Programming embedded manycore : refinement and optimizing compilation of a parallel action language for hierarchical state machines

Ivan Llopard

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École Doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Ivan LLOPARD

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Programming Embedded Manycore : Refinement and Optimizing Compilation of a Parallel Action Language for Hierarchical State Machines

soutenue le 26 avril, 2016

devant le jury composé de :

M. Reinhard von Hanxleden Professeur à l'Université de Kiel, Allemagne	Rapporteur
M. Saddek Bensalem Professeur à l'Université Joseph Fourier, Grenoble	Rapporteur
M. Robert de Simone Directeur de recherche, INRIA Sophia-Antipolis, Valbonne	Examinateur
Mme Sophie Dupuy-Chessa Professeur à l'Université Pierre Mendès France, Grenoble	Examinateur
M. Lionel Lacassagne Professeur à l'Université Pierre et Marie Curie, Paris	Examinateur
M. Eric Lenormand Thales Group R&T, Palaiseau	Examinateur
M. Albert Cohen Directeur de recherche, INRIA/ENS, Paris	Directeur de thèse
M. Christian Fabre Ingénieur Chercheur, CEA LETI, Grenoble	Co-encadrant

Remerciements

Je remercie M. Reinhard VON HANXLEDEN, M. Saddek BENSALEM, M. Robert DE SIMONE, Mme Sophie DUPUY-CHESSA, M. Lionel LACASSAGNE et M. Eric LENORMAND, qui m'ont fait l'honneur et le plaisir de participer à mon jury de thèse.

J'adresse mes plus grands remerciements à mon directeur de thèse Albert COHEN et mon co-encadrant Christian FABRE. Ils ont su me guider et m'encourager avec beaucoup de patience et de sagesse tout au long de mon parcours de thèse. J'ai beaucoup grandi à vos côtés, humainement et scientifiquement, et je vous en suis très reconnaissant.

Un grand merci à toute l'équipe LIALP du CEA Grenoble. Une équipe dynamique et chaleureuse, au sein de laquelle il fait toujours plaisir d'y être. Je remercie tout particulièrement mes anciens collègues de bureau, Nicolas HILI et Victor LOMÜLLER. J'ai partagé beaucoup de bons moments avec eux durant ce parcours, dans le bureau comme dans le bar !

Le début de ce chemin commence sans doute bien plus tôt, vers l'année 2010. A son origine Jérome MARTIN, merci pour ton premier bac à octets. Un merci particulier au *haskeller* du LIALP, Damien COUROUSSÉ. Des discussions toujours intéressantes et bien monadiques, avec comme idée de bord, les types boxés. Merci pour la bonne humeur des personnes qui resteront toujours dans mon esprit et que j'espère revoir un jour: Diego PUSCHINI, Julien MOTTIN, Thierno BARRY, Suzanne LESECQ, Fayçal BENAZIZ, Henri-Pierre CHARLES, Vincent OLIVE, Anca MOLNOS, Frédéric HEITZMANN, Maxime LOUVEL, Yeter AKGUL, Lionel VINCENT, Olivier DEBICKI, Olesia MOKRENKO.

Je tiens également à remercier toute l'équipe PARKAS (INRIA) pour leur accueil très sympatique pendant mon bref passage de deux semaines. Je remercie en particulier Marc POUZET, Nhat MINH LÊ, Adrian GUATTO, Tobias GROSSER et Riyadh BAGHDADI.

Mon principal et plus fort soutien, ma future épouse Léa HUMBERT. Une bonne partie de cette thèse est à toi. Tu as été d'une patience inouïe et d'une générosité remarquable. Merci pour avoir consacré trois années de ta vie autour de ce projet de thèse. Enfin, cette histoire s'achève avec une petite personne qui nous a rejoint pour terminer ce trajet à trois, mon fils Gaspar.

UNIVERSITÉ PIERRE ET MARIE CURIE

Résumé

École Doctorale d'Informatique, Télécommunications et Électronique (EDITE)

Thèse de doctorat

Programmation de systèmes embarqués many-core : Raffinement et compilation optimisante d'un langage d'action parallèle pour machines à états hiérarchiques

par Ivan LLOPARD

Afin de gérer la complexité des systèmes embarqués modernes, les langages de modélisation proposent des abstractions et des transformations adaptées au domaine. Basées sur le formalisme de machines à états hiérarchiques, connu sous le nom de *Statecharts*, ils permettent la modélisation du contrôle parallèle hiérarchique. Cependant, ils doivent faire à deux défis majeures quant il s'agit de la modélisation des applications à calcul intensif: le besoin des méthodes unifiées supportant des actions avec parallélisme de donnée; flots d'optimisation et génération de code à partir des modèles trop généralistes.

Dans cette thèse, nous proposons un langage de modélisation étendu avec une sémantique d'actions parallèles et machines à états hiérarchiques indexées, spécialement adapté pour les applications à calcul intensif. Avec sa sémantique formelle, nous présentons un flot de compilation optimisante pour le raffinement des modèles en vue d'une génération du code efficace avec parallèlisme de donnée.

PIERRE ET MARIE CURIE UNIVERSITY

Abstract

Doctoral School of Informatics, Telecommunications and Electronics (Paris)
Specialization: Informatics

Doctor of Philosophy

Programming Embedded Manycore: Refinement and Optimizing Compilation of a Parallel Action Language for Hierarchical State Machines

by Ivan LLOPARD

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.

In this thesis, 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.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

We present the context for this work, our main motivations and goals, and we conclude by giving a brief overview of the thesis organization.

1.1 Context

Modern embedded systems integrate more sophisticated functionalities to meet hard performance requirements of today's applications. This integration power enabled the implementation of complete systems into a single chip, namely System on Chip (SoC). The step forward included many core architectures providing massively parallel computation units. The inherent parallelism of these architectures makes them suitable for applications showing, preferable in the most direct manner, parallel components. In particular, data-parallelism is essential to scalable and efficient parallel programming [2–4, 9]. Although, industrial programming practices use low-level languages to keep a precise control over the platform [10, 11], and thus application developers need a deep knowledge of hardware features. On the other hand, adaptations of general purpose languages to address parallel issues cannot cope with the great number of platform-dependent tuning parameters at once. Therefore, code generation from such languages tends to be a challenging work. Both worlds need to be taken into account incrementally to get best performances.

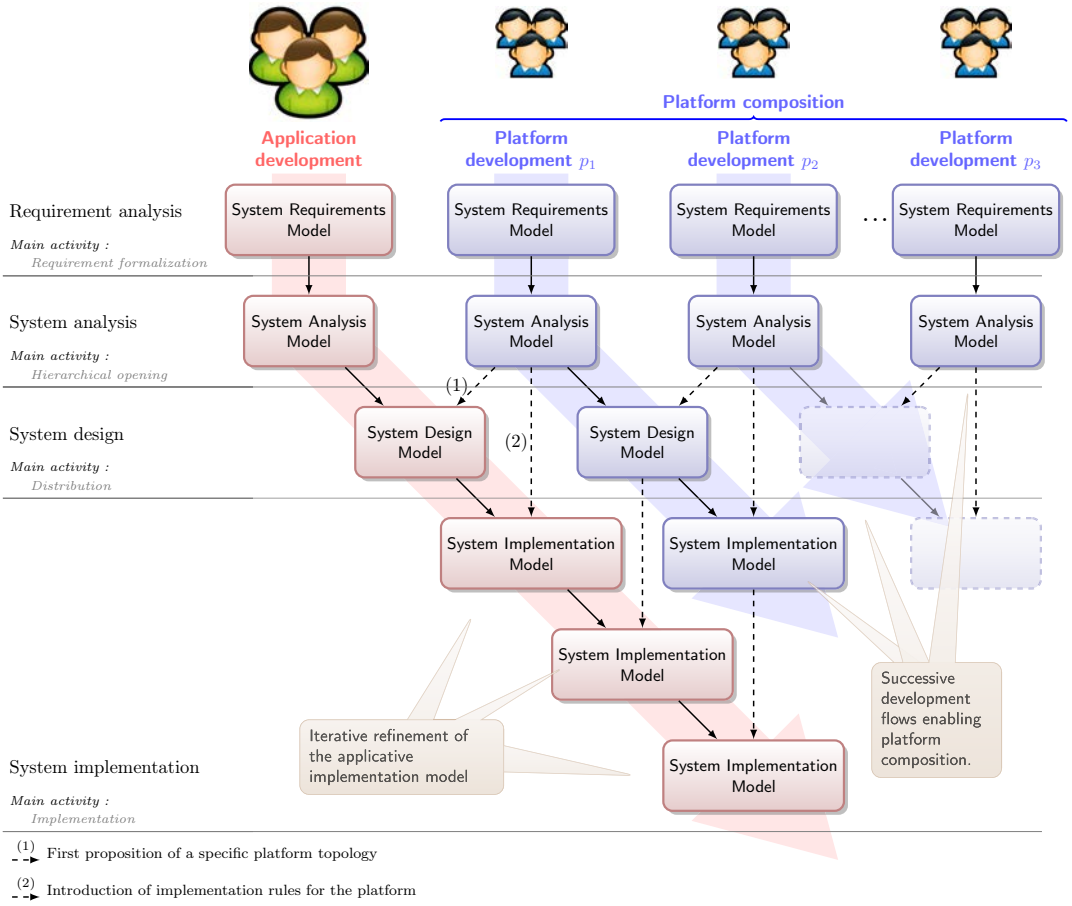


FIGURE 1.1: The $\langle$HOE$\rangle^2$ methodology for the development of embedded systems

As a result, the design and development of embedded systems is becoming increasingly complex [12]. To cope with the complexity, designers rely on multiple design methodologies and its supporting tools [13]. Recent trends push for unified model-based approaches that allow the modeling of both software and hardware [14–18]. The role of model-based techniques is to provide strong abstractions to handle in a modular manner a wide variety of system configurations. The abstraction are iteratively refined in order to get closer to the final implementation taking into account different platform constraints step by step.

Figure 1.1 shows the Highly Heterogeneous, Object-Oriented, Efficient Engineering $\langle$HOE$\rangle^2$ methodology, which is thoroughly covered in [19]. $\langle$HOE$\rangle^2$ and other similar frameworks specify the system behavior using *Statecharts*, or Hierarchical State Machines (HSMs).

Statecharts were first introduced by Harel for the specification and design of complex discrete-event systems [20]. It is a graphical language providing natural and easy-to-grasp abstractions to express parallelism, hierarchy and compositions of state machines. Figure 1.2 shows the Statechart model of a car audio system. Modern modeling languages such as Unified Modeling Language (UML) incorporate Statecharts as their main formalism to model object behavior, extending them with object-oriented traits [21].

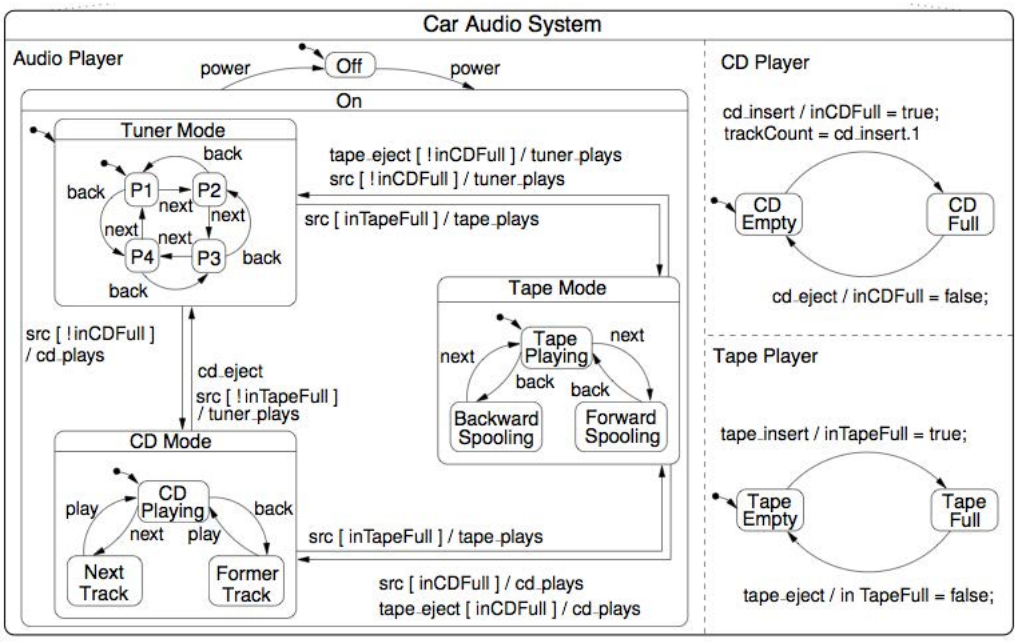


FIGURE 1.2: Modeling a car audio system with Statecharts (Source: [1])

1.2 Motivation

The motivation of this thesis is split on three complementary subjects: modeling of parallel computations, semantics of Statecharts and its exisiting compilation challenges.

Data Parallelism and Statecharts. If Statecharts are well-suited for the modeling of control decisions of cooperative components of a system, data-driven computations are hardly expressable in the formalism. Though modern embedded systems provides more and more parallelism at different granularity levels (data, thread, system), frequently presented in a hierarchical manner. On the other hand, hierarchy is one of the main properties of model-driven approaches to provide highly modular representations in addition to rigorous methodologies. A similar issue has been pointed out by Gamatié et al. where model-driven techniques are combined with a domain specific language to address the problem [8].

Semantics of Statecharts. The widely used modeling languages in the Model-Driven Engineering (MDE) community is UML [21]. UML is mainly focused on modeling issues and, in favor of generality, let many semantical points of Statecharts open to specific implementations. As a consequence, many research works propose formalizations of its rather imprecise informal semantics [1, 22–25]. Indeed, the complete formalization of UML Statecharts is a challenging work given the great number of proposed features and the already known non-constructive

properties of Statecharts [26]. A Statechart-based language should provide a well-founded semantics trying to avoid previous semantical complications in order to achieve a clear, scalable and customizable semantic implementation.

