From a Formalized Parallel Action Language to its Efficient Code Generation

Ivan Llopard, CEA, LETI, MINATEC Campus, Grenoble, France. Univ. Grenoble Alpes, Grenoble, France.
Christian Fabre, CEA, LETI, MINATEC Campus, Grenoble, France. Univ. Grenoble Alpes, Grenoble, France.
Albert Cohen, INRIA and École Normale Supérieure, Paris, France

Modeling languages propose convenient abstractions and transformations to handle the complexity of today's embedded systems. Based on the formalism of Hierarchical State Machine, they enable the expression of hierarchical control parallelism. However, they face two importants challenges when it comes to model data-intensive applications: no unified approach that also accounts for data-parallel actions; and no effective code optimization and generation flows.

We propose a modeling language extended with parallel action semantics and hierarchical indexed-state machines suitable for computationally intensive applications. Together with its formal semantics, we present an optimizing model compiler aiming for the generation of efficient data-parallel implementations.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent programming, Parallel programming; D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods, State diagrams

General Terms: Languages, Algorithms, Performance

Additional Key Words and Phrases: Action Language, Parallels Languages, Model Driven Engineering

1. INTRODUCTION

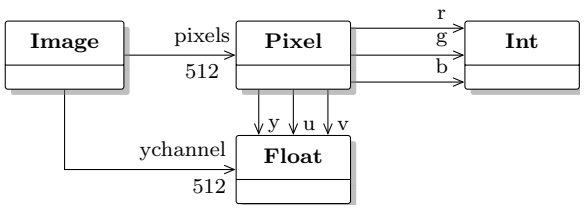
Embedded system design and development push for unified methodologies combining hardware and software [Henzinger and Sifakis 2006]. Model-Driven Engineering (MDE) techniques put forward simple model refinements as a process of specialization to specific platforms. Following the principles of model-based design, solutions emerged for embedded system design and development in a variety of applications domains [Gery et al. 2002; Object Management Group 2009; Gamatié et al. 2011; Hili et al. 2012; Object Management Group 2012; Gery et al. 2002].

Most of these frameworks rely on communicating Hierarchical State Machines (HSMs) as a model of computation, originally introduced in [Harel 1987] under the name of *Statecharts*, upon which different *action languages*—operations to be executed at each transition—are specified. Such approaches have signficant shortcomings in terms of expressiveness and efficient code generation: (*a)* they require the developer to make an up-front, bold distinction between the *control* part of an application, modeled as objects, associations and HSM, and its *data processing and computational* part, modeled as attributes and operations or coded in a foreign language [Gery et al. 2002]; (*b)* although they do capture *execution* parallelism, as each object's HSM executes independently, MDE techniques do not take advantage of the structural information in the model to exploit *communication*-based parallelism, such as data parallel and pipeline computations involving multiple instances of an individual association.

We intend to address these shortcomings by means of two complementary extensions: (*i)* modeling of *scalar data-types* with the same formalism and concepts used for regular, control-oriented,

This work is supported by the Artemis JU (Brussels, Belgium) and the *Ministère de l'Économie, du Redressement productif et du Numérique,* under Grant Agreement n°332 913, for project COPCAMS – http://copcams.eu.
Authors' addresses: ivan.llopard@cea.fr, christian.fabre1@cea.fr, albert.cohen@inria.fr.

DOI: http://dx.doi.org/10.1145/0000000.0000000



(a) $\langle$HOE$\rangle^2$ Object Model

```
object Image
  has [512]Pixel as pixels
  has [512]Float as ychannel

object Pixel
  has Int   as r, g, b
  has Float as y, u, v

scalar Int
```

(b) $\langle$HOE$\rangle^2$ Concrete Syntax

Fig. 1. Graphical and textual Image object model

objects; (*ii)* expressing *parallelism* and *data hierarchy* in a fashion that is both amenable to powerful analyses, like polyhedral techniques, and unified with control modeling based on message-passing.

More precisely, this paper builds on the Highly Heterogeneous, Object-Oriented, Efficient Engineering method, noted $\langle$HOE$\rangle^2$, that defines a parallel action language aiming at target-independence [Llopard et al. 2014; Hili et al. 2012]. While giving a broad view of the modeling language, we focus our contributions on three main aspects: *expressiveness*, *semantics* and *compilation*.

Expressiveness. This aspect is related to two language extensions that we called *generalized scalars* and *indexed regions*. We will introduce a new view of scalars in the context of modeling languages enabling arithmetic parallelism, including data-parallel operations. We also present the indexed region, a natural extension to the idea of composite states in the Harel Statecharts [Harel 1987]. We will see that they enable strong model optimizations, such as object inlining, and generation of efficient code at compilation time.

Semantics. We introduce an operational and hierarchical formalization of Statecharts as defined by $\langle$HOE$\rangle^2$. The approach relies on the language structure to model the Statecharts hierarchy, while existing formalizations tend to flatten this structure in general. It is also amenable to language extensions and its hierarchical definition allows us to separate concerns, reducing the complexity of the formalization.

Compilation. Similar to traditional compilation flows, we present an Intermediate Representation (IR) suitable for the compilation of communicating state machines. Although unusual in the domain of modeling languages, where models are translated directly into some low-level language (C/C++ or Java), we show that the IR enables the generation of efficient code through domain-specific optimizations.

The rest of this paper is structured as follows. Section 2 gives an overview of the modeling language, our main extensions and a concrete syntax used throughout the paper. In Section 3, we present our approach to the formal semantics of HSM in the context of $\langle$HOE$\rangle^2$. Section 4 introduces the Intermediate Representation (IR). The compilation flow based on this IR is presented in Section 5 together with the challenges of efficient code generation in the context of communicating automata. Section 6 illustrates our language and approach; we modeled data-intensive applications using the unified view of scalars and objects while producing efficient code. Related work is presented in Section 7 and finally we conclude with Section 8.

2. MODELING LANGUAGES: THE $\langle$HOE$\rangle^2$ACTION LANGUAGE

In the context of Model-Driven Engineering (MDE), the modeling language allows the designer to define the structure and the behavior of systems. The structure is captured by the specification of relations between different system components, or *objects*. Such relation is called an *association*. The behavior is described using *Statecharts*, or more formally HSM, which is a visual formalism for the specification of interactions between those system components [Harel 1987]. The Unified

Modeling Language (UML) is one of the most widely known languages in the field, largely accepted and supported by the industry [Object Management Group 2011].

The $\langle$HOE$\rangle^2$ modeling language inherits some structural features from UML, such as the object and association concepts. Two objects can be related to each other by means of an association and the association can be named and quantified. In the programming language sense, it is equivalent to the definition of a particular data type.

In order to specify the behavior of a particular component, the designer writes actions at each state transition using a specific *Action Language*. The $\langle$HOE$\rangle^2$ action language has been informally introduced in earlier work by Llopard et al. [Llopard et al. 2014]. This action language is parallel and specialized for parallel updates of object associations. Its formal semantics constitutes one of the contributions of the present paper and will be described in a follow-up section. The syntax structure of the transition is highly inspired from that of UML Statecharts and it is shown hereafter.

```
on <triggers> [ <guard> ] / <updates> : <sends>
```

The execution of the transition is enabled following a set of triggers, or messages, and a certain condition. The action language decouples the action into two imperative parts: update of associations (*i.e.*, assignments) and message sending.

We will give a quick introduction to the main features of the $\langle$HOE$\rangle^2$ language using the example of Figure 1. The example shows the object model of `Image` in graphical and textual notations, which can be viewed as a standard data type description of a digital Image. Listing 1 gives its behavior specification in textual form. It defines a state machine, `ImageSM`, with a creator called `raw`. At the creation transition, new `Pixel` objects are instantiated from a sequential array of RGB values where the iteration domain is enclosed between braces. We have two states, `GetY` and `GettingY`, with one and two outgoing transitions, respectively. From state `GetY`, we launch the computation of the `Y` (luminance) component by broadcasting message `getY` to all pixels where we found again an iteration domain. The iteration domain is used to send *indexed messages*. Note that the language uses a dot notation to denote a message sending action, where the destination is on the left-hand side and the message (together with its parameters) on the right-hand side. Then, the state `GettingY` collects all answers by capturing each particular instance of the indexed replies, together with its index value, to fill association `ychannel`. We bind the index values from a message using the braced notation in the triggering part, `takeY{i}`. Finally, the ending transition (**`endon`**) under the finishes the state machine execution. The condition `i.all` denotes the reception of all `takeY` messages.

```
object Image
  has [512] Pixel as pixels
  has [512] Float as ychannel
  sm ImageSM.
    creator raw(rawImage: Int[1536]) /
                { i: 0..pixels.len - 1 }
                pixels[i] = new Pixel.RGB(rawImage[3*i..3*i+2]) to GetY
    state GetY. on /: { i: 0..pixels.len - 1 } pixels[i].getY() to GettingY
    state GettingY. on takeY{i}(y: Float) / ychannel[i] = y to GettingY
                    endon [i.all]
```

Listing 1. Image object model

A specific feature of $\langle$HOE$\rangle^2$ is the interface definition of objects. The interface of $\langle$HOE$\rangle^2$ objects is an important language feature. As $\langle$HOE$\rangle^2$ objects are communicating state machines, they interact with each other by means of message passing. The set of valid messages that objects may exchange depends on their interface definition. The object interface specifies the set of accepted incoming and possible outgoing messages *with respect to the external world*. That is, it exports the set of input and output messages its users may observe. Everything else is related to internal message exchanges needed to fulfill the interface definition, not seen by external users.

```
object Pixel
  interface ins  getY()
            outs takeY(Float)
```

```
        on   getY() -> takeY(Float)
```

The above listing shows the definition of accepted input messages, `getY`, and output messages, `takeY`. Input-output relations can also be defined. For instance, `takeY` is declared to be a consequence of the reception of `getY` under all program contexts. The messages needed to guarantee such relation are not seen by Pixel users.

Within the context of modeling languages, and particularly $\langle \text{HOE} \rangle^2$, we propose two extensions: *generalized scalars* and *indexed regions*.

2.1. Scalars

Every language provides a set of scalars as built-in types, *e.g.*, **`int`** or **`float`** in C, C++ or Java. High-level languages allow to lift built-in arithmetic operations to composite data types, and to define new operations with custom properties. The designer of a domain library may use such capabilities to define an extended set of *generalized scalars*. These domain-specific scalars share an intrinsic property across programming languages: *immutability*. They usually denote the carrier set of some algebraic structure. Equational reasoning on this structure may be essential to productive developments and domain optimizations/simplifications. Indeed, many compiler optimizations are built upon properties of scalars, *e.g.*, constant propagation, strength reduction or value numbering [Aho et al. 2006].

On the other hand, the behavior of built-in types has a strong influence on the language design. For instance, let us consider the pure, non-strict functional language Haskell. Due to its lazy evaluation strategy, variables of any type, including primitives types, can be undefined until their values are required [Scott 1971]. The *undefined value*, noted $\perp$, is part of all Haskell types and must be taken into account. Technically, the language calls the integer type `Int` a *boxed type* even though standard arithmetics is applicable on them and all known algebraic properties still hold. A specific analysis called *the strictness analysis* tries to avoid boxed types as much as possible in order to improve performance trading-off its non-strict semantics at compilation time [Brown and Wilson 2012].

