}$
  update
    $[]\ \mathit{true} \rightarrow \mathit{create}' := \mathsf{new}\ \mathit{Cell}(\mathbf{id})$
    $[]\ \mathit{true} \rightarrow$
atom *prev_l*
  init
    $[]\ \mathit{true} \rightarrow \mathit{prev_l}' := \mathbf{0}$
  update
    $[]\ \mathit{true} \rightarrow \mathit{prev_l}' := \mathit{left};$

atom *right*
  init
    $[]\ \mathit{true} \rightarrow \mathit{right}' := \mathit{pid}', \mathit{right}$
  update
    $[]\ \mathit{create} \neq \mathit{create}' \rightarrow \mathit{right}' := \mathit{create}'$
atom *left*
  init
    $[]\ \mathit{true} \rightarrow \mathit{left}' := \mathit{pid}'$
  update
    $[]\ (\mathit{left.create}) \neq (\mathit{left_p}'.\mathit{create}) \rightarrow$
    $\mathit{left}' := \mathit{left_p}'.\mathit{create}$

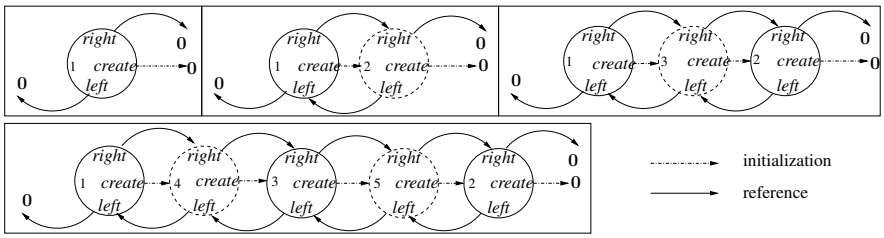

Fig. 4. Dynamic reactive system *CellModule* and dividing cells.

- **Something about assertion language. Include assertions in the modules. Assertions can refer to anything that is viable by the module. To handle problematic accesses it will have to be three valued.**
  The main assertions of the entire system are those that are in the initial module.
- **We can model death by adding a Boolean variable that can only change from 0 to 1. Once this variable is set to 1 all other variables cannot change anymore (requires every part of the transition to include a conjunct). References to this module can check if the module is dead.**

References

1. R. Alur and R. Grosu. Dynamic Reactive Modules. Tech. Rep. 2004/6, Stony Brook, 2004.
2. R. Alur and T.A. Henzinger. Reactive modules. *FMSD*, 15(1):7–48, 1999.
3. P.C. Attie and N.A. Lynch. Dynamic input/output automata, a formal model for dynamic systems. In *PODC*, pages 314–316, 2001.
4. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
5. J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous data-flow language. In *EmSoft*, pages 230–239. ACM, 2004.
6. W. Damm, B. Josko, A. Pnueli, and A. Votintseva. A discrete-time UML semantics for concurrency and communication in safety-critical applications. *SCP*, 55(1-3):81–115, 2005.

7. S. Efroni, D. Harel, and I.R. Cohen. Toward rigorous comprehension of biological complexity: Modeling, execution, and visualization of thymic T-cell maturation. *Genome Res.*, 13(11):2485–97, 2003.
8. S. Efroni, D. Harel, and I.R. Cohen. Emergent dynamics of thymocyte development and lineage determination. *PLoS Comput. Biol.*, 3(1):e13, 2007.
9. J. Fisher, N. Piterman, A. Hajnal, and T.A. Henzinger. Predictive modeling of signaling crosstalk during C. elegans vulval development. *PLoS Comput. Biol.*, 3(5):e92, 2007.
10. J. Fisher, N. Piterman, E.J.A. Hubbard, M.J. Stern, and D. Harel. Computational insights into Caenorhabditis elegans vulval development. *Proc Natl Acad Sci*, 102(6):1951–6, 2005.
11. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
12. D. Harel and H. Kugler. The Rhapsody semantics of statecharts (or, on the executable core of the UML). In LNCS 3147, pages 325–354. Springer, 2004.
13. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.
14. Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Inf. Comput.*, 163(1):203–243, 2000.
15. S.C. Kleene. *Introduction to Mathematics*. North Holland, 1987.
16. N. Lynch and M. Tuttle. An introduction to input/output automata. *Distributed Systems Engineering*, 1988.
17. L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *PPDP*, pages 82–93. ACM, 2005.
18. R. Milner. *A Calculus of Communicating Systems*, LNCS 92. Springer, 1980.
19. R. Milner. The polyadic pi-calculus (abstract). In *Concur*, LNCS 630. Springer, 1992.
20. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i & ii. *Inf. Comput.*, 100(1):1–77, 1992.
21. G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.