Compilation of Statecharts. The purpose of models is simulation or as a path to rapid implementations of software architectures. Even though Model-Driven Architecture (MDA) approaches push the idea of an iterative integration of hardware constraints into the initial parallel application (or model because of the unified view), the model is not subject to any kind of optimization and the code generation process is known for its faithfulness with respect to the input model. Recent research efforts focus on an executable and performance-aware compilation flow from models. The goal is to apply compilation techniques at the model level itself, by considering intermediate representations if necessary, in order to achieve high-performance and platform-specific implementations [6, 7, 27, 28].

1.3 Objectives and Contributions

In this thesis, I propose an adapation of the already known Statechart formalism to the modeling of parallel computations. We propose a parallel *action language* and several key extensions to the formalism, which are compliant with plain objects.

Together with a new language, we introduce its semantics in a hierarchical manner. Compared to existing approaches, it allow us to some extent to separate semantical points in multiple layers while following the language structure in an operational way.

In order to produce efficient code from Statecharts, we introduce a new intermediate representation, which is more expressive than existing representations for Statecharts. We include object creation, struct-like variable accesses and indexing of associations while preserving most of the information coming from the front-end language. It allow us to reason about communication issues, inlining and unreachable states, among many other factors of performance.

Finally, we contribute a new compilation flow for HSMs closely following the data-parallel extensions of the input language, which are preserved at the intermediate representation. As mentioned earlier, the intermediate representation allow us to reason about particular semantical implications concerning communication, inlining, data dependences, loops and parallelization issues.

1.4 Thesis Organization

The thesis organization is as follows.

Chapter 2, 3, 4: State-of-the-art. We separate the state-of-the-art into three related chapters. Concerning our main motivation, we look in Chapter 2 for inspiration into existing parallel languages. Going from very high-level languages to low-level ones, we show how the parallelism is presented to the programmer. Given that our approach is based on Statecharts, we review in Chapter 3 semantics of Statecharts to define the foundations of the language presented in this thesis. Naturally, we continue in Chapter 4 with the survey of related research work on efficient compilation of Statecharts.

Chapter 5: The $\langle$HOE$\rangle^2$ Language. Our proposed extension to Statecharts. It is a complete language called $\langle$HOE$\rangle^2$, as our proposition fits into a more wide context of a model-driven methodology also named $\langle$HOE$\rangle^2$.

Chapter 6: The $\langle$HOE$\rangle^2$ Language Semantics. We present a hierarchical approach for the formalization of our Statechart-based language.

Chapter 7: An Intermediate Representation. We introduce a new intermediate representation as a layer for the optimization and compilation of Statecharts.

Chapter 8: The Compiler. Upon the previous intermediate representation and many ideas of the $\langle$HOE$\rangle^2$ language, we build the optimizing compiler. We present static analyses and optimizations that apply in the context of communicating objects and allows us to produce efficient code.

Chapter 9: Results. Finally, we prove that a generation of efficient code from our parallel extensions of Statecharts is possible. We stress the compiler with more complex models and show how it can greatly help the designer to achieve automatic model optimizations in the context of the $\langle$HOE$\rangle^2$ methodology.

Part I

State of the Art

Chapter 2

Hierarchical Data Representations for Parallel Applications

Contents

We review in this chapter parallel programming techniques in a more general context than Model-Driven Engineering (MDE) and its Statecharts.

2.1 The Data-Parallel Paradigm

Different data and computational abstractions are used for parallel programming. The *data-parallel* paradigm is heavily used across many approaches [29]. In this paradigm, the data set is considered to be divided into regular blocks with different granularities on which common operations have parallel semantics. Therefore, operations have a degree of parallelism which depends on the data abstraction. Distributed computing systems pushed the evolution of existing languages looking for support of data-parallel operations over possibly distributed data [30, 31].

New custom directives were introduced for the specification of data distributions, as well as the underlying computing grid in some cases, in order to specify solutions for a complex problem: " mapping" or adaptation of parallel applications over a parallel architecture.

2.1.1 The Functional Style

Nested Data Parallel (NDP) is a key extension to the data-parallel paradigm. Originally proposed by Blelloch, it allows the programmer to hierarchically compose distinct data-parallel operations [32]. Among the most abstract implementations of NDP, we found two approaches based on functional languages: NESL [32] and Haskell Data-Parallel (HDP) [9].

The current implementations integrate this paradigm intuitively into the type system. In HDP, we find a new native type called *parallel array*, written as `[:a:]`. It allows the programmer to build hierarchical parallel expressions based on the idea that a parallel array is polymorphic and may be composed by itself. For instance, the following types denote parallel arrays of floating point values, `Vector`, and tuples of integer and floating point values, `SparseVector`.

```
type Vector       = [: Float :]
type SparseVector = [: (Int, Float) :]
```

The construction of values with the above types is built in parallel. The listing below shows, using list comprehensions, the multiplication of a sparse vector and a full one.

```
dotp :: SparseVector -> Vector -> Float
dotp sv v = sumP [: x * (v!:i) | (i, x) <- sv :]
```

The multiplication accesses each element of the sparse vector, `(i, x)`, and multiply it by the correct scalar position in the non-sparsed vector, `x * (v`:i)!. `sumP` folds the, possibly distributed, parallel array. In HDP, multiplications are grouped and performed by different *gangs* of execution threads.

Following this approach we can express nested parallel operations in an architecture-independent way which is relatively simple to understand with a concise complexity model [33]. For instance, the listing below builds another parallel array based on the parallel definition of `dotp`.

```
type SparseMatrix = [: SparseVector :]

smvm :: SparseMatrix -> Vector -> Vector
smvm sm v = [: dotp row v | row <- sm :]
```

On the other hand, the approach relies on complex compiler techniques to optimize all performance factors in order to produce efficient code under a fixed execution model (gang of threads). The code generation flow strictly based on the functional approach let the application

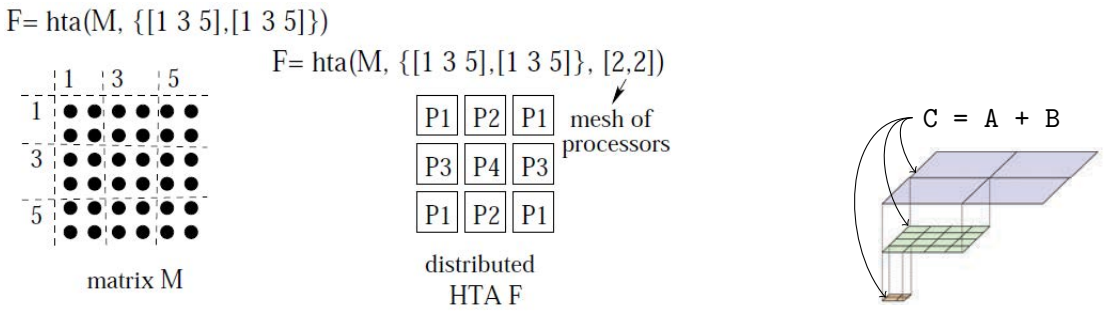


FIGURE 2.1: HTA type partitioning and operation mapping (Source [2])

developer focus on algorithmic issues while the architecture constraints are automatically taken into account. They consider a distributed memory. Nevertheless, the variety of embedded multi-core architectures make the automatic pass from the HDP to code generation an ambitious and complex task. It is mainly described by rewriting rules with the introduction of new types to represent data distribution [9].

2.1.2 HTA: Hierarchically Tiled Arrays

In the class of imperative high-level parallel programming, we have the Hierarchically Tiled Arrays (HTA) approach proposed in [2]. Similar to HDP, parallelism is introduced by means of the new data type called HTA. The type representation differentiates primarily from that of parallel array by having additional information concerning the specification of data distribution as well as the computing grid.

Figure 2.1 shows the creation of tiled arrays. In addition to the chosen array partitioning, we can also specify the size of the available mesh of processors over which the array blocks are going to be mapped. We can see that the mesh of processors is replicated modulo its size over the partitioned array. The data-parallel operations are constrained to work on arrays with the same "shape" (type). The HTA model is an interesting contribution in terms of coding productivity and allows to control data distribution, something that is hidden in HDP. Apart from productivity reasons, the model does not propose formal optimization methods to achieve performance improvements and directly exposes to the programmer a specific architecture model based on a non-hierarchical distributed memory.

2.2 Hierarchical Tasks

The partitioning of data proved to be an efficient technique to provide parallel operations to the programmer in an intuitive manner. Another idea is to concentrate into the (hierarchical)

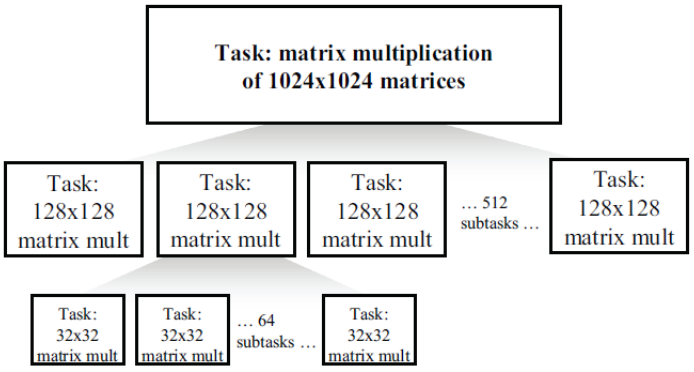


FIGURE 2.2: Hierarchy of tasks in Sequoia (Source [3])

partitioning of computation, which is implicit in previous approaches by the data type (excepting HTA), to map them into dedicated memory models or data-driven architectures.

2.2.1 The Sequoia Approach

Looking for performance of parallel programs, the abstraction level lowers and recent language trends follow specific architecture representations. Based on the fact that approaches focusing on non-hierarchical memory model are difficult to adapt to new memory hierarchies, Fatahalian et al. proposed a new language called *Sequoia* [3].

Sequoia is based on the memory model named Parallel Memory Hierarchy (PMH), where the memory is modeled as a tree whose leaves represent computing nodes. The computation is specified through hierarchical, isolated and parametric tasks (see Figure 2.2). A task works on explicitly partitioned arrays and operations using data-parallel directives (map, reduce). A task may take parameters and may contain other tasks to which it communicates via input and output parameters. Therefore, communication between tasks is explicit. The task hierarchy is mapped to the memory hierarchy, leading to data transfers in the case where two tasks get mapped to different stages. The execution model allows the programmer to define task variants which are selected in the mapping process.

Unlike NDP, mapping choices such as data and computing distribution, among others parameters proposed by Sequoia, are specified by the programmer. The Sequoia approach is located at a lower abstraction level. It follows a programming approach in correspondence to a specific architecture in order to achieve better performance than previous solutions.

2.2.2 HiDP: Hierarchical Data Parallel Language

Recents trends for programming highly parallel embedded systems shows the need to expose a general enough architectural model. Similarly to Sequoia, Mueller and Zhang have developed

```
# implementing C = alpha * A * B + beta * C 2
function GEMM
input float alpha, beta, A[M][K], B[K][N];
inout float C[M][N];
{
  float C1[M][N];
  map {
    m:=[0:M); n:=[0:N); k:= [0:K);
      c0 = a[m][k] * b[k][n];
    } reduce ("+", C1[m][n], c0, k:=[*]);
    C = alpha * C1 + beta * C;
}
```

LISTING 2.1: GEMM implementation in HiDP

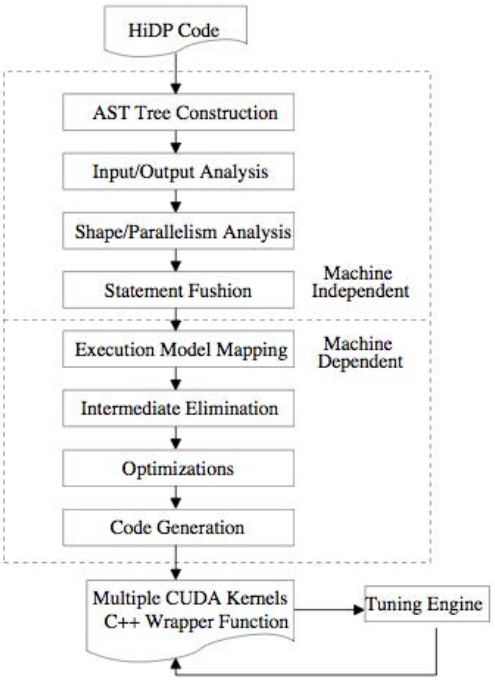
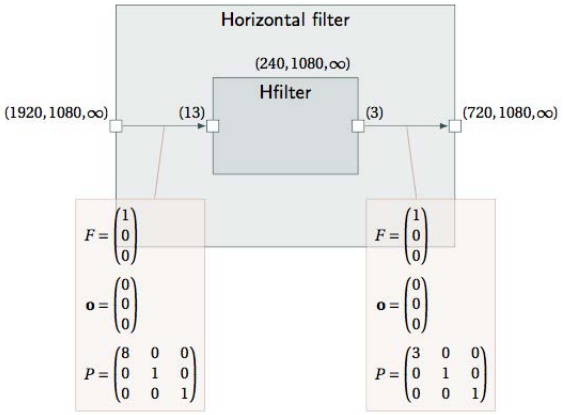


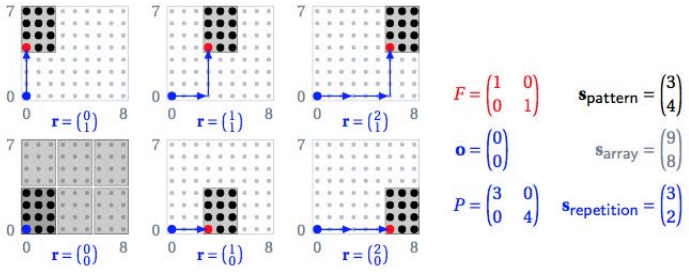
FIGURE 2.3: Optimizing compiler of HiDP (Source [4])

a hierarchical data-parallel language for General-Purpose computing on Graphics Processing Units (GPGPU) called Hierarchical Data Parallel Language (HiDP) [4]. HiDP proposes a hierarchy of threads called *map blocks*, similar to the actual execution model in Sequoia.