Data-flow languages introduce another interesting example of built-in types [Caspi et al. 1987; Gamatié 2009]. In the Kahn denotational semantics of data-flow languages [Kahn 1974], everything is a *stream*, even primitives values. A simple integer value in such programming language denotes a stream of values where all arithmetic operations are applied point-wise. This feature allows the programmer to easily write parallel programs, increasing the language expressiveness.

Inspired from these remarks and the intrinsic properties of scalars, we define an extended view of *scalars* as *communicating state machines*. To preserve the algebraic properties of scalars, including referential transparency, *we choose a functional semantics for these state machines*: a transition results in the construction of a *fresh machine in a new state*. As a result, in a functional setting, scalar values follow the same semantics as generic immutable objects. This has a lot of advantages in modeling languages. For instance, consider the state machine of a `Pixel` object shown in Listing 2. It extracts the luminance information from a `Pixel` in RGB format interacting with scalars, which can communicate. Note that all interactions ("operations") with scalars, at `ComputeY` for instance, are explicitly parallel and equivalent to generic objects from the modeling perspective. Such interactions model arithmetic operations in term of message passing. As a side-effect, it introduces a natural notation for (data) parallelism. The representation of scalars is compliant with the message passing semantics of $\langle \text{HOE} \rangle^2$, providing a consistent and homogeneous abstraction to the language.

```
object Pixel
  has Int   as r, g, b
  has Float as y
  sm PixelSM.
    creator RGB(rgb: Float[3])/ r = rgb[0], g = rgb[1], b = rgb[2]
                    to ComputeY
    // Wait for getY and launch multiplications
    state ComputeY.   on getY() / : r.mult(0.299), g.mult(0.587), b.mult(0.114)
                    to Multing
    // Collect two multiplications and launch the addition
```

```
    state Multing.    on multed(v1: Float), multed(v2: Float) / : v1.fadd(v2)
                      to MultAdding
    // Collect last multiplications and launch another addition
    state MultAdding. on multed(v3: Float), fadded(v4: Float) / : v3.fadd(v4)
                      to Adding
    // Reply to the sender of getY by using keyword "initiator"
    state Adding.     on fadded(result: Float) / y = result: initiator.takeY(y)
                      to ComputeY
```

Listing 2. Pixel object model

The scalar *interface* defines input-output message relations that we call *operational transitions*. For instance, the listing below shows a scalar representation of Int objects with operational transitions `addOp` and `multOp`.

```
scalar Int
  interface on add(Int)  -> added(Int)  ~> addOp
            on mult(Int) -> multed(Int) ~> multOp
  model Z where (+): addOp, (*): multOp
```

The `Int` operational transition models the beginning of an operation and its acknowledgment in an asynchronous manner. We will show in Section 5 that operational transitions are *foldable* in the context of the intermediate representation. It means that send and receive pairs can be transformed into an "in-place" operation, which is exactly what the scalar interface captures: *i.e.*, one may also see `addOp` as a function $addOp : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$.

From messages to operations. More generally, the question arises about what type of arithmetic operation a message exchange may model in the particular case of scalars. The answer should allow us to replace the message exchange by an efficient in-place operation whenever possible. Let us consider the Float scalar as another example. The interface entry for the division operator is defined as follows:

```
scalar Float
  interface on div(Float) -> dived(Float) or error() ~> divOp
```

The input message captures the operation itself—the `div` type constructor—and the floating point divisor. The outgoing message has a sum type, $Dived$ or $Error$, describing the possible outcomes of the division. Internally (and algebraically) `divOp` can be seen as a function $divOp : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$, where $\mathbb{F}$ denotes the set of floating point numbers in the IEEE754 standard [IEEE754 2008].

Following the IEEE754 standard again, one may refine the definition of the division operator:

```
scalar Float
  interface on div(Float) -> dived(Float) or nan() or inf() or nInf() ~> divOp
```

The more precise sum type of reply messages distinguishes among the possible non-standard outcomes of the division. In a nutshell, *the purpose of the input message is to represent the operation with a specific number of parameters, while output messages model its result type*.

2.2. Hierarchical Composition and Indexed Regions

In Statecharts, the regions inside states represent parallel portions of code, modeled using state machines, hence the hierarchical composition. It is the essence of the well-known Statecharts [Harel 1987]. Whenever the state machine enters a *composite* state (state containing one or more regions), it automatically jumps to the entry point of each of its internal regions.

Let us extend the HSM formalism with *indexed regions* where its graphical notation is presented in Figure 2. It represents N parallel regions, each one indexed by i. The i-th region contains two states, A_i and B_i, where A_i is the initial one. Indexed regions model "forall" block expressions, found natively in other parallel languages or as parallel extensions to sequential languages [Chakravarty et al. 2007; Chapman et al. 2007]. Besides its expressiveness, we will see that such a construction is important when aiming for efficient code generation.

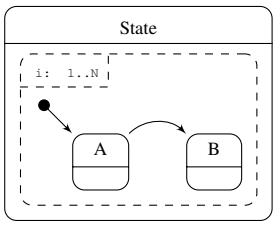


Fig. 2. $\langle \text{HOE} \rangle^2$ Indexed regions

2.3. Syntax

To introduce our approach to the formal semantics of hierarhical state machines and show the translation of $\langle \text{HOE} \rangle^2$ into our IR, let us introduce a concrete grammar of the top-level concepts. Unless indicated otherwise and for conciseness purposes, the grammar operators $\langle rule \rangle$* and $\langle rule \rangle$+ define comma-separated statements.

An object is a type definition containing a state machine and defining associations to other objects.

$\langle object \rangle$ ::= **'object'** $\langle id \rangle$ $\langle interface \rangle$ $\langle associations \rangle$ $\langle sm \rangle$

Therefore, $\langle object \rangle$ is simply a record type constructor. $\langle associations \rangle$ is a sequence of tuples (f_i, t_i) where f_i is the field name (or role name) and t_i the association type. t_i is defined as

$\langle T \rangle$ ::= $\langle t \rangle$ | $\langle t \rangle$ **'['** $\langle R \rangle$+ **']'**
$\langle R \rangle$ ::= $\langle INT \rangle$ | $\langle INT \rangle$ **'..'** $\langle INT \rangle$ | $\langle INT \rangle$ **'..'** **'*'** | **'*'**

where the array type constructor may have integer ranges, a very common feature among modeling languages.

Together with each new record type, the programmer defines an interface and its HSM in an object-oriented way.

2.3.1. State Machine. $\langle \text{HOE} \rangle^2$ state machines contain creators and states.

$\langle sm \rangle$ ::= **'sm'** $\langle id \rangle$ **'.'** $\langle creator \rangle$+ $\langle state \rangle$+
$\langle creator \rangle$ ::= **'creator'** $\langle id \rangle$ **'('** $\langle param \rangle$* **')'** [**'/'** $\langle update \rangle$+] $\langle to \rangle$

States can be simple or composite.

$\langle state \rangle$ ::= $\langle sstate \rangle$ | $\langle cstate \rangle$

Simple states are composed by external and final transitions.

$\langle sstate \rangle$ ::= **'state'** $\langle id \rangle$ **'.'** $\langle trn \rangle$+
$\langle trn \rangle$ ::= $\langle external \rangle$ | $\langle final \rangle$
$\langle external \rangle$ ::= **'on'** $\langle trg \rangle$* [**'['** $\langle guard \rangle$ **']'**] **'/'** [$\langle action \rangle$] **'to'** $\langle id \rangle$
$\langle final \rangle$ ::= **'endon'** $\langle trigger \rangle$* [**'['** $\langle guard \rangle$ **']'**] [**'/'** $\langle action \rangle$]

The composite state defines regions denoting parallel composition.

$\langle cstate \rangle$ ::= **'cstate'** $\langle id \rangle$ **'.'** $\langle region \rangle$+ $\langle transition \rangle$+
$\langle region \rangle$ ::= **'region'** [$\langle indexset \rangle$] $\langle initial \rangle$ $\langle state \rangle$+
'endregion'

The initial transition indicates the entry state of regions.

$\langle initial \rangle$ ::= **'initial'** $\langle id \rangle$

$\langle \text{HOE} \rangle^2$ actions are defined as

⟨*action*⟩ ::= ⟨*update*⟩* [**':'** ⟨*send*⟩+]
⟨*update*⟩ ::= ⟨*supdate*⟩ | ⟨*iupdate*⟩
⟨*send*⟩ ::= ⟨*ssend*⟩ | ⟨*isend*⟩

where ⟨*iupdate*⟩ and ⟨*isend*⟩ are indexed update and send, respectively. Note that triggers can also be indexed where the index set specifies the set of accepted index values for the trigger to be enabled. Indexed expressions were introduced informally with the example of Listing 1.

3. FORMAL SEMANTICS OF HIERARCHICAL STATE MACHINES

Formalizing modeling languages is challenging. Since the introduction of the block-diagram formalism of Statecharts, many authors have proposed interesting approaches with Statecharts as the main formalism, or embedded into modeling languages such as UML [Mikk et al. 1997; Liu et al. 2013; Seifert 2008; Brger et al. 2004]. Existing approaches deal with flattened versions of Statecharts implementing specific mathematical structures to handle the *hierarchy* of states.

The complexity of the existing formalizations led to rather limited semantic specifications. None of these attempts leverage Structural Operational Semantics [Plotkin 2004]. This is the starting point of our work, aiming to embed non-determinism and hierarchy in a more natural way, and providing a formal basis that remains easy to read and amenable to language extensions.

From the syntax definition, we present a layered approach on which the semantics of ⟨HOE⟩^{2} actions form the bottom layer. On top of it, we define the semantics of transitions and finally the semantics of state machine configurations.

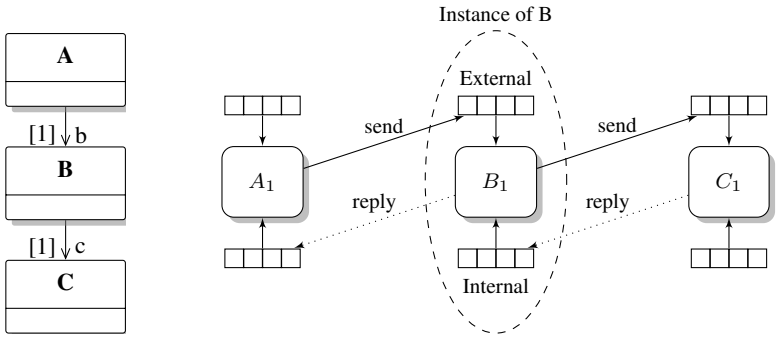
Before entering into specific definitions, let us present the very general picture of communications among ⟨HOE⟩^{2} objects. ⟨HOE⟩^{2} is based on asynchronous state machines communicating through message passing. Each object corresponds to a state machine implementing *external* and *internal* First In-First Out (FIFO) buffers. As mentioned in section 2, the interface definition exposes valid exchange of messages. The messages not shown at the interface are not visible to the external world and hence they flow through an internal FIFO buffer. Figure 3 shows the proposed communication flow that exposes external and internal FIFOs. In 3(a) we have a simple ⟨HOE⟩^{2} object model and Figure 3(b) shows the communication flow of a given instance of such model. For instance, object B_1 will write to external FIFO C_1 using send primitives whereas all replies from C_1 to B_1, triggered by `initiator` inside C_1, will write to the internal FIFO of B_1. Note that objects can be shared, and hence we may have multiple producers for the external or internal FIFO buffers. However, there is always one consumer per buffer. The selection of asynchronous semantics is consistent with the need to model non-deterministic choice when receiving messages from multiple producers in UML.