Each map block definition introduces a set of iterators to defines its *shape*, where all iterations are intended to be parallel. For instance, Listing 2.1 describes the General Matrix Multiplication (GEMM) in HiDP. Map blocks are composable similar to NDP. The model is then analyzed and optimized following the compilation chain of Figure 2.3. For instance, multiple map blocks may be merged together according to their shape factor to get as much in-place computations as possible.



(a) Filter block with fitting and paving matrices



(b) Tiling the array

FIGURE 2.4: Array-OL visual formalism (Source [5])

2.3 Low-level Trends

Finally, we found low-level approches such as OpenCL [10], CUDA [11] and OpenMP [34] where the programmer needs to handle almost everything: synchronization, data and computing distributions, etc. They do not offer data abstractions for parallel programming and are mostly implemented as libraries in low-level host languages such as C/C++.

The amount of work to be done in order to have a ready-to-run parallel implementation is considerable and we must take into account device-specific operations and constraints such memory copies from host to device and viceversa, task partitioning.

2.4 Model-Driven Representations

The model-driven community made interesting progress towards the application of MDE techniques for the design and development of Embedded Systems (ESs) [35]. It allows designers to model the system gradually by separating functionalities from its supporting platform. It pushes models everywhere as the abstraction method and model transformations as the path to

final system implementation. For instance, Model-Driven Architecture (MDA) combines different models for the applicative and platform parts, called Platform-Indepedent Model (PIM) and Platform-Dependent Model (PDM) respectively, and proposes model transformations to progressively adapt the PIM to the given architecture model (PDM) to obtain a Platform-Specific Model (PSM) model. PSM contain platform specific information and it is ready for final code production.

Recent works explore different adaptations or extensions of MDE techniques to address data-parallel applications. A mature solution to modeling massively parallel ES is GASPARD [8]. Model based technologies like GASPARD have a great flexibility in terms of program specification because it is not tied to any particular architecture. For this reason, the system is described via two models based on abstractions proposed by *MARTE* and the Array Oriented Language (Array-OL) [5, 36]. Indeed, Gamatié et al. rely on Array-OL for the modeling of parallel computations [5].

We show in Figure 2.4 the visual Array-OL formalism. The designer specifies tiling strategies for the input and output data via a *fitting* matrix F, a *paving* matrix P and an offset $\mathbf{o}$ such that

- Tiling repetition formula: $\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{repetition}, \mathbf{ref_r} = \mathbf{o} + P \cdot \mathbf{r} \bmod \mathbf{s}_{array}$
- Iteration inside each tile: $\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{pattern}, \mathbf{e_i} = \mathbf{ref_r} + F \cdot \mathbf{i} \bmod \mathbf{s}_{array}$

where $s_{pattern}$ is the pattern or tile size, $s_{repetition}$ its repetition and s_{array} the array size. Both strategies are related by the repetition iterator $\mathbf{r}$, which precisely determine data-dependences between input and output tiles. Therefore, Array-OL is used to partition the computations and define affine dependencies without any specific computational model attached to it. It is particuraly adapted to stream-like applications.

Compared to other parallel languages, MDE approaches separate application from platform related informations. Both models are combined iteratively following a specific methodology such that platform-specific constraints are introduced in an incremental manner. The embedding of Array-OL inside GASPARD makes it a powerful framework where domain-specific languages leverages from well-known high-level abstractions and rigorous processes of the MDE community.

2.5 Conclusion

In Figure 2.5, we classify reviewed approaches with respect to its abstraction level and how much hardware information they expose to the programmer. Bigger points indicate also higher

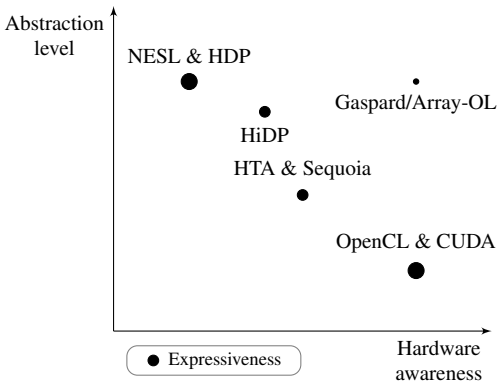


FIGURE 2.5: Language classification: Abstraction, Hardware-specific features and expressiveness

language expressiveness. The language expressiveness refers to the support of features such as arrays, structs, pointers, dynamic object creation and communications, among other things. We observe that each approach offers different ways to manage architectural constraints while offering nice abstractions to handle the complexity of parallel programming. Hardware adaptation is done automatically with optimizations and no user interaction (HDP), user-controlled with (Sequoia, HiDP) or without (HTA) automatic optimizations, or completely left to the programmer without any prior optimizations other than the host compiler or runtime may achieve (OpenCL, CUDA, OpenMP).

Several reviewed approaches are based on data-parallel operations. However, recent works orientate theirs efforts in task parallel implementations, integrating in some cases architecture-specific informations. Representations are mainly hierarchical, a key characteristic of model-driven approaches. However, existing model-driven approaches with parallel computation support have often limited expressiveness.

Inspired from the data-parallel paradigm and in the context of model-based design, we apply the data-parallel paradigm on Statecharts proposing a very expressive modeling language, as we will show in Chapter 5. However, Statecharts have a rather loose semantics in current modeling languages. In the next chapter, we introduce the formalism and study several semantic formalizations.

Chapter 3

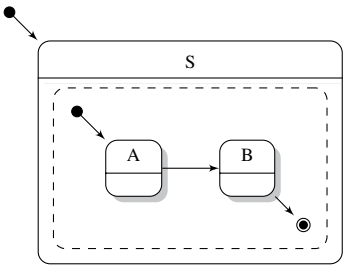
Semantics of Hierarchical State Machines

Contents

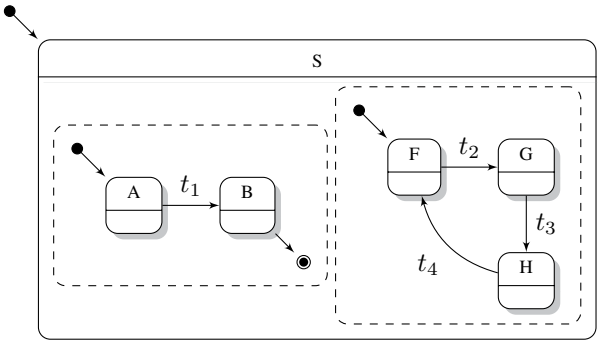
In this chapter we will present the foundations of Hierarchical State Machines (HSMs), widely known as the Harel's Statecharts. Then, we will discuss its informal and formal semantics and its evolution into the Unified Modeling Language (UML). We will point out main drawbacks to achieve a complete formalization of HSMs and we will review some of the most influencial research work on this matter.

3.1 Harel's Statecharts

The formalism of HSM was originally proposed by Harel in [20]. It is an extension of the formalism of state machines for the specification and design of complex discrete-event and reactive



(a) OR composition: either A or B can be active inside state S



(b) AND composition: two states are simultaneously active inside S, one for each region

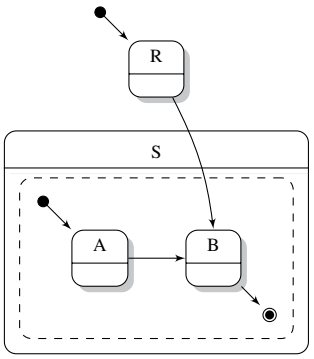
FIGURE 3.1: Hierarchical state machines

systems. It is presented as a visual formalism that mainly extends conventional transition systems with AND and OR compositions of states and inter-level transitions. That is, a state may contain another state machine or parallel ones allowing state abstractions (bottom-up view) and refinements (top-down view). For instance, Figure 3.1 shows two instances of the Harel Statecharts. The state machine of Figure 3.1(a) is an OR composition where only one state can be active inside S. Figure 3.1(b) is an AND composition with two regions where only one state can be active at each region. By active, we refer to the current state of the state machine at execution time.

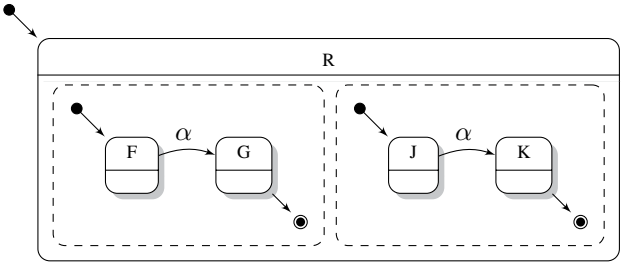
Actually, the original Harel's Statechart is highly expressive. In addition to hierarchical compositions, it also integrates many more features:

1. **History states**: Ⓗ. It represents the last visited state in a group of states, which allows to model resumption.

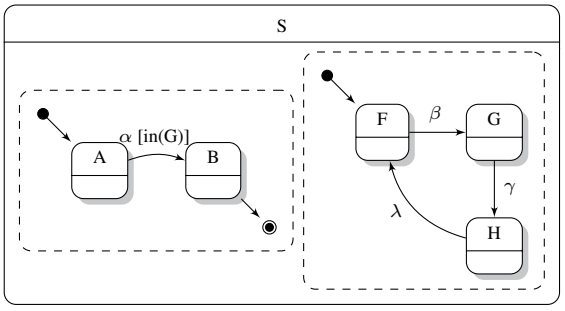
2. **Inter-level transitions**: Transitions may link states across the hierarchy. From a language point of view, the property is clearly non-constructive. However, it turns out to be a useful artifact for system modeling.



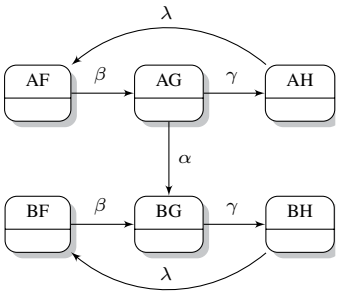
3. **Synchronous transitions**: Harel Statecharts extend the automata product with synchronous transitions, i.e., two active states in an AND composition waiting for the same message may progress at the same logical time. For instance, the Figure below shows an AND composition where both regions progress at the same logical time at reception of message α.



4. **Constrained asynchronous compositions**, also known as *in-state* conditions: In terms of the automata product, it allow us to consider only a subset of the transition set given by the product of two parallel automata.

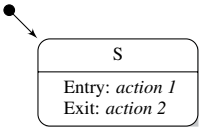


In-state condition



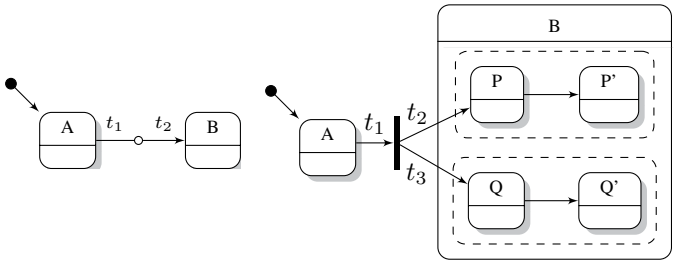
Reduced automata product

5. **Entry and exit state actions**: Actions to be executed when entering and exiting a given state.



6. **Static reactions** (SR), or *internal transitions* in the UML dialect: They are simply reactions that do not enter or exit the current state.

7. **Compound transitions**: A compound transition represents a group of transitions to be executed at the same logical instant. For instance, in the left state machine t_1 and t_2 form a single compound transition and are executed atomically and *in parallel*. The right state machine implements a *fork* compound transition $t = \{t_1, t_2, t_3\}$, which are also parallel and atomic.



The syntax of transitions, as defined by STATEMATE [37], is the following one:

$$\alpha[C]/\beta;$$

The transition is sensitive to message α under condition guard C and performs action β (which may contain message sending). The actions can be separated by a semicolon which, contrary to the traditional sense, denotes parallel execution.

3.2 Semantics of Statecharts

We reviewed in the precendent section some of the visual features present in Statecharts as well as its intended semantics. The Statecharts informal semantics is detailed exhaustively in [38] and implemented in a system modeling tool called STATEMATE. Its semantics and tool support has evolved years later into a complete working environment for the development of complex reactive systems, *Rhapsody* [39, 40].[1]

3.2.1 The STATEMATE Semantics

The *system* is a single state machine where its parallel components are supposed to describe different, and usually communicating, pieces of it. The behavior of a system is a set of possible runs or system responses to a given set of external stimuli coming from the environment. A sequence of *status* represents a run of the system. Such sequence is computed by the evaluation

[1] Owned by IBM.

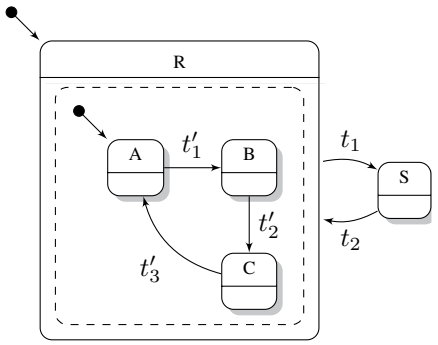


FIGURE 3.2: Hierarchical priority between t_1 and t'_1

step algorithm. More precisely, the status is the global system context containing the current state, valuation of variables, generated events and information regarding the system's history.

Statecharts implement broadcast communications, i.e., the set of events is the same for the entire system. There are two kinds of events: internal and external. Internal events are generated by the state machine and are intended to be consumed by the state machine itself. For instance, entering and exiting states triggers internal events called $en(s)$ and $ex(s)$, which can be captured by any transition. External events comes from the system environment.

The system reaction depends on the HSM *configuration*, which is the maximal set of states the system can be in simultaneously. That is, the set of active states along the hierarchy. It depends on the state composition type, whether we have an OR (only one state can be active) or an AND composition (only one active state per parallel region). In any case, the current configuration can be determined from the active set C of basic, or leaf, states in the hierarchy. By transitive relation of containment, parents of state $s : C$ must also be part of the active configuration. For instance, a possible active configuration of HSM in Figure 3.1(b) could be $C = \{S, A, G\}$.

From an active configuration, the evaluation step consists of finding an active and non-conflicting compound transition. Two transitions are in conflict if they share the same source state and are both enabled at the same instant. Harel defined a priority over transitions following the Statechart hierarchy (outside-in) to resolve conflicting transitions at different hierarchical levels. For instance, consider Figure 3.2 with current configuration $C = \{R, A\}$. We may either take t'_1 or t_1 if both transitions have the same trigger. Then, the transitions are said to be in conflict. The priority policy favors t_1 over t'_1 such that we can safely take t_1 and quit state R no matter what the active substate of R was. This property gives to Statecharts a *try-catch* style semantics. Conflicting transitions at the same hierarchical level introduce non-determinism.

The *basic step algorithm* performs the following actions:

The Inputs

- The status of the system, i.e., the current global context, which consists of

  - The state configuration.
  - Valuations of variables.
  - Relevant information on the history of states.
  - List of events that were generated internally in the previous step.
  - A list of external changes presented by the environment since the last step.

The Outputs: A new system status

- Stage 1: Step preparation
  - Add the external events to the list of internally generated events.
  - Execute all actions implied by the external changes. It implies for instance, the update of all variables, conditions.
  - Execute scheduled actions and generate timeout events.
- Stage 2: Compute the contents of the step
  - Compute the set of enabled CTs.
  - Remove conflicting transition by applying priorities.
  - For each CT, compute the set of enabled SRs defined in states that are currently active but not being exited by any of the CTs in the set.
  - If there are not enabled transition then the step is empty. Otherwise, this set of transitions consistutes the current step.
- Stage 3: Execute the CTs and SRs
  - From the set of enabled transitions computed in stage 2, execute its associated actions.
  - Let Sx and Sn be the sets of exited and entered states from the current state, then:
    * update the history of all parents of states in Sx
    * delete the states in Sx from the list of states in which the system resides
    * execute the actions associated with exiting the states in Sx
    * execute the actions of the current state
    * execute the actions associated with entering the states in Sn
    * add to the list of states in which the system resides all states in Sn

In conclusion, the Statechart senses its inputs, external and internal, and reacts to them by adding more events, if any, to the set of generated events. The generated events are not taken into

account in the current step — avoiding inifinte loops — but in the following one. The actions to execute are taken from the set of enabled transitions (CTs and SRs). Because the actions are supposed to run in parallel, they may introduce what the author called *write-write racing* situations. That is, two actions writing to the same variable.

3.2.2 Formal Semantics of Statecharts

As shown in the precedent chapter, Statecharts as defined by Harel implies a quite involved semantics. Soon after its invention, many researchers dived into the question of defining a formal semantics and it turned out to be harder than excepted. As a consequence, many Statecharts variants appeared since then [26]. However, none of them formalizes the Statecharts semantics as implemented in the industrial tool STATEMATE, till the work of Mikk et al.. They introduce in [41] the official semantics as supported in STATEMATE by closely following [38].

The STATEMATE semantics defines a set of finite state labels, Σ, and the kind of states, $TYPE \triangleq AND \mid OR \mid BASIC$. From these sets, it defines the mathematical structure $StateTree$, which consists of the following parts:

- $root$, a special state that it is intended to contain the actual Statechart
- $init$, an initial state
- $\rho : \Sigma \rightarrow \mathcal{P}(\Sigma)$, the state hierarchy partial function that maps a state to its contained substates
- $\phi : \Sigma \rightarrow TYPE$, the map kind partial function

Above definitions provide enough information to formalize the state hierarchy and to introduce some correctness conditions. For instance, Definition 3.1 is an example of a correct hierarchy of states with respect to the $root$ state as defined in [41].

Definition 3.1. The $root$ state is of kind OR and no state may contain it

$$dom(\rho) \setminus \bigcup ran(\rho) = \{root\} \wedge \phi(root) = OR$$

Based on the set of possible events EV, the allowed event expressions are defined as:

$$EE \triangleq TRUE_E \mid B\langle EV \rangle \mid NOT_E\langle EE \rangle \mid AND_E\langle EE \times EE \rangle \mid OR_E\langle EE \times EE \rangle$$

Equivalently, the set of valid conditions C is inductively defined as

$$C \triangleq TRUE_C \mid In\langle \Sigma \rangle \mid NOT_C\langle EE \rangle \mid AND_C\langle EE \times EE \rangle \mid OR_C\langle EE \times EE \rangle$$

Then, the transition expression is a $LABEL$ with three parts

1. $event\_expr : EE$, an expression to validate according to the current set of events
2. $condition : C$, a guard expression for the transition to be executed
3. $action : \mathcal{P}(EV)$, a set of events to be emitted

The transition itself $t : TR$ is a structure containing a label and going from a non-empty set of states, $source : \mathcal{P}(\Sigma)$, to another non-empty set of states, $target : \mathcal{P}(\Sigma)$. Source and target states becomes sets because the state hierarchy has been flattened. The semantics defines evaluation functions to check if the transition is enabled.

The Statechart, SC, is formalized using the $StateTree$ structure that provides the state hierarchy and the set of labeled transitions TR that links them:

$$\sigma : StateTree$$
$$\tau : \mathcal{P}(TR)$$

According to this semantics, the status $s : STATUS$ of a Statechart contains

- $csts : \mathcal{P}(\Sigma)$, the set of active states or the state configuration.
- $events : \mathcal{P}(EV)$, the current set of events.
- $sc : SC$, the actual Statechart.

Based on this status definition, the semantics instantiates a transition system

$$TS = (STATUS, INIT, STEP)$$

where $STATUS$ is the universe of states, $INIT$ the initial one and $STEP \subseteq STATUS \times STATUS$ the transition relation. The run of the system is an infinite sequence of formal statuses following relation $STEP$.

To give an idea about the complexity of the step relation, we cite hereafter its formal definition using Z notation [42]:

$$
\begin{aligned}
&\text{let } ET \quad == \{tr : sc.\tau \mid enabled(tr, csts, events)\} \\
&\text{let } HPT \quad == \{etr : ET \mid (\forall tr : ET \bullet \neg SAnc(scope(tr, sc.\sigma), \{scope(etr, sc.\sigma)\}, sc.\sigma) \\
&\text{let } MNS \quad == (\mu\, ncs : \mathcal{P}(\mathcal{P}(HPT)) \mid (\forall set : ncs \bullet \neg conflicting(set, csts, sc.\sigma)) \\
&\qquad \wedge (\forall set : ncs; tr : HPT \bullet \neg conflicting(\{tr\} \cup set, csts, sc.\sigma) \Rightarrow tr \in set)) \\
&\qquad \bullet (\#MNS == 0 \Rightarrow csts' = csts \wedge events' = \emptyset) \\
&\qquad \wedge (\#MNS \neq 0 \Rightarrow (\exists EN : MNS \bullet \\
&\qquad\quad \text{let } Exited == \bigcup \{tr : EN \bullet exit(tr, csts, sc.\sigma)\}; \\
&\qquad\qquad Entered == \bigcup \{tr : EN \bullet enter(tr, csts, sc.\sigma)\} \bullet \\
&\qquad\qquad\quad (csts' = (csts \setminus Exited) \cup Entered) \\
&\qquad\qquad\quad \wedge events' = \bigcup \{tr : EN \bullet tr.label.action\} \\
&\qquad\qquad\quad \cup \{st : Exited \bullet exited(st)\} \\
&\qquad\qquad\quad \cup \{st : Entered \bullet entered(st)\}
\end{aligned}
$$

Clearly, the semantics is too complex to maintain and hardly extensible. Furthermore, it still does not take into account the valuations of variables and history states.

We see two possible reasons for such complexity, whether some features of Statecharts are actually difficult to formalize or the approach taken in the STATEMATE formalization turned out to be too cumbersome. As we will see in next sections, the former seems to be the main cause. Features like parallelism, event handling, asynchronous compositions, lead to a non-determinstic semantics. Non-deterministic properties of languages are frequently formalized using power-domains, to keep track of all possible runs of the system, or using a transition semantics [43]. However, inter-level transitions, preemption and try-catch semantics are deeply non-modular and tend to require global and complex structures along the entire formalization. Not surprisingly, such complexity reappears in recent research work on formal specifications of Statecharts in the context of UML. In the next section, we review the most important ones to us.

3.3 The Unified Modeling Language

The formalism of HSM has been successfuly adopted by the industry for system level modeling, simulation and code generation. It continues to be improved by a standard organization the Object Management Group (OMG), under the widely known name UML [21].[2]

[2]In this thesis, we reviewed UML version 2.4.1, 2011.

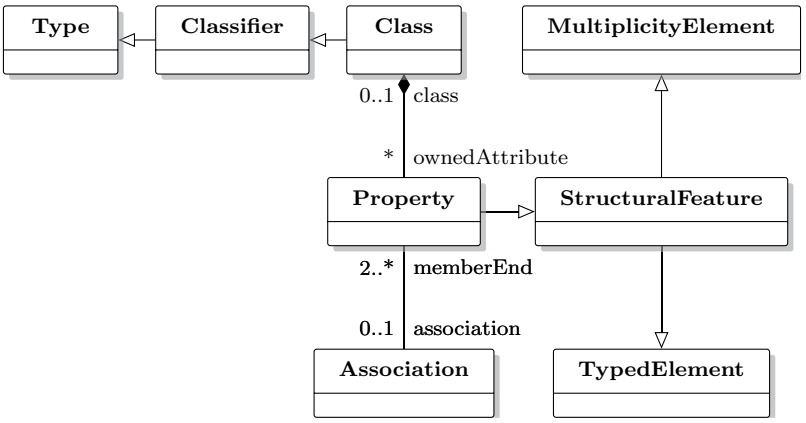


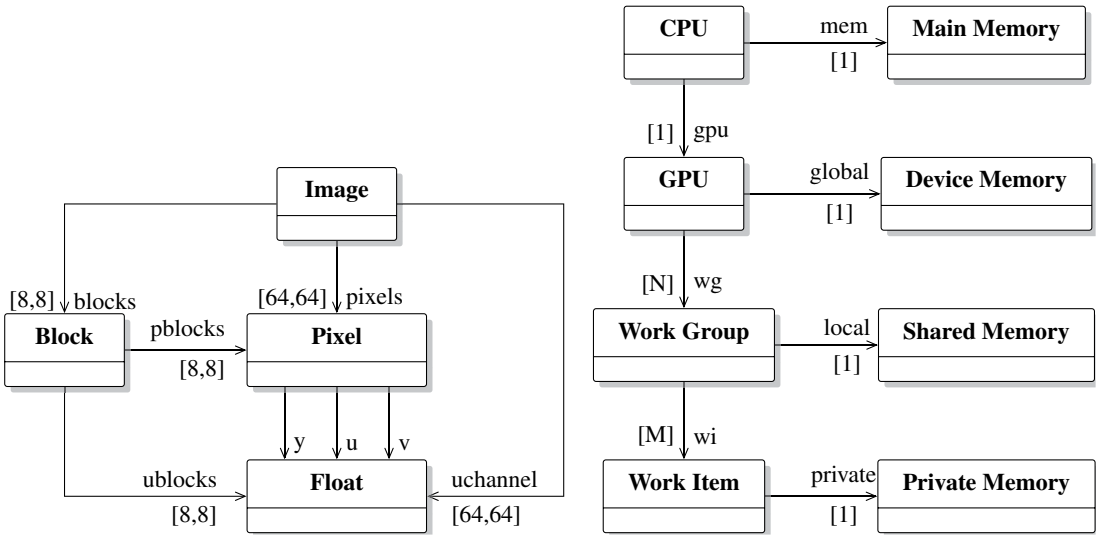
FIGURE 3.3: UML class and associations

UML is a unified and model-driven approach for the modeling of complex hardware/software systems. It proposes interesting features to manage and specify different system development aspects:

- Abstractions to classify relevant features of the system and how they relate to each other: *Structured Classifiers*.
- Activity graphs that allow the designer to organize the actions to perform for a given task.
- Modeling of each feature's behavior using hierarchical state machines: *State Machine*.
- A set of concrete actions to perform on the model by the state machine, such as communication primitives or structural updates.
- Models for the specification of use cases.

The UML specification is entirely model-based. For instance, Figure 3.3 shows the classifier implementation in UML v2.4.1 [21]. The class is intended to model a particular component of the system. It contains properties that *associates* it to zero, one or more components, according to the specific multiplicity of the association. More precisely, it is the model view of the type definition of an object in Object-Oriented Programming (OOP) languages like C++ or Java. For instance, Figure 3.4 shows in a visual manner two instances of the class meta-model: an Image and a General-Purpose computing on Graphics Processing Units (GPGPU) class model.

In addition to structural features of a class, the UML classifier may provide a specific behavior using HSMs. Therefore, the language enables the modeling of a structured network of communicating Statecharts.



(a) Image object model (b) GPU model

FIGURE 3.4: Class instances

3.3.1 A Structured Network of Statecharts

The UML class abstractions enclose Statecharts and connect them in a well-defined and structured manner. Therefore, we refer to the model as a structured network of Statecharts.

The class meta-model allows the designer to create models that can describe many possible configurations. The class allows the programmer to create a well-defined context to the state machine in order to communicate with the external world (including other classes) through its associations.

The UML Statechart is an extension of the Harel's Statechart. Its structure is presented in a model based way in Figure 3.5. As we can observe, state kinds are abstracted into a common concept, Vertex. A Vertex may be a basic state if it does not contain any region, an OR state if it contains only one region and an AND state if it contains more than one region. States containing at least one region are called *composite*. From Vertex, the state inherits incoming and outgoing transitions that complete the HSM structure. Special states such as initial, final, history and join/fork, are considered Pseudostates.

Similar to Harel's syntax of transitions, the transition is composed by a trigger, a constraint (or condition) and an opaque representation of an action called `Behavior`. Under this opaque type, the designer is free to choose his or her preferred language and it is frequently referred to as the *Action Language* of the state machine.