The message reception procedure will look at external FIFO if the required message belongs to the object interface. Otherwise, it looks at the internal FIFO for any available message. In case we have an available message, we follow a message dropping policy. That is, if the popped message does not correspond to the current waiting one, then it is dropped.

System. The set of communicating objects forms the *system*, or simply the program state. We define the system as a memory mapping slots to objects. Formally, it is defined as $\mathcal{S} = (\mathcal{R}, A)$, the current object reference and the mapping $A = \mathcal{R} \to O$ from references to objects where $\mathcal{R}$ is the finite set of references (or memory slots). Note that mapping A may grow as new objects are created and, hence introduced into the system.

Objects. The object has an associated state machine program, its current configuration, message pool and its corresponding valuation context. We define the object as $O = (\langle sm \rangle, K, E, \Phi)$. The configuration $k \in K$, defined more precisely later, corresponds to its active hierarchical states.

The object context $\Phi = \mathcal{V} \to V$ maps variable identifiers to values. The set of values V is made of references, which implies that Φ is only valid under a certain system. More precisely, values are $V = \mathcal{R} + \bar{A} + \mathbb{Z} + \mathbf{Null}$. That is, we may have references, arrays $\bar{A}$ which are indexed maps to references as usual, index values $\mathbb{Z}$ and the null reference $\mathbf{Null}$.



(a) $\langle \text{HOE} \rangle^2$ Object model

(b) $\langle \text{HOE} \rangle^2$ Instance model with external and internal FIFO

Fig. 3. Functional view of communicating $\langle \text{HOE} \rangle^2$ objects

$$a, a' \in \Gamma_A \qquad t_i, t'_i \in \Gamma_T \qquad k_i, k'_i \in \Gamma_K$$

$$\frac{}{a \to a'} \qquad \frac{a \to a' \qquad t_1 \to t'_1}{t_2 \to t'_2} \qquad \frac{t \to t' \qquad k_1 \to k'_1}{k_2 \to k'_2}$$

Fig. 4. Hierarchical semantics approach

The objects communicates through message passing. As explained earlier, they contain input message pools or FIFOs. Instead of implementing two different ones, we choose to mark each message as internal or external as required. Then, the message pool is a list of messages $E = \overline{M}$.

Messages. The message $M = (\mathcal{R}, I, \mathcal{M}, \mathbb{B}, \Psi)$ contains the sender reference, a sequence of index values $I = [\mathbb{Z}]$, a valid message identifier $\mathcal{M}$, a boolean flag to indicate its external or internal status, and a mapping to its parameter references $\Psi = \mathbb{N} \to V$.

In the following sections, we present required notations and definitions. Then we develop our hierarchical approach where the semantics of $\langle \text{HOE} \rangle^2$ actions, Γ_A, form the bottom layer. On top of it, we define the semantics of transitions, Γ_T, and finally the semantics of state machine configurations, Γ_K. Figure 4 describes the general mathematical view. A transition relation on Γ_K depends on Γ_T, which in turns depends on Γ_A, hence forming a decoupled approach. This separation allow us to deal with each particular problem (*e.g.* initiator semantics, message send/reception, message dropping policy) mostly in isolation.

3.1. Semantics of Actions

Hereafter, we introduce some useful notations:

— We note $sys^{[r \mapsto o]}$ the new system that maps a reference r to object o.
— We use α to range over the $\langle \text{HOE} \rangle^2$ syntax as defined in section 2.3.
— Function update is defined as $f' = [f \mid r' \mapsto o]$, which is equivalent to $f'(r) = \textbf{if}\, r' = r\, \textbf{then}\, o\, \textbf{else}\, f(r)$.
— We note projections with subscripts, *e.g.*, A_{sys} is the mapping of system $sys \in \mathcal{S}$.
— The context extension operator $\triangleright : \Phi \to \Phi \to \Phi$ is defined as

$$\phi \triangleright \phi' = \lambda ref.\textbf{if}\, \phi(ref) = \textbf{Null}\, \textbf{then}\, \phi'(ref)\, \textbf{else}\, \phi(ref) \tag{1}$$

In the following, we consider $sys = (\hat{r}, A)$ where $\hat{o} = A_{sys}(\hat{r})$ is the object under evaluation and $\hat{r}$ the current reference.

Following the language definition, we define a transition semantics on $\Gamma_A = \langle action \rangle \times \Phi \times \mathcal{S} + \Phi \times \mathcal{S}$ where $\Phi : \mathcal{V} \to V$ represents the local context, *i.e.*, the bindings at the transition level.

The set of actions, $\langle$*action*$\rangle$, implements comma-separated parallel update and send primitives, both sequentially ordered by the separator '**:**'. Non-determinism in case of parallel execution and sequentiality are simple to specify thanks to the operational semantics approach. Let $\alpha_{action} = \alpha_{update}$**:**$\alpha_{send}$, they are introduced by the following rules

$$\frac{(\alpha_{update}, \phi, sys) \to (\alpha'_{update}, \phi', sys')}{(\alpha_{update}\textbf{:}\alpha_{send}, \phi, sys) \to (\alpha'_{update}\textbf{:}\ \alpha_{send}, \phi', sys')}\ \text{ASeq}$$

$$\frac{(\alpha_{update}, \phi, sys) \to (\phi', sys')}{(\alpha_{update}\textbf{:}\alpha_{send}, \phi, sys) \to (\alpha_{send}, \phi', sys')}\ \text{ASeqEnd}$$

where parallel compositions of updates (with equivalent rules for sends) are given below.

$$\frac{(\alpha_{update}, \phi, sys) \to (\alpha'_{update}, \phi', sys')}{(\alpha_{update}\textbf{,}\ \alpha''_{update}, \phi, sys) \to (\alpha'_{update}\textbf{,}\ \alpha''_{update}, \phi', sys')}\ \text{AParUL}$$

$$\frac{(\alpha_{update}, \phi, sys) \to (\alpha'_{update}, \phi', sys')}{(\alpha''_{update}\textbf{,}\ \alpha_{update}, \phi, sys) \to (\alpha''_{update}\textbf{,}\ \alpha'_{update}, \phi', sys')}\ \text{AParUR}$$

For instance, ASeqEnd gives a new relation if there exists an evaluation on Γ_A of the action α_{update} under context ϕ and system sys terminating into final context ϕ' and system sys', then the sequential statement below evaluates to the send action under such new system. From there, we continue to evaluate such send using the new system.

Non-determinism is cleanly expressed with rules AParUL and AParUR. In case of parallel updates, they allow both udpates to be evaluated without prior order. It can be shown that non-deterministic inference rules have an equivalent denotational semantics on power-domains [Reynolds 1999].

$\langle$HOE$\rangle^2$ has three mains actions: update, send and receive. The interpretation of one transition involve all combinations of them. Given that the communication model is "single consumer-multiple producers", then an object may modify other objects in the system by pushing new messages to their event pool. It may also add new objects to the system, hence new references.

The premises for the parallel and sequential rules are given by the specific semantics of actions, which we present hereafter.

Update. For completeness, we introduce the denotational semantics of simple updates

$$\frac{(\phi', sys') = [\![\alpha_{supdate}]\!](\phi, sys)}{(\alpha_{update}, \phi, sys) \to (\phi', sys')}\ \text{ASUpdate}$$

where the grammar is defined as

$\langle$*supdate*$\rangle$ ::= $\langle$*var*$\rangle$ **'='** ($\langle$*var*$\rangle$ | $\langle$*new*$\rangle$)

The left-hand side of updates, $\langle$*var*$\rangle$, are single or array variables while the right-hand side may also contain **`new`** expressions denoting object creation. For the particular case of single variables, we have $[\![]\!] : \langle update \rangle \to (\Phi \to \mathcal{S}) \to (\Phi \to \mathcal{S})$ defined as

$$
\begin{array}{lll}
[\![\mathbf{v} = \mathbf{v'}]\!](\phi, sys) & = (\phi, sys^{[\hat{r} \mapsto o]}) & \\
& \text{where} & \phi' = \phi \rhd \phi_{\hat{o}} \\
& & o = (sm_{\hat{o}}, k_{\hat{o}}, e_{\hat{o}}, [\phi_{\hat{o}} \mid \mathbf{v} \mapsto \phi'(\mathbf{v'})]) \\
[\![\mathbf{v} = \alpha_{new}]\!](\phi, sys) & = (\phi, sys^{[\hat{r} \mapsto o, r' \mapsto o']}) & \\
& \text{where} & (r', o') = [\![\alpha_{new}]\!](\phi \rhd \phi_{\hat{o}}, sys) \\
& & o' = (sm_{\hat{o}}, k_{\hat{o}}, e_{\hat{o}}, [\phi_{\hat{o}} \mid \mathbf{v} \mapsto r'])
\end{array}
$$

Let us explain the two variable assignment semantics. We take the local context ϕ which we extend with the object context $\phi_{\hat{o}}$ in order to obtain the reference of v'. Then, we update the object context $\phi_{\hat{o}}$ with such reference $[\phi_{\hat{o}} \mid \mathbf{v} \mapsto \phi'(\mathbf{v'})]$. Finally, the updated object o replaces the current one, $sys^{[\hat{r} \mapsto o]}$.

These new definitions enable the evaluation of upper constructions such as those presented earlier (*e.g.* ASeqEnd). We continue with the semantics of sends and receives.

Send. Similar to updates, we define a denotational semantics of sends

$$\frac{(\phi', sys') = [\![\alpha_{ssend}]\!](\phi, sys)}{(\alpha_{send}, \phi, sys) \rightarrow (\phi', sys')} \text{ ASSend}$$

where the grammar of simple sends is defined as

$\langle \textit{ssend} \rangle$ ::= $\langle \textit{var} \rangle$ **'.'** $\langle \textit{msg} \rangle$

Let $push = \lambda o, m.(sm_o, k_o, e_o{+}{+}\overline{m}, \phi_o)$ be the function that pushes message m into object o ('++' denotes list concatenation), and $[\![]\!]_{msg} : \langle msg \rangle \rightarrow (\Phi \rightarrow \mathcal{S}) \rightarrow M$ the function that takes a message definition and instantiate $m \in M$. Then, the denotation of updates $[\![]\!] : \langle update \rangle \rightarrow (\Phi \rightarrow \mathcal{S}) \rightarrow (\Phi \rightarrow \mathcal{S})$ for the case of single variables is defined as

$$
\begin{array}{lll}
[\![\mathbf{v} \,.\, \alpha_{msg}]\!](\phi, sys) & = (\phi, sys^{[r' \mapsto o']}) & \\
& \text{where} & \phi' = \phi \rhd \phi_{\hat{o}} \\
& & r' = \phi'(\mathbf{v}) \\
& & o' = push(A_{sys}(r'), [\![\alpha_{msg}]\!](\phi', sys))
\end{array}
$$

Note that the actions are similar to the update semantics. We extend the local context and get the referenced object by v. We proceed with the push and update the system accordingly.