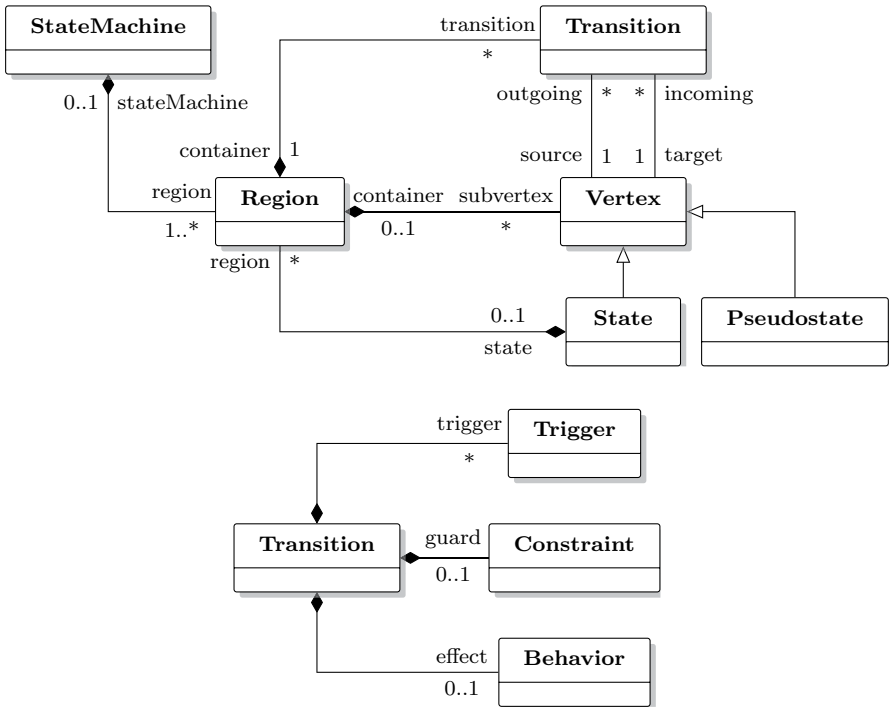


FIGURE 3.5: Simplified Meta-model of UML Statecharts

3.3.2 Informal Semantics

In [21], the OMG introduces an informal semantics of Statecharts where they specify the event processing algorithm for state machines, called the Run-To-Completion (RTC) step. The RTC informally describes the actions to perform in order to determine the set of enabled transitions to fire and how the compound transition is executed afterwards.

Contrary to traditional Statecharts, the state machine handles the arrival of events one at a time and it is not based on broadcast communications. That is, it exposes a message oriented, First In-First Out (FIFO)-like communication. Moreover, there is no difference between internal or external events, they are all considered external.

Following almost the same policies as Harel for the computation of enabled compound transitions, the actions of each fireable transition are executed *sequentially* — and not in-parallel as before. In UML Statecharts, the transition with higher priority is the more nested one (inside-out).

At a given state configuration, the RTC step specifies the following actions to perform:

1. From the transition source state, all substates are exited and their corresponding exit actions are executed. Note that there is no internal events generated here.

2. The chain of state exits continues until the first region that contains, directly or indirectly, both the main source and main target states is reached. The region that contains both the

main source and main target states is called their least common ancestor, or *lca*, and it constitutes the scope of the transition.

3. The target configuration of states is entered and theirs corresponding actions are executed starting with the outermost one.

3.3.3 Formal Semantics

A complete formal specification of UML is a challenging work. The OMG undertakes it on a subset of UML called foundational UML (fUML) [44]. However, their semantics does not cover the formalization of state machines, neither its communications. They map models to a particular "surface" language, such as Java, in order to concretize theirs semantics.

We found multiple formalizations of UML state machines in the literature [1, 22–25]. The most accomplished work to the best of our knowledge is due to Liu et al.. He introduces its own abstract syntax for HSM closely following the UML state machine specification. For instance, the state is a tuple of the following form:

$$s = (\widehat{r}, \widehat{t_{def}}, \alpha_{en}, \alpha_{ex}, \alpha_{do}, \widehat{en}, \widehat{ex}, \widehat{ex}, \widehat{cr}, sm, \widehat{t})$$

where

- $\widehat{r}$ is the set of regions.
- $\widehat{t_{def}}$, α_{en}, α_{ex} and α_{do} are the set of deferred events, the entry, exit and do behaviors.
- $\widehat{en}$ and $\widehat{ex}$ are the set of entry point and exit point pseudostates.
- $\widehat{cr}$ is the set of connection point references. sm is the owner state machine.
- $\widehat{t}$ is the set of internal transitions.

The region is the cartesian product of vertex and transitions

$$r \triangleq (\widehat{v}, \widehat{t})$$

where $\widehat{v} \subset (S \cup PS \cup S_f)$, i.e., it is either a state, a pseudostate or a final state. Compared to the previous formal representation, here we have two mutually recursive definitions, region r and state s through $\widehat{v}$.

The transition is the tuple $t = (sv, tv, \widehat{tg}, g, \alpha, \iota, \widehat{tc})$ where

- sv and tv are the source and target vertex, or states.

- $\widehat{tg}$, g, α and ι are the set of triggers, the guard, the behavior and the container of the transition.
- $\widehat{tc}$ represents the special situation that a join or fork pseudostate connects multiple transitions to form a compound one.

The author defines an evaluation function $\mathcal{A}$ that takes the behavior part of the transitions and returns a set of events to be sent.

Then, the state machine is defined as $sm \triangleq (\widehat{r}, \widehat{cp})$. It contains a region (and by transitivity a set of states) and the entry and exit points of sm.

Finally, Liu et al. defines *the system* of n communicating state machines as

$$sys \triangleq |^{C}_{i\in[1,n]} Sm_i$$

where Sm contains the actual state machine, its events pool and a set of shared variables: $Sm \triangleq (sm, P, GV)$. The operator $|^C$ synchronizes all state machines on events in C. This operator is needed because of the synchronous sending of messages as defined by UML. That is, state machine sm_i sends a message to sm_j and its RTC is not finished until the sm_j has finished its own step.

Given a tuple of n configurations of state machines $(k_1, \ldots, k_n)$, the semantics instantiates a Labeled Transition System (LTS). The configuration is defined as $k = (ks, P, GV)$ where ks is the set of active states of the state machine sm_k. Then, it defines three main LTS rules:

$$\frac{|^{C}_{i\in[1,n]} Sm_i, k_j \to k'_j}{(k_1, \ldots, k_j, \ldots, k_n) \to (k_1, \ldots, k'_j, \ldots, k_n)} \text{ LTS1}$$

$$\frac{|^{C}_{i\in[1,n]} Sm_i, k_j \to k'_j, e = SendSignal(j, l), Merge(e, EP_l)}{(k_1, \ldots, k_l, \ldots, k_j, \ldots, k_n) \to (k_1, \ldots, k'_l, \ldots, k'_j, \ldots, k_n)} \text{ LTS2}$$

$$\frac{|^{C}_{i\in[1,n]} Sm_i, k_j \to k'_j, e = Call(j, l), e \in C, k_l \xrightarrow{e} k'_l}{(k_1, \ldots, k_l, \ldots, k_j, \ldots, k_n) \to (k_1, \ldots, k'_l, \ldots, k'_j, \ldots, k_n)} \text{ LTS3}$$

We have asynchronous composition with special cases for asynchronous and synchronous sending of signals, encoded in LTS2 and LTS3 respectively.

The main advantage of this formalization resides on its completeness with respect to the UML specification. It defines mutually recursive structures to formalize the state machine and the author uses such recursive definitions to define elegant evaluators of entry and exit behaviors

(not shown here though). Even if the state machine is defined recursively, the state configuration is still flattened. The action language is not specified but the author handles behaviors such as $SendSignal$ and $Call$. If state machines are willing to pass valued signals, they must use shared variables. Similar to the formal semantics of Harel Statecharts, the lack of modularity leads to scalability issues. Nevertheless, it is clear that such complexity can be hardly avoided given the specifities of UML state machines such as deferred messages, synchronous messages sending and local transitions.

Another interesting formalization was done by Seifert where he followed an equivalent path in his work [1]. It defines an abstract syntax but instead of using recursion on states and regions, it specifies a state hierarchy similar to the STATEMATE formalization. One of the most important differences with respect to the work of Liu et al. is that events may carry parameters. It defines events as an indexed family of sets $(P_e)_{e\in\mathcal{E}}$ where $\mathcal{E}$ is a set of events. Then, a parameterized event set E_v with $v \in \mathcal{E}$ is defined as

$$E_v = v\langle P_1 \times \cdots \times P_n \rangle$$

Such definition enables the author to introduce the set of all event instances as

$$E = \sum_{v\in\mathcal{E}} E_v$$

For instance, a particular event set $\mathcal{E} = \{a, b\}$ with $P_1^a = \mathbb{N}$, $P_2^a = \mathbb{B}$ and $P_1^b = \mathbb{B}$ gives the set of all event instances:

$$\begin{aligned} E = E_a + E_b &= a\langle \mathbb{N} \times \mathbb{B} \rangle + b\langle \mathbb{B} \rangle \\ &= \{a(0, true), a(0, false), \ldots, b((true))\} \end{aligned}$$

Seifert distinguishes two kind of actions, event sending and updates of the binding context. Indeed, the state machine step depends on a global binding context of variables.

If his formalization works nicely on complex data structures as a binding context, it does not handle multiple communicating state machines as [22] does.

3.4 Synchrounous Statecharts

Defining a suitable semantics for Statecharts is challenging as we have seen in the previous sections. Many problems arise concerning inter-level transitions, history states and asynchronous compositions, which are source of non-determinism. Harel and UML Statecharts are hence subject to racing conditions due to their asynchronous semantics. In the domain of safety-critical

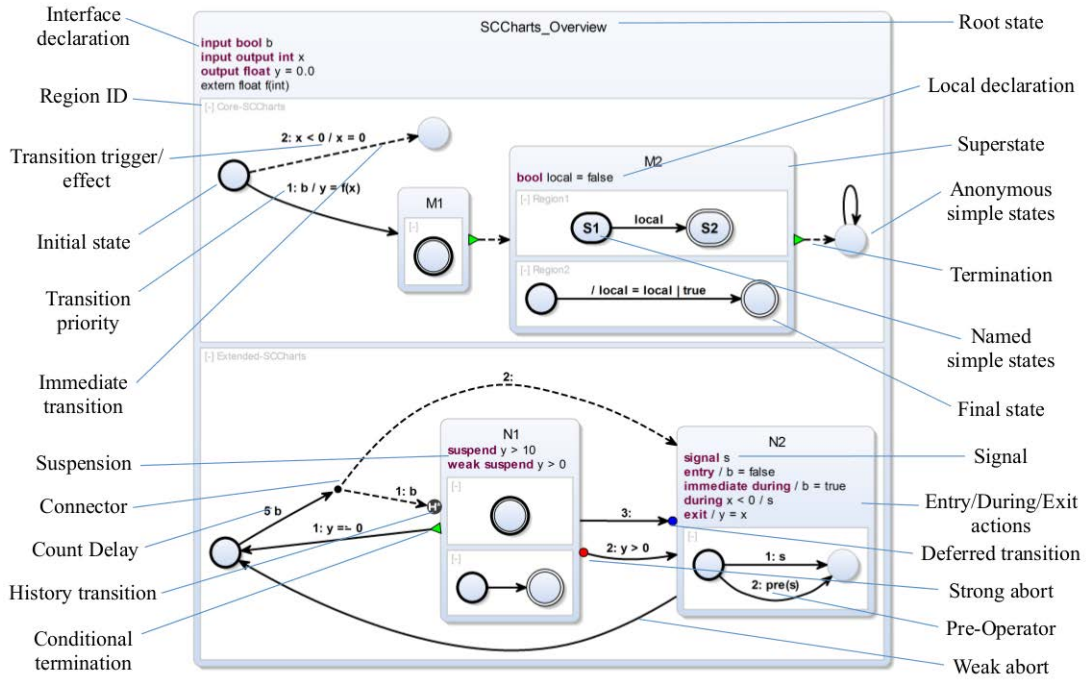


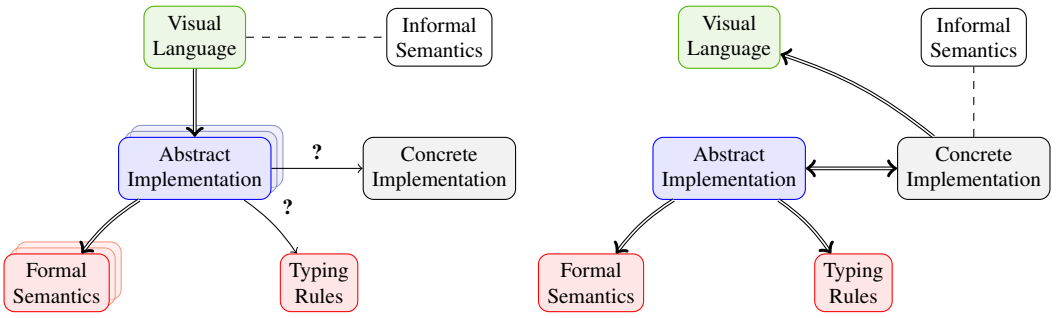
FIGURE 3.6: Visual syntax overview of SCCharts (Source: [6])

reactive systems, non-deterministic behaviors are not desirable. We found very interesting and recent research work on synchronous and hierarchical Statecharts [6]. von Hanxleden et al. introduces a new visual modeling language for reactive systems called *Sequentially Constructive Statecharts*, or *SCCharts*. It is inspired from SyncCharts [45], a visual formalism for the ESTEREL language [46], but removes some of its limitations such as multiple variable assignments while keeping determinism. Quoting Harel's statement about Statecharts [37], the author nicely states the evolution of all three formalisms:

> Statecharts = State-diagrams + Depth + Orthogonality + Broadcast Communications
> SyncCharts = Statecharts syntax + Esterel semantics
> SCCharts = SyncCharts + Sequential constructiveness + Extensions

Figure 3.6 shows many of the Statecharts features supported in SCCharts. The formalism relies on a concrete language implementation, the *SC Language* (SCL), and its corresponding graphical elements. SCL has a very basic — yet expressive enough — kernel of language constructions:

$$
\begin{aligned}
\langle s \rangle \quad &:= \langle x \rangle \text{ '}\mathtt{=}\text{' } \langle e \rangle \\
&\mid \langle s \rangle \text{ '}\mathtt{;}\text{' } \langle s \rangle \\
&\mid \text{'}\mathtt{if}\text{' '}\mathtt{(}\text{' } \langle e \rangle \text{ '}\mathtt{)}\text{' } \langle s \rangle \text{ '}\mathtt{else}\text{' } \langle s \rangle \\
&\mid \langle l \rangle \text{ '}\mathtt{:}\text{' } \langle s \rangle
\end{aligned}
$$



(a) Visual Statechart based approaches (b) Formal language based approaches

FIGURE 3.7: From Statecharts to formal languages

| '**goto**' $\langle l \rangle$
| '**fork**' $\langle s \rangle$ '**par**' $\langle s \rangle$ '**join**'
| '**pause**'

Based the above language, the author introduces its constructive semantics in an operational manner [47]. Essentially, it extends the synchronous Model of Computation (MoC) [48] by allowing variables to be read and written multiple times as long as the sequential specification of the program provides enough information to avoid race conditions. Because of the structured semantics, i.e., guided by the language constructions, the formalization is cleaner than the precendent ones, extensible and modular.