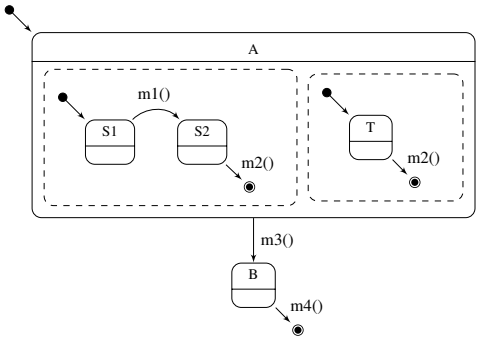
Receive. In order to evaluate the entire transition, we need to define the semantics of receives expressions. They control the evaluation of actions. Let $t = (\alpha_{trigger}, \alpha_{guard}, \alpha_{action}, \alpha_{to}) \in \langle trn \rangle$, we define a transition relation on

$$\Gamma_T = \langle trn \rangle \times \mathcal{S} + \mathcal{S} \tag{2}$$

Let $a, a' \in \Gamma_A$, we note $a \rightarrow^* a'$ the transitive closure of relation $\rightarrow$ on Γ_A. Transitions are evaluated unconditionally if there is no trigger provided that the guard is true.

$$\frac{\alpha_{trigger} = \emptyset \wedge [\![\alpha_{guard}]\!]_g(\phi_{\hat{o}}, sys) \wedge (\alpha_{action}, \phi_{\hat{o}}, sys) \rightarrow^* (\phi', sys')}{(\alpha_{trn}, sys) \rightarrow sys'} \text{ ENoTrigger}$$

If there is a trigger, we check if the first message in the event pool matches it and guard evaluates to true under the local binding context, which is constructed from the input message by $mbind : M \rightarrow \langle trigger \rangle \rightarrow \Phi$.



(a) ⟨HOE⟩^{2} Graphical Representation

```
sm SM.
  creator c() to A
  cstate A. region
              initial S1
              state S1. on m1() to S2
              state S2. endon m2()
            endregion
            region
              initial T
              state T. endon m2()
            endregion
            on m3() to B
  state B. endon m4()
```

(b) ⟨HOE⟩^{2} Code

Fig. 5. Non-Indexed model

$$\frac{\begin{array}{l}\hat{o} = (sm, k, m :: e', \phi) \wedge match(m, \alpha_{trigger}) \\ \phi_m = mbind(m, \alpha_{trigger}) \rhd \phi \wedge [\![\alpha_{guard}]\!]_g(\phi_m, sys) \\ (\alpha_{action}, \phi_m, sys) \rightarrow^* (\phi', sys')\end{array}}{\hat{o}' = A_{sys'}(\hat{r}) \wedge (\alpha_{trn}, sys) \rightarrow sys'^{[\hat{r} \mapsto (sm_{\hat{o}'}, k_{\hat{o}'}, e', \phi_{\hat{o}'})]}} \text{ ETrigger}$$

Note that we must pop message m from the event pool of $\hat{o}$ to construct the new system sys' with $\hat{o}'$.

It is important to note that the transition rules are possible if and only if the evaluation of actions is defined, hence our layered approach.

3.2. State Machine Evaluation

Given the semantics of transitions, we define here the hierarchical state machine step. We start by considering the non-indexed configuration model. A configuration indicates the current *active* states down the hierarchy of the HSM. Contrary to existing Statecharts semantics involving a transitive relation of substates to model the hierarchy, we create a recursive configuration that closely follow the input language structure.

Therefore, we must model simple, composite and final conditions. We define the hierarchical state machine configuration:

$$K(a) = a + a \times [K(a)] + End \tag{3}$$

with injections for simple configurations $\iota_s : a \rightarrow K(a)$, composite $\iota_c : a \rightarrow [K(a)] \rightarrow K(a)$ configurations, and $\iota_{end} = End$. For instance, if we take configurations over state identifiers $K(String)$ then one possible configuration of the state machine shown in Figure 5 is:

$$k = \iota_c(\text{'A'}, [\iota_s(\text{'S1'}), \iota_s(\text{'T'})]) \tag{4}$$

We define the K transition semantics by lifting $\langle state \rangle$ such that $\Gamma_K = K(\langle state \rangle) \times \mathcal{S} + \mathcal{S}$ and we note the relation $\rightsquigarrow$. Using the evaluation relation defined on (2), we introduce the set of evaluation rules shown in Figure 6 where α^i_{trn} is the i-th transition out of $s \in \langle state \rangle$ and the next state configuration function $nextK : \langle sm \rangle \rightarrow \langle id \rangle \rightarrow K(\langle state \rangle)$. The rules $KsExt$ and $KcExt$ handle the evaluation of simple and composite configurations in presence of *external* transitions. In the same manner, the rules $KsFinal$ and $KcFinal$ cover the simple and composite configurations in case of *final* transitions. Finally, $KPar$ implements the parallel semantics where any parallel region may be evaluated. We illustrate these rules with an example.

$$\frac{(\alpha^i_{trn}, sys) \to sys' \wedge isExtT(\alpha^i_{trn}) \wedge ks' = nextK(sm, \alpha^i_{to})}{(\iota_s(s), sys) \rightsquigarrow (ks', sys')} \text{ KsExt}$$

$$\frac{(\alpha^i_{trn}, sys) \to sys' \wedge isEndT(\alpha^i_{trn})}{(\iota_s(s), sys) \rightsquigarrow (\iota_{end}, sys')} \text{ KsFinal}$$

$$\frac{(\alpha^i_{trn}, sys) \to sys' \wedge isExtT(\alpha^i_{trn}) \wedge ks' = nextK(sm, \alpha^i_{to}) \wedge \forall kl_i \in kl \mid kl = \iota_{end}}{(\iota_c(s, kl), sys) \rightsquigarrow (ks', sys')} \text{ KcExt}$$

$$\frac{(\alpha^i_{trn}, sys) \to sys' \wedge isEndT(\alpha^i_{trn}) \wedge \forall kl_i \in kl \mid kl_i = \iota_{end}}{(\iota_c(s, kl), sys) \rightsquigarrow (\iota_{end}, sys')} \text{ KcFinal}$$

$$\frac{(ks_i, sys) \rightsquigarrow (ks'_i, sys')}{(\iota_c(s, [\cdots, ks_i, \cdots]), sys) \rightsquigarrow (\iota_c(s, [\cdots, ks'_i, \cdots]), sys')} \text{ KPar}$$

Fig. 6. Evaluation rules of configuration K

Example. As a running example, consider the state machine of Figure 5 at configuration (4). Let $e_{\hat{o}} = [m]$ be the message pool of the object under evaluation $\hat{o}$ where message m contains the identifier M_m = m1. For the sake of simplicity, we assume that system sys contains only one object, *i.e.* $sys = (\hat{r}, [\hat{r} \mapsto \hat{o}])$.

The state machine evaluation of $\hat{o}$ starts at the configuration level, Γ_K. We observe that the current configuration is composite ι_c, then rule $KPar$ applies. The rule asks to evaluate a particular region. Let us take the left region configuration of state A, *i.e.* $\iota_s('S1')$. The configuration of this region is a simple one where we see that rule $KsExt$ applies. Such rule evaluates the configuration into another one providing that there exists an evaluation (relation) on Γ_T such that we have an external transition and it evaluates to sys'.

Indeed, from state S1 we can go to S2 because message $m1$ is present into the message pool of the object under evaluation as stated earlier, then rule $ETrigger$ applies. In this particular case we have no action. Nevertheless, we see that the idea applies hierarchically. That is, rule $ETrigger$ will try to evaluate the action on Γ_A.

Finally, the new configuration will be $\iota_s('S2')$ under system sys'. In sys', object $\hat{o}$ will not contain message m1 anymore. Note that in general k and k' may not have the same hierarchy.

3.3. Indexed Configurations

We extend the hierarchical configuration K of (3) as follows

$$K(a) = a \times [IK(a)] + a + End$$
$$IK(a) = K(a) + \mathbb{Z}^n \to K(a)$$

We model indexed configurations $IK(a)$ as a disjoint union of classical ones $K(a)$ and indexed ones, which we model with a partial function from indexes, $\mathbb{Z}^n$, to configurations.

Let $\iota_k : K(a) \to IK(a)$ and $\iota_i : \mathbb{Z}^n \to K(a) \to IK(a)$ be the injections for classical and indexed regions, respectively. Considering first region of Figure 5 to be indexed with index set `[i: 0 <= i <= 255]`, we extend sample configuration (4)

$$k = \iota_c(\text{'A'}, [(\iota_k \circ \iota_s)(\text{'S'}), \iota_i(fr)]) \tag{5}$$

where

$$fr(i) = \begin{cases} \iota_s(\text{'T'}) & \text{if } 0 \leq i \leq 255 \\ End & \text{otherwise} \end{cases}$$

```
1  object Image {
2    associations {
3      Pixel|512| pixels;
4      Float|512| ychannel;
5    }
6  }
7  fsm Image.ImageSM(Image this) {
8  GetY:
9    forall [i: 0 <= i and i < 512] sendfrom[i] this this.pixels[i] getY<>;
10   goto GettingY;
11 GettingY:
12   wait this for msg_takeY = recv[i: 0 <= i and i < 512] takeY<Float>
13     then when msg_takeY   goto UpdateY
14     then if msg_takeY.all goto Final;
15 UpdateY:
16   this.ychannel[i] = msg_takeY.y;
17   goto GettingY;
18 Final:
19   done this;
20 }
21 creator Image.raw(Integer|1536| rawImage) {
22   this = new Image;
23   Int|3| rawSlice;
24   forall [i: 0 <= i and i < 512] {
25     forall [j: 3*i <= j and j <= 3*i+2] rawSlice[j - 3*i] = rawImage[j];
26     this.pixels[i] = create Pixel.RGB rawSlice;
27   }
28   start ImageSM of this;
29 }
```

Listing 3. Translated object model of an Image

The indexed configuration (5) puts the state machine at state A, where its nested first region is at state S and the second indexed one is at T_i such that $0 \leq i \leq 255$.

4. INTERMEDIATE REPRESENTATION

When targeting languages such as C/C++ or Java, existing model compilers use a particular implementation of state machines. The analysis and optimization work is left entirely to the host compiler, which does not know about the initial model and is therefore unable to discover state machine related optimizations (*e.g.* unreachable states). A new language as a bridge between the front-end and target languages is necessary to handle high-level optimizations and translation issues related to each particular target language. We present an Intermediate Representation (IR) suitable for the efficient compilation and optimization of communicating HSMs. As we will see, the IR language is very expressive supporting creation and destruction of state machines, send and receives primitives, forall loops and multidimensional arrays, among other features.

The IR disassociates structural definitions from behavioral specification. It allows to "compile the concurrency of the model into platform-specific threads and operations", by making asynchronous messages among state machines explicit, and by capturing their substitution with (sequential) in-place operations. Syntactically, it is inspired from the object-oriented implementation in procedural imperative languages. To give a quick overview on the purpose and design of the IR, consider the translation of Listing 1 into the IR code of Listing 3. Type definitions, state machines and creators are split into different code blocks. They contain imperative code where each statement is a parallel list of expressions, closely related to the $\langle \text{HOE} \rangle^2$ action language. We can observe a parallel and indexed send at Line 9 and an indexed receive at Line 12. Creators are implemented in another scope where, for instance, we can see the creation of pixels at Line 26.