3.5 Conclusions

We presented the Statechart visual formalism as first introduced by Harel and discussed about its informal semantics implemented in the industrial tool STATEMATE. Its official and formal semantics are also presented where we can see the complexity that Harel's visual constructions imply throughout the formalization. Such complexity comes back into recent research work around the UML language and its Statecharts. The reviewed formal semantics concerning UML did not manage to handle the overwhelming amount of informal features that the language proposes.

Research work on synchronous languages takes the other way around. Starting from strong mathematical foundations, the researchers advanced towards convenients visual elements "à la Statecharts". The constructive synchronous semantics is well-established, allowing imperative extensions such as those found in SCCharts.

Figure 3.7 summarizes the approaches for the semantical specification of visual and textual languages we reviewed. We see that visual formalisms such as Harel Statecharts, including

Statechart Features		Approaches			
		STATEMATE [41]	UML [22]	UML [1]	SCCharts [6]
HSM	Inter-level Transitions	yes	yes	yes	no
	Composition	async/sync	async	async	sync
	History	no	no	no	yes
Communications	Distributed	no	yes	no	no
	Message Parameters	no	no	yes	no
Action Language	Send/Receive	yes	yes	yes	yes
	Single Variables	yes	no	yes	yes
	Array Variables	no	no	no	yes
	Dynamic Statecharts	no	no	no	no

TABLE 3.1: Supported Statechart features of reviewed research work

UML, has given rise to many abstract implementations and their corresponding formalizations. In contrast to graphical approaches, the structural approaches are mainly driven by a concrete language implementation (synchronous Statecharts) over which the semantics and, possibly, its type system are specified. From a well-founded concrete language, we can more easily found convenient visual elements and, more importantly, build a constructive semantics, which is modular and amenable to further extensions.

We show some common features between the reviewed semantics in Table 3.1. We classify them according to the structure they belong to. Transitions, history states and type of state compositions concern the state machine structure. Message with parameters and communications between HSMs concern the communication layer. Finally, we highlight the action language expressiveness in terms of support for send/receives, simple and array variables and dynamic creations of HSM.

The distributed component is key for the specification of an entire model-based Statechart program, where Statecharts use object definitions as their contexts in the UML style of programming. Inspired from these works, we develop in Chapter 6 a constructive and hierarchical semantics of Statecharts with asynchrounous communications based on a concrete action language.

A precise semantics enables a formal reasoning on the model. Additionally, it may serve as a support for direct interpretation or code generation. In the next chapter, we review optimization and code generation approaches in order to produce efficient code from high-level models.

Chapter 4

Code Optimization and Generation of High-Level Models

Contents

This chapter introduces the compilation of Statecharts "à la Harel". We will overview most used techniques for optimization and code generation in the industrial and research context.

4.1 Compilation of UML Statecharts

The Unified Modeling Language (UML) language is uniquely designed as a modeling/specification language and it is not tied to any particular compilation method or optimizing flow. UML favors model-based design at all levels of the development process. Furthermore, the standard left many semantic variation points open to specific implementations [49]. Most Model-Driven Development (MDD) frameworks based on or inspired from UML, such as Rhapsody [50] or GASPARD [8], support model-driven code generation and use object-oriented design patterns to produce executable models.

MDD tools propose different *back-ends* to produce executable code from UML models. Executable implementations of UML Statecharts are frequently based on one of three different

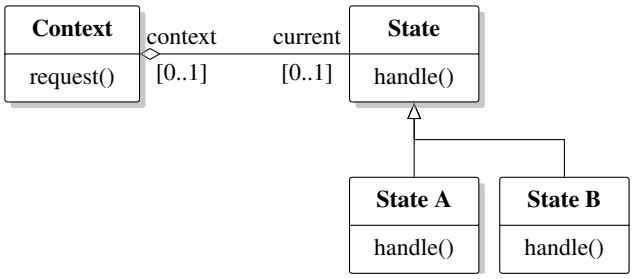


FIGURE 4.1: State design pattern

methods: state design pattern [51], the State Table Transition (STT) [52] and the nested switch case statements [53].

The most used technique for generating code from Statecharts relies on the state design pattern [54] shown in Figure 4.1. Basically, each state is translated into a class of the host language that handles a certain number of events depending on its outgoing transitions. If an event arrives, the context calls the virtual method `handle()`, which returns the next state according to the encoded transition triggered by such event.

The STT approach creates a table describing the relation between states and events. The boost C++ Statechart library [55] hides behind a heavy use of C++ metaprogramming a transition table to map efficiently states to transitions.

Finally, the Nested Switch Case method consists of, as its name implies, a set of nested switch case statements where the top ones filter the current state of the state machine and further nesting levels apply the final action following the received event.

The optimizations are frequently done by the backend-end compiler (C/C++, Java), which does not necessary know the original Statechart structure, and hence may miss interesting optimizations [56]. On this direction, we reviewed different approaches for model transformations and optimizations either before code generation or all along the optimizing and code generation process.

4.1.1 Model-to-Model Optimizations

Certain research works handle optimization issues via Model-to-Model (M2M) transformations. There are a good variety of dedicated tools, most of them based on the Eclipse Framework, for M2M transformations such as ATL [57], Epsilon Transformation Language (ETL) [58], Query/View/Transformation (QVT) [59] and many others. The concept of M2M is to specify the language transformations based on the language meta-model, within a meta-modeling

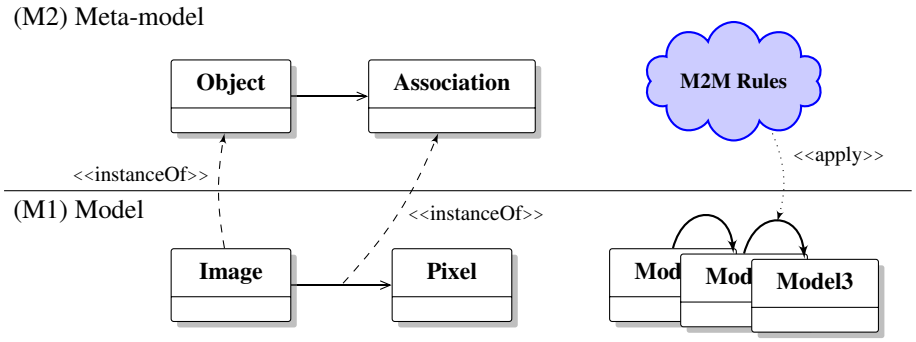


FIGURE 4.2: Model-to-Model transformations

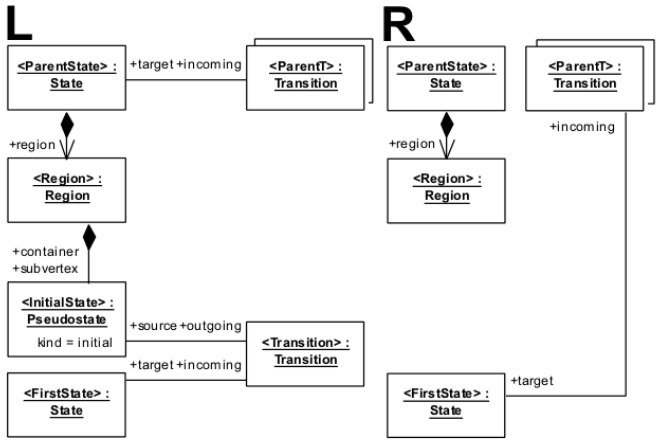


FIGURE 4.3: Moving input incoming transitions to the initial substate (Source: [7])

framework. Figure 4.2 sketches the approach. Given a meta-model, we encode model transformations using a particular Domain-Specific Language (DSL) by directly referencing meta-model elements. The rules are then instantiated and applied at the model level.

In [7], Schattkowsky and Muller propose a set of rewriting rules in a M2M setting for a subset of the UML Statecharts, called *Executable State Machines*. We pointed out in Chapter 3 that UML Statecharts are independent of the underlying action language. Therefore, their proposed transformations are only related to the state machine structure, such as move entry/exit activities to input/output transitions, resolve conflicting triggers along the hierarchy, among others. For instance, Figure 4.3 shows the formal specification of a transformation that, given any composite state (ParentState), moves its incoming transitions (ParentT) to the state pointed by the initial pseudostate (FirstState). If not formally shown, the set of rules allow the authors to flatten the hierarchical state machine for direct execution. Even though the set of rewritting rules are generic, they lack of a concrete action language and cannot reason about the actions to be performed.

The implementation of the synchronous visual formalism SCCharts, reviewed in Chapter 3,

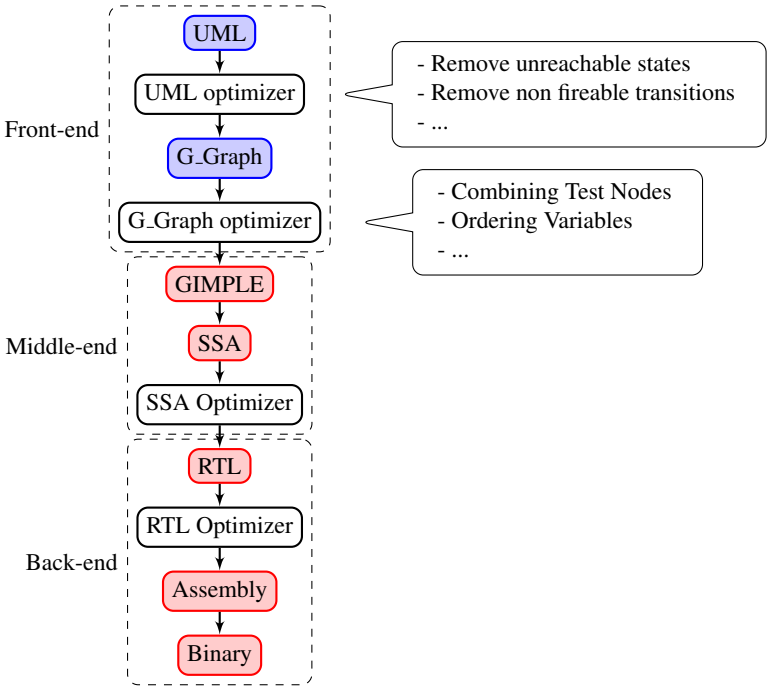


FIGURE 4.4: GUML code generation flow

seems to have a corresponding meta-model according to [6]. Using the Eclipse Metamodeling Framework (EMF) [60], they write model-to-model transformations in plain Xtend language [61].[1] We discuss in detail its compilation process in Section 4.3.

4.1.2 Optimizing Beyond the Back-end Language

In [27], Charfi et al. pointed out the recurrent problem of modeling frameworks concerning the gap between models and code production. A concrete specification of a convenient 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 specializations, necessary to meet performance requirements of the given models.

To overcome the problem, Charfi et al. pass semantical information about the input model to the compiler, thus enabling further optimizations such as elimination of unreachables states, condition combining, etc.. Using a new representation called *GUML*, information concerning the structure of the original state machine is passed down to the C compiler via embedded GIMPLE nodes. GIMPLE is a language-independent tree representation used in GCC for SSA-based optimizations [62]. The approach enables high-level optimizations inside the intermediate representation of a compiler. Figure 4.4 show the compilation flow as proposed in [27]. They

[1] Xtend is a flexible and expressive language that compiles into Java code.

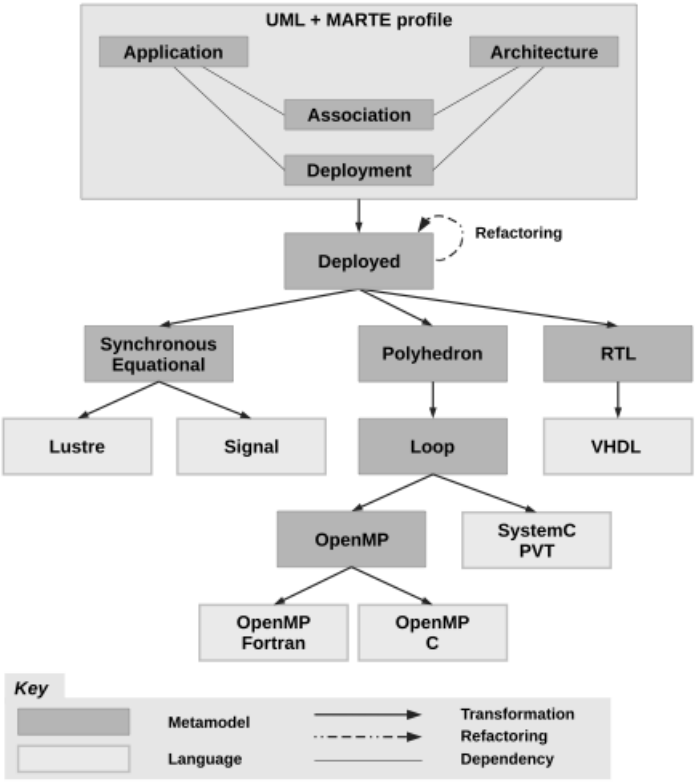


FIGURE 4.5: GASPARD internal transformations (Source [8])

also show encouraging results with respect to assembly code size comparing three different industrial tools under different code generation patterns.