More precisely, the available statements are: send and receive expressions: *sendfrom*, *reply*; forall expressions; unconditional branches: *goto*; communication-dependent and guarded branches: *wait*, *wait-in*; updates with dot-like and bracket-like structural accesses; inline application of operational

transitions; creation and destruction of automata: *create*, *done*; structure allocation: *new* (only allowed on creators);

Formally, a statement in the IR is defined as

⟨*stmt*⟩ ::= ⟨*import*⟩ | ⟨*object*⟩ | ⟨*scalar*⟩ | ⟨*creator*⟩ | ⟨*fsm*⟩

The ⟨*import*⟩ rule adds new type definitions into the type context whereas object and scalar are type definitions

⟨*object*⟩ ::= **'object'** ⟨*ID*⟩ **'{'** ⟨*interface*⟩ ⟨*associationlist*⟩ **'}'**
⟨*scalar*⟩ ::= **'scalar'** ⟨*ID*⟩ **'{'** ⟨*interface*⟩ ⟨*representation*⟩ **'}'**

4.1. Hierarchical State Machine

A statement in the state machine is

⟨*fsmstmt*⟩ ::= ⟨*fsmstmt*⟩ **';'** ⟨*fsmstmt*⟩ | ⟨*parstmt*⟩ | ⟨*forall*⟩
| ⟨*vardecl*⟩ | ⟨*wait*⟩ | ⟨*waitin*⟩ | ⟨*goto*⟩ | ⟨*done*⟩

Hereafter, we present a comprehensive description of each one of them.

Parallel Statements. Similiar to ⟨HOE⟩[2], the IR proposes parallel (indexed) updates and send expressions. These are introduced by the ⟨*parstmt*⟩ rule:

⟨*parstmt*⟩ ::= ⟨*parstmt*⟩ **','** ⟨*parstmt*⟩ | ⟨*sendexpr*⟩ | ⟨*updateexpr*⟩
⟨*sendexpr*⟩ ::= ⟨*send*⟩ | **'forall' '['** ⟨*indexset*⟩ **']'** ⟨*send*⟩
⟨*updateexpr*⟩ ::= ⟨*update*⟩ | **'forall' '['** ⟨*indexset*⟩ **']'** ⟨*update*⟩

Send and updates can be enclosed inside index domains providing forall semantics. We use an explicit keyword, `forall`, with its index set between brackets.

The statement `sendfrom` contains a (optionally) list of comma-separated arithmetic expressions ⟨*arithexprs*⟩, afterwards it must be present source and target objects, the message to be sent and all the required parameters according to the message type.

⟨*sendfrom*⟩ ::= **'sendfrom' ['['** ⟨*arithexprs*⟩ **']']** ⟨*var*⟩ ⟨*var*⟩ ⟨*msgtype*⟩ ⟨*var*⟩*

The ⟨*update*⟩ rule is

⟨*update*⟩ ::= ⟨*vardef*⟩ **'='** (⟨*varexpr*⟩ | ⟨*create*⟩ | ⟨*applyon*⟩)

where `create` and `applyon` denote creation of new objects and "in-place" operation, respectively. The scalar definition abstracts away in-place operations which are materialized at the IR using statement `applyon`. Rule ⟨*vardef*⟩ is either a ⟨*varexpr*⟩ or a variable declaration where ⟨*varexpr*⟩ is

⟨*varexpr*⟩ ::= ⟨*varexpr*⟩ **'.'** ⟨*varidx*⟩ | ⟨*varidx*⟩
⟨*varidx*⟩ ::= ⟨*ID*⟩ **['['** arithexprs **']']**

As described by the above grammar, IR supports accesses to object fields via "`.`" operator and indexed associations using "`[]`".

Reply. Statement `reply` has almost the same format as `sendfrom` with an additional and implicit action: *index-forwarding*. We develop more on this property in Section 4.2.

⟨*reply*⟩ ::= **'reply'** ⟨*var*⟩ ⟨*var*⟩ ⟨*msgtype*⟩ ⟨*var*⟩*

Branching. Because state machines rely on message-passing semantics, the IR needs to provide intrinsic support for send and receive primitives with explicit communication-dependent control flow. Unconditional and conditional branches are supported by means of `goto` statement and receive primitives, `wait` and `waitin`. The conditional branches ⟨*wait*⟩ and ⟨*waitin*⟩ are

⟨*wait*⟩ ::= **'wait'** ⟨*var*⟩ [⟨*waitfor*⟩] ⟨*waitthen*⟩+
⟨*waitin*⟩ ::= **'wait'** ⟨*var*⟩ **'in'** ⟨*region*⟩+ [⟨*waitfor*⟩] ⟨*waitthen*⟩+

The ⟨*wait*⟩ clause requires an object to listen from, a set of accepted messages to wait for (⟨*waitfor*⟩) and a set of ⟨*waitthen*⟩ conditions (or branches) to be fulfilled in order to effectively branch. This statement summarizes the information found in trigger and guard expressions of the ⟨HOE⟩^{2} language for all outgoing transitions of a given state. The difference between ⟨*wait*⟩ and ⟨*waitin*⟩ is that ⟨*waitin*⟩ allows the specification of a list of parallel code blocks, *i.e.* the so called *regions* in the context of ⟨HOE⟩^{2}. A region is composed by a sequence of ⟨*fsmstmt*⟩ and, recursively, it may contain other regions.

After the object from which the ⟨*wait*⟩ statement is going to listen, we have a comma-separated list of message and/or sender object variables definitions.

⟨*waitfor*⟩ ::= **'for'** ⟨*recvexpr*⟩+
⟨*recvexpr*⟩ ::= ⟨*recvdef*⟩ **'='** **'recv'** **['['** ⟨*indexset*⟩ **']']** ⟨*msgtype*⟩
⟨*recvdef*⟩ ::= ⟨*var*⟩ | **'('** ⟨*var*⟩ **','** ⟨*var*⟩ **')'**

The receive expression defines a message variable (of type indicated by the rule ⟨*msgtype*⟩) and, optionally, the sender object (corresponding to the second variable if ⟨*recvdef*⟩ instantiates to a tuple) and index variables taken from the received message. Essentially, the receive expression provides a way to access all fields of the received message and bind them to local variables. Message variables have a struct-like type grouping all message parameters. All defined variables at receive expressions are considered to be unique.

After the set of accepted messages introduced by receive expressions, ⟨*waitthen*⟩ rule specifies a set of messages variables that will trigger the guard evaluation if they are present and where to branch to if guard evaluates to true.

⟨*waitthen*⟩ ::= **'then'** **['when'** ⟨*var*⟩+**]** **['if'** ⟨*guard*⟩**]** **'goto'** ⟨*ID*⟩

Regions. ⟨HOE⟩^{2} regions are implemented using the IR ⟨*waitin*⟩ statement. Naturally, in addition to traditional regions we support indexed regions.

⟨*waitin*⟩ ::= **'wait'** ⟨*var*⟩ **'in'** ⟨*region*⟩+ [⟨*waitfor*⟩] ⟨*waitthen*⟩+
⟨*region*⟩ ::= **['['** ⟨*indexset*⟩ **']']** **'{'** ⟨*fsmstmt*⟩+ **'}'**

4.2. Translating ⟨HOE⟩^{2}

Objects. ⟨HOE⟩^{2} objects together with its structural features are translated almost unchanged into object and/or scalar instances in the IR side. The IR follows the approach of object-oriented implementations in low-level languages such as C, *i.e.* it separates structural from behavioral code.

Creators. They are similar to constructors in traditional object-oriented languages and denote the same action, *object initialization*. IR translates these transitions into ⟨*creator*⟩ functions. For instance, Figure 7 describes the creator `RGB` of the object Pixel and its IR translation. Send and receive primitives are forbidden inside creators.

State Machines. The behavior specification in the form of graphical HSM is no more than a formal model for sequentially executable ⟨*fsmstmt*⟩ statements. Figure 8 shows a translation example of ⟨HOE⟩^{2} transition into IR code. The state machine of Image is named `ImageSM`. As described in the example, the transition is split into three IR statements: wait-branch, update and send.

Indexed Actions. Indexed actions and messages forms a key feature in ⟨HOE⟩^{2}. Figure 9 shows the main loop of the `Image` translation, which implements indexed send, update and receive actions. In Figure 9(b), the indexed version of **`sendfrom`** at line 2 creates indexed messages. The **`wait`** at line 5 contains an indexed receive and two possible branches. The indexed receive indicate which index values are considered valid receptions. The *when* branch triggers whenever message `m_recv` is

```
creator RGB(rgb: Int[3]) /
  r = rgb[0], g = rgb[1], b = rgb[2]
  to Init
```

(a) $\langle$HOE$\rangle^2$

```
creator Pixel.RGB(Int|3| rgb) {
  this = new Pixel;
  this.r = rgb[0],
  this.g = rgb[1],
  this.b = rgb[2];
  start PixelSM of this;
}
```

(b) IR

Fig. 7. Translation of creators

```
state GetY.
  on takeY(y: Float) / ychannel[0] = y : pixels[1].getY()
  to Next
```

(a) $\langle$HOE$\rangle^2$

```
GetY:
  wait this for m_recv = recv takeY<Float>
    then when m_recv goto UpdateY;
UpdateY:
  this.ychannel[0] = m_recv.y;
  sendfrom this this.pixels[1] getY<>;
  goto Next;
```

(b) IR

Fig. 8. Translation of $\langle$HOE$\rangle^2$ transitions

```
state GetY.     on /: { i: 0..pixels.len - 1 } pixels[i].getY()
                to GettingY
state GettingY. on takeY{i}(y: Float) / ychannel[i] = y to GettingY
                endon [i.all]
```

(a) $\langle$HOE$\rangle^2$

```
1  GetY:
2    forall[i: 0 <= i and i < 512] sendfrom[i] this this.pixels[i] getY<>;
3    goto GettingY;
4  GettingY:
5    wait this for m_recv = recv[i: 0 <= i and i < 512] takeY<Float>
6      then when m_recv   goto UpdateY
7      then if m_recv.all goto Final;
8  UpdateY:
9    this.ychannel[i] = m_recv.y;
10   goto GettingY;
11 Final:
```

(b) IR

Fig. 9. Translation of indexed actions

```
state ComputeY. on getY() / : r.mult(0.299), g.mult(0.587), b.mult(0.114)
                to Multing
[...]
state Adding. on added(result: Float) / y = result: initiator.takeY(y)
              to ComputeY
```

(a) $\langle$HOE$\rangle^2$

```
  wait this for (_m1_getY, Object src) = recv getY<>
    then when _m1_getY goto SBB_GET_Y;
[...]
UBB_ADDING:
  this.y = _m5_added.res;
  reply this src takeY<Float> this.y;
```

(b) IR

Fig. 10. Translation of initiator

present, while the *if* branch triggers if the guard is true, *i.e.*, once all messages of type `takeY<Float>` have been received.

Initiator. $\langle$HOE$\rangle^2$ language introduces the keyword *initiator* to denote replies. As shown in the example of Figure 10, replies are performed on objects variables binded at receives expressions. The reply has an important implication, *index-forwarding*. By tracking which receive has defined the sender object – and its corresponding message – the IR compiler automatically transfers index values from the incoming message into the new reply message.

```
cstate GettingY.
  region { i: 0..pixels.len - 1}
    [...]
  endregion
  on takeY(y: Float) to UpdateY
```

(a) $\langle \text{HOE} \rangle^2$

```
GettingY:
  wait this in [i: 0 <= i and i < 512] {
      [...]
  } for m_getY = recv takeY<Float>
    then when m_getY goto UpdateY;
```

(b) IR

Fig. 11. Translation of $\langle \text{HOE} \rangle^2$ regions

Regions. As pointed out early in this section, the hierarchy of HSM is defined through $\langle$*waitin*$\rangle$ statements. The equivalent IR expression of an indexed region is shown in Figure 11.