4.2 Domain Specific Languages Supporting Model Based Design

The GASPARD modeling framework is the closest work to $\langle \text{HOE} \rangle^2$ we found in the literature [8]. They propose a combination of MARTE for the modeling of embedded systems and the Array Oriented Language (Array-OL) [5, 36], which offers generation of efficient code for data-intensive applications.

Figure 4.5 shows the transformation flow of GASPARD where we precisely see different models for Embedded System (ES) co-design. We remark multiple underlying representations generated from the final Platform-Specific Model (PSM) model. However, Array-OL is the chosen one for the modeling of massively parallel computations.

GASPARD provides a set of modeling concepts for the specification of Array-OL applications within the MARTE UML profile. They showed interesting results of an unified modeling of modern platforms (GPGPU) and video processing algorithms (H-263) [17]. However, they focused on the data-driven part of the algorithm (filtering) because Array-OL does not support

control dependent flow, as shown in Chapter 2. Therefore, they have to mix different formalisms for control and data-driven components. Array-OL is mainly a stream-oriented language, known to have good code generation properties [63], although not a Statechart-like formalism.

We also found similar combinations in the world of synchronous languages [28]. Alras et al. pointed out the innefficient code generation process from models and tried to address this issue by supporting Model Based Design (MBD) with an intermediate representation based on LUSTRE [64], which has a strong background on code generation methods.

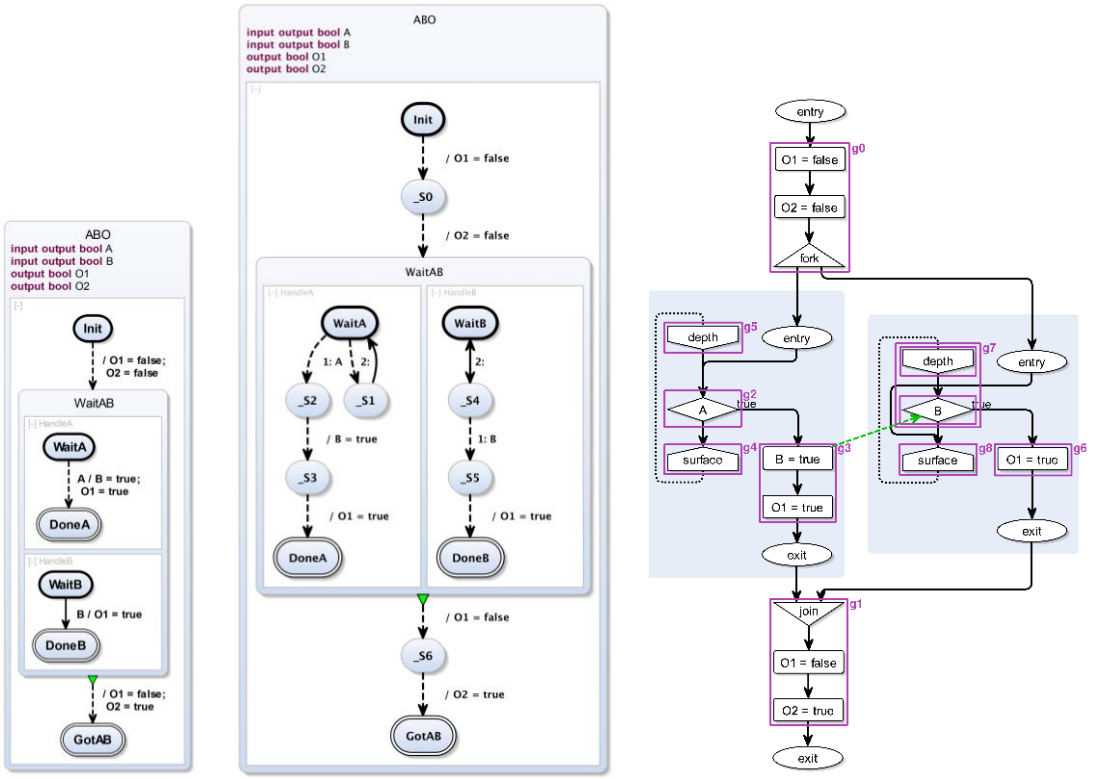
4.3 Synchronous Statecharts Compilation

In contrast to UML, synchronous Statecharts are strongly guided by a concrete language implementation (and its semantics as well). The code generation of synchronous languages is a well-known subject in the research community since the LUSTRE and ESTEREL programming languages [65].

SCCharts [6], presented in Chapter 3, is a safe imperative extension to synchronous languages with a Statechart-like visual representation. The compilation of SCCharts is decoupled into different intermediate representations:

- Extended SCChart: the front-end language.
- Normalized SCChart: the extended SCChart is reduced through a series of model-to-model transformations to primitive transitions only, which facilitates the mapping to the following representation.
- SC Graph: It is a directed graph that captures control dependencies in the form of a graph of basic blocks as well as data dependencies between parallel statements.

We show in Figure 4.6 the SCChart representations. For instance, note that the normalized one split transitions with more than one sequential statement into multiple states (transition $Init \rightarrow WaitAB$). As explained above, the SC Graph decouples the normalized SCChart into basic blocks or "scheduling units". The green arrow denotes a concurrent data dependency. In order to guarantee a deterministic behavior, writes are scheduled before reads, which split g_7 into two different scheduling units. The SC Graph relies on the SCL language introduced in Chapter 3 and it is from this language that the authors perform the software and hardware synthesis. SCL and its respective dataflow view make hardware synthesis almost straighforward. However, they need extra software support to generate the "tick" function that computes each signal following the chosen scheduling. Indeed, there is another representation slightly different from SCL called SCL_P, which consists of a subset of the sequential C core.



(a) Extended SC-Chart (b) Normalized SCChart (c) The SC Graph

FIGURE 4.6: SCChart representations during compilation (Source: [6])

Parallel regions produce concurrent threads of execution which are statically scheduled into a single C function. The scheduling process generate a schedule table that stores the continuations and is indexed by fixed thread priorities. It is a very clever strategy that allows to manage the switching between thread actions in order to satisfy data and control flow constraints of the Model of Computation (MoC) and to implement join/fork semantics by dynamically changing threads priorities.

4.4 Conclusions

Current MDD frameworks based on UML Statecharts generally focus their code generation strategies for simulation purposes and they are known for the faithfulness of the initial model with respect to the generated code. Thus, it is not intended to consitute a final code production process. On this direction, MDD frameworks use Object-Oriented Programming (OOP) code generation techniques to improve readability of the generated code and provide a rapid software architecture to work further on. Looking for performance, recent works try to fill the gap between modeling techniques and final efficient generated code. Existing approaches take into account high-level semantics all along the compilation flow.

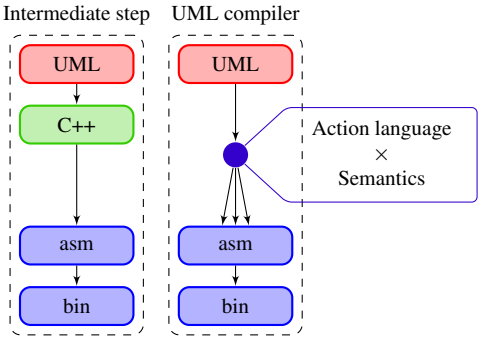


FIGURE 4.7: Recent compilation of UML models with Statecharts

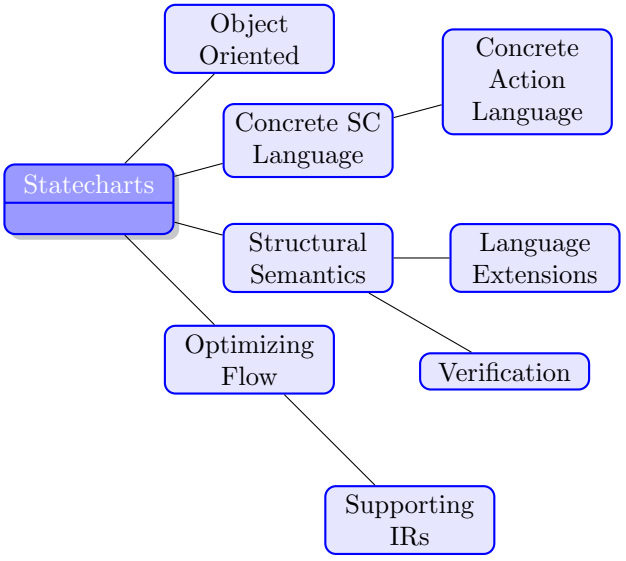


FIGURE 4.8: Aspects of a Statechart based approach

Figure 4.7 shows the evolution of the UML compiling flow. In the context of UML Statecharts, we highlight two main drawbacks: the lack of a concrete action language and the multiple semantical variation points of Statecharts. To cope with the former, an initiative for the specification of a concrete syntax from UML exists [66]. However, it does not specify Statecharts notations. Meanwhile, industrial tools mixed foreign languages such as C/C++ or Java inside transition actions, with UML providing an object-oriented context. Reviewed research work built upon C/C++ as the main action language and adapted compliant compilers (gcc) to host and to optimize Statechart based applications.

Unfortunately, the compilation of UML Statecharts faces a number of open questions related to their semantics. As shown in Chapter 3, it led to multiple and complex formalizations and it will necessarily lead to different compiler implementations.[2]

The important semantical gap between high-level models and its supporting languages (C/C++) calls for intermediate representations to further optimize the model regarding different aspects:

[2]We observe a similar issue between different C/C++ compilers due to undefined behaviors in the language specification.

communications, computations and data structuring. Different works propose theirs internal representations (GUML, SCCharts/SCL, Lustre, Array-OL) based on which many strong optimizations can be achieved. However, they are either too specific and tied to a particular low-level language (GUML) or have limited expressiveness (SCCharts, Lustre, Array-OL).

To summarize, Figure 4.8 shows all different aspects studied so far. From languages that allow the modeling of parallel computations (Chapter 2), the semantics of Statecharts (Chapter 3) and its optimization and compilation methods, we conclude that a well-founded Statecharts formalism should provide

- Modularity: The widely known object-oriented paradigm is well-suited to structure the communication between different Statecharts and provide a explicit context to the state machine. UML has strong foundations on this matter.
- A concrete language implementation with parallel actions: In order to be as precise as possible and to build the foundations for a structured semantics, we need an expressive action language to that can allow us to model parallel operations.
- A structured formal semantics: on the grounds of a concrete language, a structured semantics should enable a simpler and scalable specification.
- An intermediate representation for optimization and code generation: if we look for performance, an intermediate representation more expressive than previous works exposing communications, computations and type definitions is necessary.

Part II

Proposition

Chapter 5

$\langle$HOE$\rangle^2$ Language

Contents

In order to fill the gap between the modeling objects and the expression of data-intensive and parallel computations, we introduce the Highly Heterogeneous, Object-Oriented, Efficient Engineering $\langle$HOE$\rangle^2$ language. In $\langle$HOE$\rangle^2$, we take advantage of hierarchical representations with multi-valued associations to expose deep nested parallelism. Such parallelism is inherently available at the model structure but often not taken into account to represent primitive data.

We designed $\langle$HOE$\rangle^2$ to be able to express, in an unified manner, fine-grain hierarchical parallelism relying on the structural parallelism already built-in into the model. With data-intensive applications in mind, we also looked to provide a path to the generation of efficient code by introducing a set of behavioral abstractions amenable for analysis and optimizations.

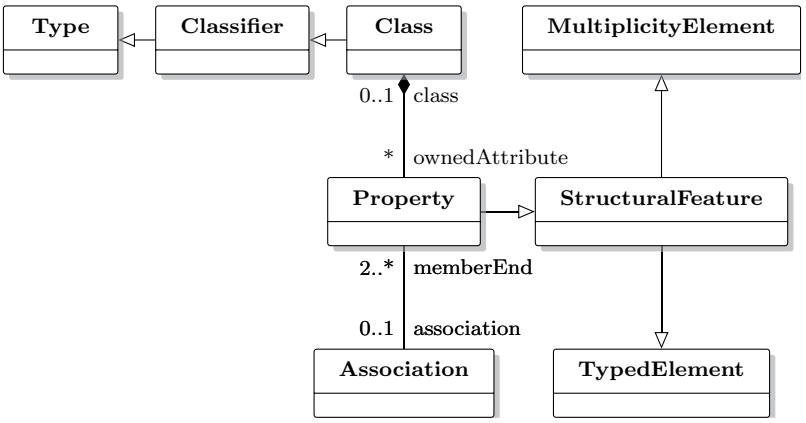
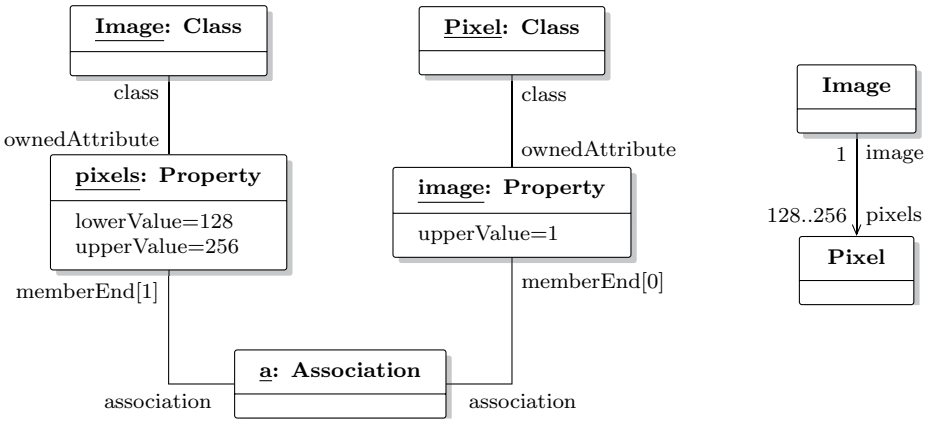


FIGURE 5.1: UML class and associations



(a) UML instance (b) Model view

FIGURE 5.2: Image object model

We present the general structure of the ⟨HOE⟩^{2} language, its Statecharts, and, most importantly, the action language. We also recall some of the important features of the Unified Modeling Language (UML) class meta-model and its Hierarchical State Machine (HSM) definition.