5. EFFICIENT CODE GENERATION FROM HIGH-LEVEL MODELS

Let us now outline the optimizing compilation strategy based on the IR and the informal translation from $\langle \text{HOE} \rangle^2$ presented in Section 4. The compiler takes advantage of the properties of $\langle \text{HOE} \rangle^2$, preserved at IR level, to optimize the model and produce efficient code patterns.

5.1. Challenges

In the context of optimizations for communicating automata, we face a an important difficulty concerning *data dependencies*. Efficient code generation of data-intensive applications calls for an accurate knowledge of data dependencies. However they are decoupled as asynchronous message exchanges between concurrent objects. That is, messages have a dual purpose, synchronisation and data carriers. The former imposes a precedence relation between computations of concurrent objects and the latter add a layer of input-output data relations on the precedence one. Ideally, we would like to find the data-flow of a dynamic network of communicating automata.[1] However, it is widely known that even in the case of static networks the problem is undecidable [Peng and Puroshothaman 1991]. Peng formulates the problem of communicating automata as a set of recurrence equations over the domain of infinite streams of messages. He shows that given two objects A and B we will not be able to link a certain computation outcome from A to its corresponding use in B. It is equivalent to say that the chain of data definitions and uses cannot be precisely determined, which is a major issue when looking for performance in computationally intensive applications based on state machines. On this context, we consider the hypothesis of *instantaneous reaction* to enable strong optimizations.

Instantaneous Reaction. We introduced in Section 2 the definition of the object interface. In addition to input and output messages accepted by the object from the user perspective, we can add a precedence relation between them. The precedence relation allows us to assume that an exported input-output message relation will hold under all program contexts. That is, it ensures that a given response will eventually come back. However, it does not specify precisely when. The handling of such a request may not be atomic and external objects may undertake other actions in the meantime. For instance, let A and B be two objects where A is the user of B, and B exposes the relation $m_1 \rightarrow m_2$, *i.e.* message m_2 will be sent as a response to the reception of m_1. Since the interface level exposes a transactional semantics, the compilation flow can be made modular and rely on the *instantaneous reaction* hypothesis. That is, among the possible orderings, one may safely assume that when object A sends message m_1 to B, B will handle it and send back the result according to its interface definition *at the same logical instant* from A's perspective (in the absence of deadlocks among internal transitions of B).

5.2. Optimizing Compiler for Communicating State Machines

The optimizing compiler chain is shown in Figure 12. The basic idea of the compilation flow is to transform the IR code such that it matches known and efficient code patterns on the target language.

[1] A model instance correspond to a network of connected objects.

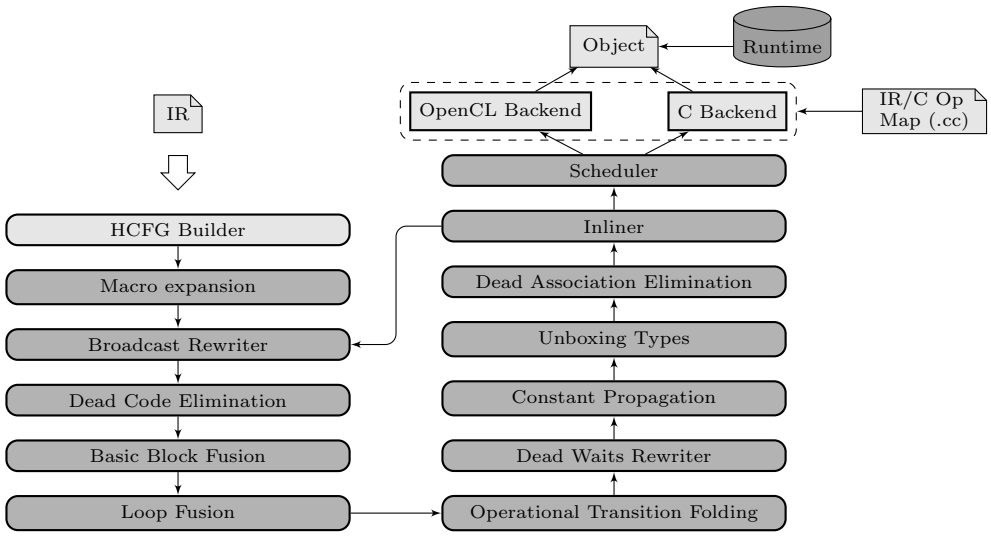


Fig. 12. Compilation flow

A mapping between IR operations (in the form of message exchanges) and target operations is provided separately. The backend is based on a runtime that implements the system semantics, *i.e.* communication primitives, object creation, state machine activation. It works on the intermediate representation presented in section 4. Internally, the compiler builds the Control-Flow Graph (CFG) of the state machine based on the branch kinds introduced in Section 4.1. Note that in contrast to other traditional branch schemes of imperative code compilers, the control flow cannot be reduced to two successors per basic block (see ⟨*wait*⟩ and ⟨*waitin*⟩ branches). All transformations are applied recursively on the state machine structure, *i.e.* across regions. Due to space constraints, we will only present the main idea of some of the analyses and transformation to illustrate the compilation process.

5.2.1. Broadcast Rewriter. This pass transforms what we called *wait-all* loops into indexed regions. Wait-all loops are loops that wait for all indexed messages under a certain domain and quit the main loop once they are all received. For instance, consider Listing 3, the translation of Listing 1, where its main waiting loop is shown hereafter

```
1 GettingY:
2   wait this for msg_takeY = recv[i: 0 <= i and i < 512] takeY<Float>
3     then when msg_takeY   goto UpdateY
4     then if msg_takeY.all goto Final;
```

The loop that covers basic blocks `GettingY` and `UpdateY` forms a wait-all loop. The wait construct at line 2, waits for all `takeY` messages on the specified range of index values. Given that there is no specific order of incoming messages, the compiler transforms such a loop into an indexed region. It results in the code of Listing 4 where the new statement is marked with an arrow.

```
 1   forall [i: 0 <= i and i < 512]
 2     sendfrom[i] this this.pixels[i] getY<>;
 3   goto GettingY;
 4 GettingY:
 5 ⇒wait this in [i: 0 <= i and i < 512] {
 6       wait this for msg_takeY = recv[i] takeY<Float>
 7         then when msg_takeY goto UpdateY;
 8     UpdateY:
 9       this.ychannel[i] = msg_takeY.y;
10   } then goto Final;
11 Final:
```

Listing 4. Indexed region from broadcast

5.2.2. Loop Fusion. The loop fusion moves indexed statements into indexed regions. For instance, the send statement at line 2 of Listing 4 is under the same index set as the indexed region at line 5.

```
GettingY:
  wait this in [i: 0 <= i and i < 512] {
⇒     sendfrom[i] this this.pixels[i] getY<>;
      wait this for msg_takeY = recv[i] takeY<Float>
        then when msg_takeY goto UpdateY;
    UpdateY:
      this.ychannel[i] = msg_takeY.y;
  } then goto Final;
Final:
```

The above listing shows the result of moving the send statement into the new indexed region. In general, index sets in $\langle \text{HOE} \rangle^2$ and IR can be precisely described as a (optionally labeled) parametric polyhedra and can be manipulated using an integer set library such as *isl* [Verdoolaege 2010]. Therefore, we can safely move a statement with index domain P_1 into an indexed region with domain P_2 iff $P_1 \subseteq P_2 \wedge P_2 \subseteq P_1$, where set inclusion and intersection are classical polyhedral operations.

5.2.3. Inlining. The compiler inlines two objects, inliner and inlinee, under the following conditions: (1) Inlinee object never escapes. (2) The state machine of the inlinee has a delimited transaction determined by its reachable states from receive to reply actions. (3) The inliner *completes* the transaction. For instance, consider the transaction of Pixel objects defined as `on getY() -> takeY(Float)`. The compiler computes the region of states covered by receptions of `getY` and `takeY` replies in order to inline it into Image, resulting in the following IR

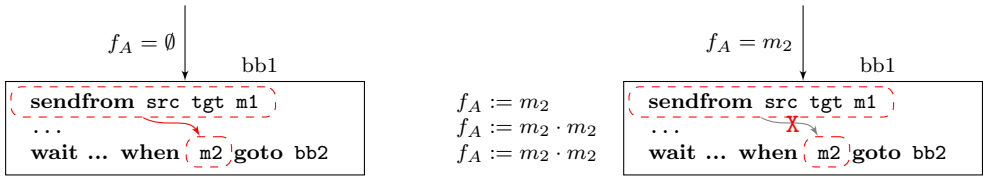
```
wait this in [i: 0 <= i and i < 512] {
⇒  sendfrom[i] this.pixels[i] this.pixels[i].r mult<Float> this.pixels[i].rcst;
⇒  wait this.pixels[i] for _m1_multed = recv multed<Float>
⇒    then when _m1_multed then goto UBB_MULTING;
⇒UBB_MULTING:
⇒  Float __new_var_1 = _m1_multed.val;
    [...]
} then goto Final;
```

5.2.4. Operational Transition Folding. Objects in $\langle \text{HOE} \rangle^2$ expose valid input and output messages as well as precedence relations between them via their interface. The transaction folding pass relies on such specification to build a set of foldable send and receive expressions inside the state machine.

Consider an object A communicating only with object S, where S defines a set of single input-single output operational transitions $\mathcal{T}_S = \{(m_i, n_i)\}$ such that message n_i is a response to m_i. Let $\overleftarrow{m_i^S} = n_i$ iff $(m_i, n_i) \in \mathcal{T}_S$ and $HCFG_A = (BB_A, E_A)$ the control flow of A's state machine. Figure 13 shows the FIFO state problem when trying to relate send and receive expressions. It describes the $HCFG_A$ of object A communicating with S where $(m_1, m_2) \in \mathcal{T}_S$. In 13(a), the FIFO of A is an empty list and we can safely assume that **`sendfrom`** `m1` is related to next **`wait`** expression according to $\mathcal{T}_S$. On the other hand, if the FIFO already contains a copy of `m2` generated by a response to a precedent **`sendfrom`**, the described one is not going to be related to next **`wait`** anymore (see Figure 13(b)). Therefore, whether **`sendfrom`** `m1` is related to next receive expression or not will depend on the incoming FIFO state.

In order to relate send and receive expressions, the Operational Transition Folding pass precisely track the FIFO state of two communicating objects. Given that it is an undecidable problem in general as explained earlier, it narrows the analysis to objects comunicating with other objects that specify precedence relations for all of its valid input messages under the hypothesis of instantaneous reactions.

The pass creates in-place operations yielding the final code shown below.



(a) Empty FIFO (b) Initialized Incoming FIFO

Fig. 13. Send-receive problem

```
wait this in [i: 0 <= i and i < 512] {
⇒Float __new_var_0 = applyon this.pixels[i].r multOp this.pixels[i].rcst;
  Float __new_var_1 = __new_var_0;
  [...]
} then goto FINAL;
```

We replace send and receive clauses by their corresponding operation. Additionally, removing sends and receives may lead to what we called *dead waits*. Dead waits are goto-like statements, *i.e.* they have only one transition without receive nor guard specifications and hence they can be safely replaced by goto statements. Finally, indexed regions without receive expressions can be easily interpreted as forall blocks and scheduled lexicographically resulting in an efficient for loop, as it will be shown in next section.

6. EXPERIMENTAL RESULTS

To provide a clear interpretation of the experimental results, let us first describe the underlying message-passing runtime and its interactions with $\langle \text{HOE} \rangle^2$ objects. We will then discuss the optimizations and code generated for a simple model, followed by a detailed analysis of a more complex example. In the process, we will consider multiple optimization levels to highlight the impact of the optimizing compiler, and we will illustrate it with a snippet of generated code.

6.1. Runtime and Structure of the Generated Code

Our toolchain emits one C file per $\langle \text{HOE} \rangle^2$ object and this file embodies the optimized code of this object. This C code can call back the runtime for a number of services, like creating or destroying (`hoe2_rt_run` and `hoe2_rt_done`), sending messages (`hoe2_rt_send`), waiting for incoming messages (`hoe2_rt_recv` and `hoe2_rt_recv_in`) and initialization and termination (`hoe2_rt_init` `hoe2_rt_finish`). The runtime starts the application by calling the `main` constructor of a root object defined by the programmer. This root object is in turn responsible for creating the initial objects and boostraping the application. Our runtime implementation is based on a widely available thread library called *QThreads* [Wheeler et al. 2008]. Each $\langle \text{HOE} \rangle^2$ object has its own userland thread and communicates through runtime callbacks. Currently, the C backend uses this implementation to manage a large number of concurrent objects: we have been able to successfuly run applications with up to 100k $\langle \text{HOE} \rangle^2$ objects on an Intel Core i5 (I5-4258U) with 8GB of RAM.

6.2. Experimenting the Flow Over Sample Code

We exercised the flow over an example previously discussed in Listings 1 and 2 for which the object model is shown at Figure 1. The former implements the object model of an image that creates RGB pixels and uses parallel sending actions to get its gray values. Underneath, the Pixel implements the actual gray conversion shown at Listing 2. Everything is done through message passing, including basic arithmetics for which one message initiates the computation and another returns the result once computed. From such implementation, the compiler outputs the C code of Listing 5.

```
1 aligned_t __obj_Image_ImageSM(struct Image *this) {
2         for (int i = 0; ((i >= 0) && (i < 512)); (i+=1)) {
```

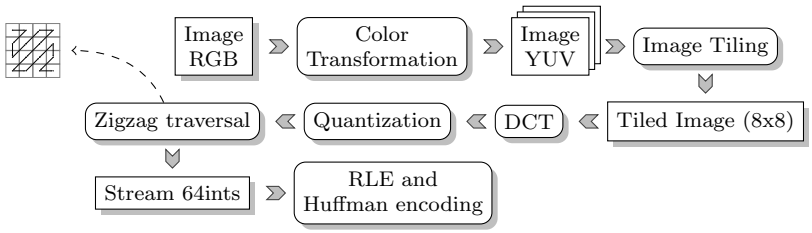


Fig. 14. Phases of the JPEG algorithm

```
3                float __new_var_8 = 0.114;
4                float __new_var_6 = 0.299;
5                float __new_var_7 = 0.587;
6                float __new_var_1 = this->pixels[i]->r->value * __new_var_6;
7                float __new_var_2 = this->pixels[i]->g->value * __new_var_7;
8                float __new_var_4 = this->pixels[i]->b->value * __new_var_8;
9                float __new_var_5 = __new_var_4 + __new_var_1;
10               float __new_var_3 = __new_var_2 + __new_var_5;
11               this->pixels[i]->y = __obj_new_Float_float(__new_var_3);
12               this->ychannel[i] = this->pixels[i]->y;
13       }
14       hoe2_object_done((struct hoe2_object *)this);
15       return 0;
16 }
```

Listing 5. Image object model: Generated C code

We can see how optimizations explained at section 5 are applied successively and successfully: *(a)* The broadcast sending of `getY` to all pixels has been translated to an indexed region—see optimizations described sections 5.2.1 and 5.2.2. *(b)* The `Pixel` 's state machine are inlined into each indexed region—See section 5.2.3. *(c)* The message-passing arithmetics are replaced by *in-place* operations thanks to interface definition of `Pixel , on getY() -> takeY(Float)` —See section 5.2.4. *(d)* Finally, the regions are translated into an efficient C for-loop.

Given that `Image` is the main object of our running program, its associations are preserved (considered as a side-effect action) and we found remaining "boxing" and "unboxing" operations of scalar objects at assignement and computation points of associations, respectively. We say that $\langle \text{HOE} \rangle^2$ scalars are *boxed* primitive types because they provide state machine semantics similar to other objects. For instance, Line 6 shows an unboxing operation of an $\langle \text{HOE} \rangle^2$ Int scalar—implemented as pointer accesses. Line 11 shows a boxing operation necessary to store values on Image associations. For performance reasons, using unboxed values over boxed ones is always preferred. The optimization pass for automatic unboxing of $\langle \text{HOE} \rangle^2$ scalars is not shown in this paper.

6.3. Metrics and Results

The semantics of our language is based on message passing, even for arithmetics operations. Our tool flow aims at optimizing-out as much messages as can be: If we acheive deep inlining of objects we should expect a reduction on sent and received messages, hence a reduction of application/runtime communications that slow down the $\langle \text{HOE} \rangle^2$ application. In order to show the impact of our optimizing chain, we define three transformation levels: (0) No optimization, (1) Operational transition folding and indexed region generation and (2) Fully optimizing with inlining. Note that unoptimized code generation give us good insights on the operations involved into the application and help us to quantify them. Further optimization levels aim for efficient code generation.

To stress out the toolchain under these optimization levels, we modeled a chain of image transformations taken from the JPEG algorithm shown in Figure 14. The model of an image of 64 by 64 pixels is presented in Figure 15. After converting RGB pixels to its luminance component, the image needs to be tiled to form blocks of 8x8 luminance values. The Discrete Co-

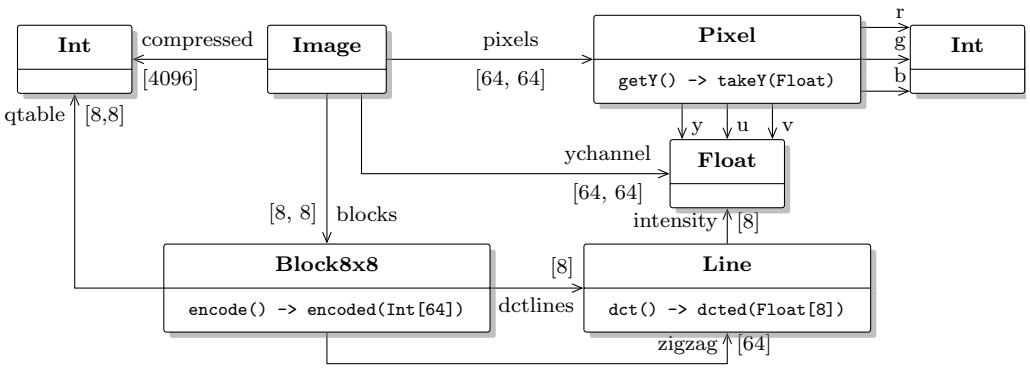


Fig. 15. Image object model implementing JPEG till zigzag traversal

sine Transform (DCT) is applied in parallel to all blocks following our broadcasting semantics.

```
state Encode. on [] / {i, j: 0 <= i < 4 and 0 <= j < 4}
                      blocks[i, j] = new Block8x8(ychannel[8*i..8*i+7, 8*j..8*j+7])
                    : {i, j: 0 <= i < 8 and 0 <= j < 8}
                      blocks[i, j].encode()
               to Encoding
```

We show in the above listing the tiling update and the sending action of message `encode` to all blocks. The `Block8x8` object does such computation as a composition of two 8-point 1D DCT, as described in [Loeffler et al. 1989].[2] This computational composition is represented at the model by the structural composition of `Block8x8` and `Line` objects. `Line` performs the 1D DCT and send its result back to `Block8x8`. The computation is triggered by a broadcast from `Block8x8` to all its `Line` objects.

```
state DCT. on / : {i: 0 <= i < 8} dctlines[i].dct1D() to DCTing
```

Once the DCT is finished, `Block8x8` divides all the result values by the quantization table `qtable` as follows

```
state Quantize. on / : {i, j: 0 <= i < 8 and 0 <= j < 8}
                       dctblock[i, j].div(qtable[i, j]) to Zigzagging
```

At the reception of all divided values, we perform the zigzag traversal to create a stream of 64 values, which are stored in `zigzag`, following the index values of received messages. Listing 6 shows its implementation (see Figure 14).

```
state Zigzagging. on dived{i, j}(v: Float) [(i + j) % 2 = 0 and i + j <= 7] /
                     zigzag[(i + j + 1) * (i + j) / 2 + j] = v
                  to Zigzagging
                  on dived{i, j}(v: Float) [(i + j) % 2 = 1 and i + j <= 7] /
                     zigzag[(i + j + 1) * (i + j) / 2 + i] = v
                  to Zigzagging
                  on dived{i, j}(v: Float) [(i + j) % 2 = 0 and i + j  > 7] /
                     zigzag[56 - (15 - i - j) * (14 - i - j) / 2 + j] = v
                  to Zigzagging
                  on dived{i, j}(v: Float) [(i + j) % 2 = 0 and i + j  > 7] /
                     zigzag[56 - (15 - i - j) * (14 - i - j) / 2 + i] = v
                  to Zigzagging
```

Listing 6. Zigzag traversal state

For this extensive implementation, the number of exchanged messages at optimization levels O0 and O1 are shown in Figure 16 (logarithmic scale). Level O0 generates naive code, preserving all

[2]The most used implementation among JPEG encoders.

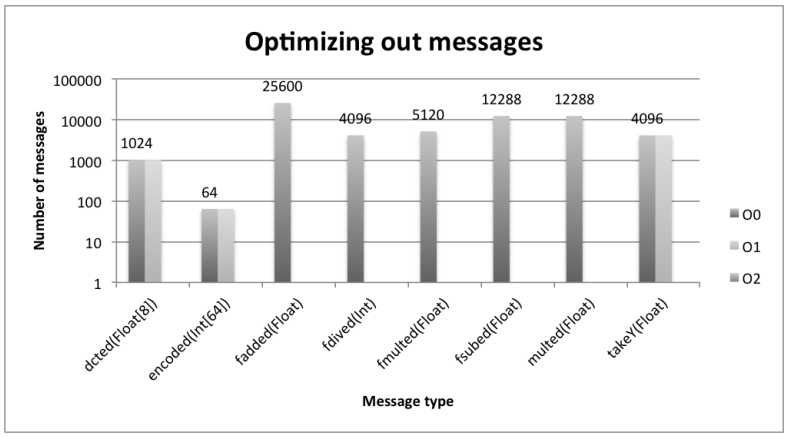


Fig. 16. Number of messages at optimization levels O0 and O1 (O2 eliminates all messages)

message exchanges. At O1, the specialization of indexed regions and folding messages into in-place operations eliminates—in this particular case—all scalar messages. Composite messages such as `dcted(Float[8])`, `encoded(Int[64])` and `takeY(Float)` are still present; missing O1 bars indicate that all messages have been eliminated. At level O2, the most agressive optimization level, deep inlining of objects in the model yields a communication-free implementation. For instance, our compiler produces the following optimized C code from the modeled quantization and zigzag storage:

```
for (int a = 0; (a <= 7); (a += 1)) {
  for (int b = 0; (b <= 7); (b += 1)) {
    ...
    for (int i = 0; (i <= 6); (i += 1)) {
      for (int j = 0; (j <= (6 - i)); (j += 1)) {
        if ((((-(i) + j) % 2) == 0)) {
          NEW_LABEL_19_56:;
          int __new_var_8 =
            dctblock[i][j]->value / this->blocks[a][b]->quant[i][j]->value;
          zigzag[(((((i + j) + 1) * (i + j)) / 2) + j)] =
            __obj_new_Int_int(__new_var_8);
}}}}}
```

The two top loops correspond to the iteration domain of blocks, while the two more nested ones come from the inlining transformation together with the intersection of `zigzag` domain and one of the guard conditions. The compiler uses polyhedral code generation to produce C for-loops from our index domains [Bastoul 2004].

7. RELATED WORK

The contributions presented in this paper cover two axes of research on the domains of MDE and HSM (a) scalar data representations for parallel computations and (b) compilation of communicating state machines. Scalar data on state-of-the-art modeling languages are generally decoupled from generic objects and frequently unrelated to the guest action language. The UML proposes scalars as *primitive types* without defining any specific operation on them, even though it loosely indicates that they may have an associated algebra. UML-based approaches such as *SysML* [Object Management Group 2012] and *MARTE* [Object Management Group 2009] define new primitive data types for specific application domains (real-time). However, their work is not focused on data-parallel operations for the modeling of data-intensive applications.

The *Gaspard2* modeling framework is the closer related work we found in the litterature [Gamatié et al. 2011]. They propose a combination of MARTE for the modeling of embedded systems and ArrayOL [Boulet 2007], which offers efficient code generation for data-intensive applications. They

showed interesting results of an "unified" modeling of modern platforms (GPGPU) and video processing algorithms (H-263) [De Oliveira Rodrigues et al. 2011]. However, theirs results focused on the data-driven part of the algorithm (filtering) because ArrayOL does not support control dependent flow. Therefore, they have to mix different formalisms for control and data-driven components. Although modern applications are composed by a complex interaction of such components.

Current research on action languages, seems to not take advantage of research work on parallel languages. We found parallel languages, or extensions, such as Data-Parallel Haskell [Chakravarty et al. 2007], the Hierarchical-Tiled Array model [Bikshandi et al. 2006], HiDP [Mueller and Zhang 2013], Sequoia [Fatahalian et al. 2006], among many others, where concepts of "data-parallelism" and hierarhical decomposition of data/computation are of main concern. Indeed, the structure of models in MDE exposes explicitly the hierarchy of the modeled application and parallel operations on collections of scalar data are fundamental. Most research work on action languages are mainly focused on model executability and/or verification. Jumbala [Dubrovin 2006] proposes Java-like actions based on UML models. They assumed that parallelism should not be modeled at the action level, as it should be shifted to the parallel constructions of the state machine model. However, we have seen that data-parallelism is required even at the scalar level for data-intensive applications.

Another active research direction when dealing with models is the optimized compilation of state machines. Asma Charfi *et al.* [Charfi et al. 2012] raised the recurrent problem of modeling frameworks concerning the gap between models and code production. Existing modeling frameworks together with a corresponding action language enables the production of executable models while providing validation and verification support. However, the early validation process and consequently the modeling effort are invalidated by hand-tuned code specialization, required to meet performance requirements of the given models. They explored a compiler extension for the compilation of UML models with state machines. Using a new representation called *GUML*, information concerning the structure of the original state machine is passed to the C compiler. It enables high-level optimizations, such as unreachable state elimination, in the intermediate representation of a compiler. They presented encouraging results with respect to code size. In [Schattkowsky and Muller 2005], the authors propose a set of rewriting rules for a subset of the UML state machines definition that they called "Executable State Machines" (ESM). In constrast to the precedent work, it is independent of the action language and they mostly handle structural optimizations, *e.g.* move entry/exit activities to input/output transitions, resolve conflicting triggers along the hierarchy, among others. Even though they propose a set of state machine rewritting rules in a very generic manner, they lack of a concrete action language and cannot produce completely executable models. They also do not address efficient code generation issues.

8. CONCLUSION

We defined two extensions to an existing a parallel action language, $\langle \text{HOE} \rangle^2$: generalized scalars and indexed regions. The former allow us to model immutable objects and their operations as message passing: a single model is amenable to fine-grain simulations as well as static optimizing compilation to sequential in-place operations, inlining and mapping to concrete target operators. The latter, indexed regions, are a natural extension to support data-parallel forall blocks over a given indexing domain, while enabling the generation of efficient target code patterns for a variety of hardware platforms. Both extensions are defined formally, and supported by an intermediate representation (IR) for code generation and optimization purposes.

The code generation from high-level models based on communicating and HSM becomes a complex task when performance matters and when targeting parallel hardware platforms. Most tools concentrate theirs efforts on model expressiveness and simulation. They frequently fill the gap between modeling and code generation by starting from the bare model to naive, hence inneficient, object-oriented implementations. Our intermediate representation is inspired from the object-oriented extensions of procedural imperative languages. The IR exposes the HSM as a control flow graph of basic blocks with communication-dependent branches. Introducing additional restrictions on index sets, namely affine constraints, we presented a compilation framework for analysis and

optimization of communicating state machines. The combination of a number of new concepts such as indexed messages, indexed regions, parallel and indexed send and receives actions as well as updates, the object interface specification and, finally, $\langle \text{HOE} \rangle^2$ scalars, conveniently extends the expressiveness of hierarchical state machines. We illustrated the translation from the $\langle \text{HOE} \rangle^2$ modeling language to the IR on concrete and realistic examples. Our optimization flow handles all this information to perform important optimizations: broadcast rewritting into indexed regions, loop fusion, folding of messages, and object inlining. As a consequence, the entry model is transformed gradually to match efficient code patterns available at the target language.

REFERENCES

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Cedric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, Washington, DC, USA, 7–16.

Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. 2006. Programming for Parallelism and Locality with Hierarchically Tiled Arrays. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*. ACM, New York, NY, USA, 48–57.

Pierre Boulet. 2007. *Array-OL Revisited, Multidimensional Intensive Signal Processing Specification*. Rapport de recherche RR-6113. INRIA. 24 pages.

Amy Brown and Greg Wilson. 2012. *The Architecture Of Open Source Applications*. Vol. II. lulu.com. 432+ pages.

Egon Brger, Alessandra Cavarra, and Elvinia Riccobene. 2004. On formalizing UML state machines using ASMs. *Information and Software Technology* 46, 5 (2004), 287 – 292.

P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. 1987. LUSTRE: A Declarative Language for Real-time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. ACM, New York, NY, USA, 178–188.

Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data Parallel Haskell: A Status Report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP '07)*. ACM, New York, NY, USA, 10–18.

Barbara Chapman, Gabriele Jost, and Ruud van der Pas. 2007. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.

A. Charfi, C. Mraidha, and P. Boulet. 2012. An Optimized Compilation of UML State Machines. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on*. 172–179.

Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'H, and Jean-Luc Dekeyser. 2011. Programming Massively Parallel Architectures using MARTE: a Case Study. In *2nd Workshop on Model Based Engineering for Embedded Systems Design (M-BED 2011) on Date Conference 2011*. Grenoble, France.

Jori Dubrovin. 2006. *JUMBALA: AN ACTION LANGUAGE FOR UML STATE MACHINES*. Technical Report HUT-TCS-A101.

Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. ACM, New York, NY, USA, Article 83.

Abdoulaye Gamatié. 2009. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification* (1st ed.). Springer Publishing Company, Incorpo-

rated.

Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. 2011. A Model-Driven Design Framework for Massively Parallel Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 10, 4, Article 39 (Nov. 2011), 36 pages.

Eran Gery, David Harel, and Eldad Palachi. 2002. Rhapsody: A Complete Life-Cycle Model-Based Development System. In *Integrated Formal Methods*, Michael Butler, Luigia Petre, and Kaisa Sere (Eds.). Lecture Notes in Computer Science, Vol. 2335. Springer Berlin Heidelberg, 1–10.

David Harel. 1987. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 3 (1987), 231–274.

ThomasA. Henzinger and Joseph Sifakis. 2006. The Embedded Systems Design Challenge. In *FM 2006: Formal Methods*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Lecture Notes in Computer Science, Vol. 4085. Springer Berlin Heidelberg, 1–15.

Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa, and Stéphane Malfoy. 2012. Efficient Embedded System Development: A Workbench for an Integrated Methodology. In *ERTS^2 2012* (2012-02-01). Toulouse, France.

IEEE754 2008. IEEE Standard for Floating-Point Arithmetic. (Aug 2008).

Gilles Kahn. 1974. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*. 471–475.

Shuang Liu, Yang Liu, tienne Andr, Christine Choppy, Jun Sun, Bimlesh Wadhwa, and JinSong Dong. 2013. A Formal Semantics for Complete UML State Machines with Communications. In *Integrated Formal Methods*, EinarBroch Johnsen and Luigia Petre (Eds.). Lecture Notes in Computer Science, Vol. 7940. Springer, 331–346.

Ivan Llopard, Albert Cohen, Christian Fabre, and Nicolas Hili. 2014. A Parallel Action Language for Embedded Applications and Its Compilation Flow. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems (SCOPES '14)*. ACM, New York, NY, USA, 118–127.

C. Loeffler, A. Ligtenberg, and George S. Moschytz. 1989. Practical fast 1-D DCT algorithms with 11 multiplications. In *Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on*. 988–991 vol.2.

Erich Mikk, Yassine Lakhnech, Carsta Petersohn, and Michael Siegel. 1997. On Formal Semantics of Statecharts As Supported by STATEMATE. In *Proceedings of the 2Nd BCS-FACS Conference on Northern Formal Methods (2FACS'97)*. British Computer Society, Swinton, UK, UK, 12–12. http://dl.acm.org/citation.cfm?id=2227850.2227862

Frank Mueller and Yongpeng Zhang. 2013. Hidp: A Hierarchical Data Parallel Language. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–11.

Object Management Group. 2009. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. (2009).

Object Management Group. 2011. *OMG Unified Modeling Language, Superstructure, v2.4.1*. Technical Report. OMG.

Object Management Group. 2012. OMG Systems Modeling Language (OMG SysML), Version 1.3. (2012).

Wuxu Peng and S. Puroshothaman. 1991. Data Flow Analysis of Communicating Finite State Machines. *ACM Trans. Program. Lang. Syst.* 13, 3 (July 1991), 399–442.

Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebr. Program.* 60-61 (2004), 17–139.

John C. Reynolds. 1999. *Theories of Programming Languages*. Cambridge University Press, New York, NY, USA.

T. Schattkowsky and W. Muller. 2005. Transformation of UML State Machines for Direct Execution. In *IEEE Symposium on Visual Languages and Human-Centric Computing*. 117–124.

Dana Scott. 1971. *Toward a Mathematical Semantics for Computer Languages*. Technical Report PRG06. OUCL. 49 pages.

Dirk Seifert. 2008. *An Executable Formal Semantics for a UML State Machine Kernel Considering Complex Structured Data*. Rapport de recherche. 21 pages.

Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *Lecture Notes in Computer Science, International Congress on Mathematical Software (ICMS 2010), Kobe, Japan, 13-17 September 2010*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer, 299–302.

K.B. Wheeler, R.C. Murphy, and D. Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. 1–8.