5.1 Objects

The ⟨HOE⟩^{2} language borrows some of the important concepts from the widely-known modeling language UML. More precisely the concept of classes, associations, multiplicities and HSM.

Figure 5.1 shows a simplified meta-model of the concept *class* defined in UML 2.4 [21]. A class contains an indeterminate number of (structural) *properties*. Two properties are linked via an *association*, which at the model level is graphically represented by a straight line connecting two classes. In addition, properties inherit multiplicity information that allows to constrain the number of linked instances. In simpler words, we are able to declare arrays of different sizes. It is a particularity of UML-like modeling language that we will introduce later.

```
object Image
  has [128..256]Pixel as pixels
object Pixel
  has Image as image
```

LISTING 5.1: Image model in ⟨HOE⟩^{2}

For instance, Figure 5.2 shows an image model as an instance of the UML meta-model 5.2(a). The `Image` instance contains a property called `pixels` allowing between 128 and 256 `Pixel` instances. By reusing graphical notations of meta-models, we draw the particular instance as shown in Figure 5.2(b). The meta-model of Figure 5.1 may be concretized by means of a grammar definition where each object represents a grammar rule. The key observation is that the inheritance relation denotes alternation and a group of associations denote concatenation. The *language* (or textual) representation of meta-models translates classes into record type constructors with properties representing named and typed fields inside the record. The intermediate concept of association is folded in both properties, becoming just a type reference. Following the concrete syntax of ⟨HOE⟩^{2} (detailed in appendix A), we write down the equivalent program of Figure 5.2(b) as presented in Listing 5.1.

Note that we use the keyword **`object`** instead of **`class`**. Formally, objects are defined as follows

⟨*object*⟩ ::= '**`object`**' ⟨*ID*⟩ ⟨*interface*⟩* ⟨*association*⟩* ⟨*SM*⟩

It specifies a set of interface entries, a set of associations and a HSM. The allowed types of associations are either already defined objects or *containers*. A container is an ordered sequence of elements that allows duplicates. In the context of UML, the meaning of order is given with respect to the element position, i.e., independently of the implemented order policy. An element A is less than A' if A is located at a lower index position than A' in the container. Whether we have lists or arrays is implementation specific and it is automatically decided at code production time.

In order to support multi-dimensional containers, we extend the multiplicity specification of UML with comma-separated list of ranges described by the following grammar:

⟨*T*⟩ ::= ⟨*t*⟩ | '`[`' ⟨*Range*⟩ '`]`' ⟨*t*⟩

⟨*Range*⟩ ::= ⟨*Range*⟩ '`,`' ⟨*Range*⟩ | ⟨*INT*⟩ | ⟨*INT*⟩ '`..`' ⟨*INT*⟩ | ⟨*INT*⟩ '`..`' '`*`' | '`*`'

where ⟨*t*⟩ is a user-defined object type. The four leaf cases of ⟨*Range*⟩ have the usual meaning

1. Container with fixed size, e.g. `[256]T`.
2. Container size has known lower and upper bound: `[4..6]T`.

```
object Pixel
  interface
    ins  GetU(), GetV()
    outs TakeU(Float), TakeV(Float)
    on   GetY() -> TakeY(Float)

  has Float as r, g, b
  has Float as y, u, v
```

LISTING 5.2: Interface entries of Pixel

3. Fixed lower bound and unknown upper bound: `[4..*]T`.

4. Unbounded number of elements: `[*]T`.

5.1.1 Interface

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 its interface definition. From the point of view of an object, the interface specifies the set of valid incoming and 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 related to the message exchanges needed to fulfill the interface definition is not seen by the external users.

Listing 5.2 shows the definition of three accepted input messages `getY`, `getU` and `getV`, and three output messages `takeY`, `TakeU` and `takeV`. Input-output relations can also be defined. For instance, `takeY` is declared to be a consequence of the reception of `getY`. They are implicitly interpreted as input and output messages, respectively, and do not need to be in the set of **`ins`** and **`outs`**.

5.1.2 Imports

In order to improve the language modularity, it is possible to organize objects by modules. A module consists of a set of objects grouped in a single file. The module name is taken from the file name. Other object may import the module using the keyword **`import`** followed by the list of desired modules.

5.2 Hierarchical State Machines

The Model of Computation (MoC) is based on UML HSMs [21], inspired from the Harel's Statecharts [20]. UML provides the HSM structure shown in Figure 5.3. It is composed by one

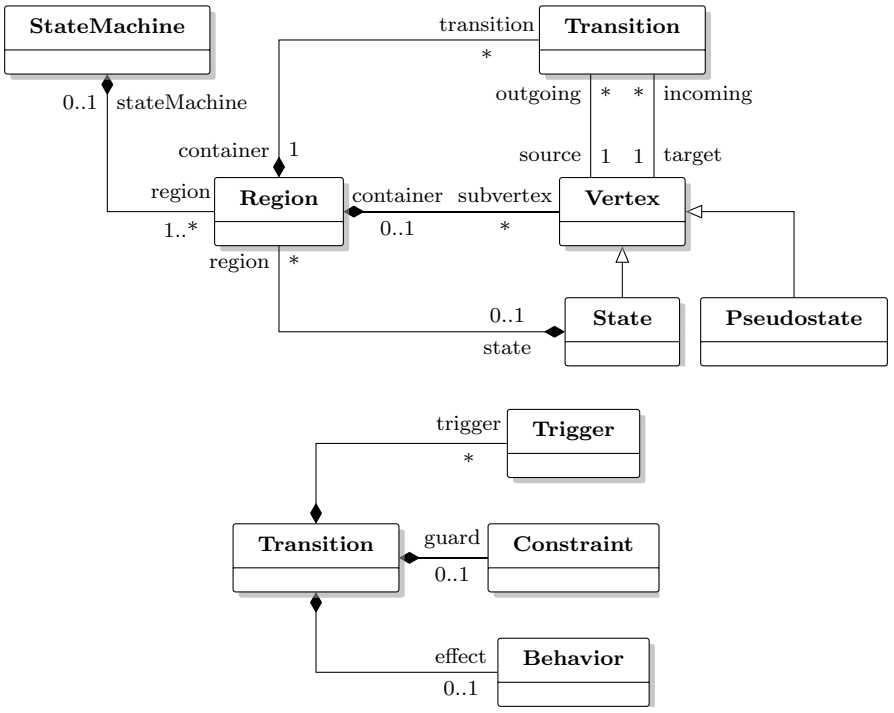


FIGURE 5.3: Meta-model of UML Hierarchical State Machine

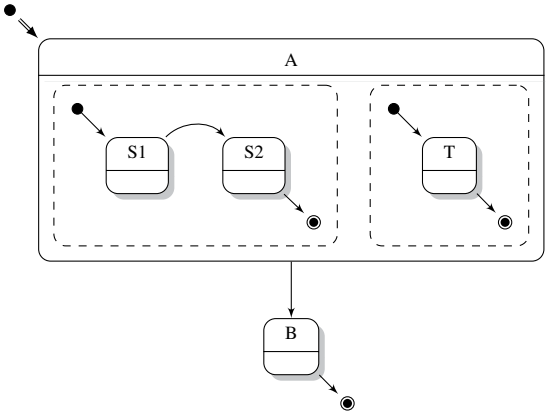


FIGURE 5.4: Graphical notation of a HSM

or more parallel regions. A region contains a set of states and transitions. The state is called *composite* if it contains at least one region. From Vertex, it inherits incoming and outgoing transitions that builds up the state machine structure. Pseudostates model, among other meanings, initial and final states.

The transition has three main components: zero or more triggers, a guard condition and its effect. The trigger specifies a list of valid messages to wait for and if the condition guard is true, the effect is executed. We will discuss about the semantic variations in Chapter 6. The effect is defined as a Behavior, which may be an opaque type so that the user is free to choose its preferred language. The behavior specification is known as the *Action Language*, which is one of our main contributions: the ⟨HOE⟩^{2} action language.

```
sm SM.
  // Creator
  creator c() to A
  // Composite state
  cstate A. // First region
            region
              initial S1
              state S1. on m1() to S2
              state S2. endon m2()
            endregion
            // Second region
            region
              initial T
              state T. endon m3()
            endregion
            // Outgoing transition from composite state A
            on m4() to B
  // Simple state with final transition
  state B. endon m4()
```

LISTING 5.3: HSM syntax in ⟨HOE⟩^{2}

Together with a new syntax to describe HSM, we introduce a new parallel action language to exploit the inherent parallelism of *multi-valued* associations in a data-parallel setting.[1] Before going into the details of our proposition, we present the general structure of the ⟨HOE⟩^{2} HSM and highlight particular differences with respect to the UML version. Figure 5.4 shows an example of a HSM and Listing 5.3 its corresponding ⟨HOE⟩^{2} program.

In contrast to UML, we defined two different concepts for simple and composite states represented by the keywords **`state`** and **`cstate`**, respectively, with transitions folded into the source state to improve code source readability. The simple state has an identifier and is followed by a list of outgoing transitions. In addition to outgoing transitions, composite states contain parallel regions (two regions in the presented example). Parallel regions contain in turn other states, building up the state machine hierarchy.

They are classified as follows:

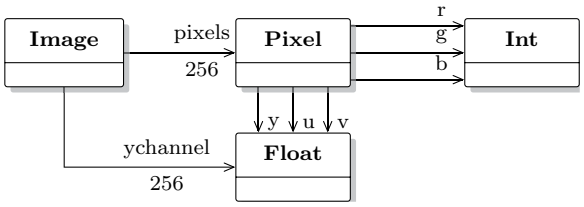
Creation. Creation transitions provide a method to create object instances and determine the entry point of execution. They are similar to constructors in Object-Oriented Programming (OOP). In the same sense, they may take parameters. We allow the specification of creation transitions only at the top level of the state machine.

External. Standard transitions.

Initial. This transition indicates the initial state of regions.

Final. The final transition marks the end of its execution context, which is either a state machine or a region.

[1] A multi-valued association is an association with a multiplicity specification greater than one.



(a) Image model

```
object Image
  has [256]Pixel as pixels
  has [256]Float as ychannel
  has [256]Float as quant

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

(b) Image code

FIGURE 5.5: Image model

```
on <triggers> [ <guard> ] / <updates> : <send messages>
```

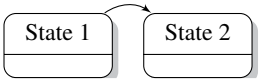


FIGURE 5.6: Structure of the ⟨HOE⟩^{2} action language

Therefore, the object contains a state machine with a default constructor or entry point using the creation transition (see incoming transition of Figure 5.4). The object defines the execution context of the state machine.

Figure 5.5 shows an enriched image model and its corresponding ⟨HOE⟩^{2} program on which we will develop all our examples troughout this chapter in order to show the main features of ⟨HOE⟩^{2} language.

In the following sections, we present the particularities of the ⟨HOE⟩^{2} action language.

5.3 Modeling Arithmetics

The action language separates arithmetic operations into three basic actions: (1) start computation, (2) receive the result and (3) store it. Point (1) represents a send primitive, (2) denotes a receive and (3) an update or assignment. Whereas the former two points are intrinsic to the state machine semantics, the latter is the additional most basic action to be defined. Figure 5.6 shows the structure of the proposed action language. We define the action, or behavior in UML terms, as a sequential composition of updates and sending primitives, separated by the token ':'. Receive and guard specifications are defined at the beginning of the transition followed by the keyword **on** and enclosed in square brackets, respectively.

```
state A. on / : r.add(g) to B
state B. on added(v: Int) / b = v
```

LISTING 5.4: Addition

```
    state GETY.
      on /: { i: 0..pixels.len - 1 } pixels[i].getY()
      to GETTING_Y
```

LISTING 5.5: Data parallel broadcasting

Listing 5.4 shows the arithmetic operation $b = r + g$ on the object model of Figure 5.5(a), decoupled as a combination of our three basic actions where the dot notation means message sending.

Messages `add(Int)` and `added(Int)` are part of the object specification as we will see later. They model the actual addition with `r` being the receiver of message `add(Int)` sent by a `Pixel` object with parameter `g`.

5.4 Parallel Actions

As pointed out in Chapter 2, the *data parallel* approach is a promising technique to make efficient use of parallel hardware. We will show that its more elaborated extension called *Nested-Data Parallelism* can be represented in a model-driven manner. The implementation of such paradigm relies on a new data structure called *parallel array* in the context of Haskell Data-Parallel (HDP) [9]. Similar to $\langle \text{HOE} \rangle^2$ objects, parallel arrays are composable to allow the expression of hierarchical or nested data parallelism.

The data parallel approach is about performing the same operation on a set of grouped elements, which in our context it is called *broadcasting*. We take advantage of multi-valued associations to define data parallel operations using *index domains*, defined between braces. Based on the model example of Figure 5.5, Listing 5.5 shows the luminance computation of all pixels in a data parallel fashion.

The above example introduces three new concepts: *indices*, *index domains* and *indexed messages*. The definition is composed by a list of indexes and a set of *constraints*. The domain is equivalent to the closed range constraint $0 \leq i \leq pixels.len - 1$. The sending expression inside an index domain instantiates a set of indexed messages where each index value is taken from its enclosing domain.

```
state GETTING_Y.
  on takeY{i}(y: Float) /
       ychannel[i] = y to GETTING_Y
```

LISTING 5.6: Reception of indexed messages

```
state GETTING_Y.
  on takeY{i}(y: Float) /
       ychannel[i] = y to GETTING_Y
  endon [i.all]
```

LISTING 5.7: Broadcast completion

```
state A. on / : r.add(cstr), g.mult(cstg), b.mult(cstb) to B
state B. on multed(v1: Int), multed(v2: Int), multed(v3: Int)
           / r = v1, g = v2, b = v3
```

LISTING 5.8: Parallel multiplications

In order to complete the parallel operation, the answers triggered by a data parallel broadcasting need to be captured precisely. Therefore, messages have an attached index value which is inspected using the notation presented in Listing 5.6.

Above listing shows the reception of an indexed message where, instead of passing index values around as message parameters, we expose indexes as a special field in the message structure. The advantage of this approach is twofold:

- Given that the index domain concerns only the sender, the receiver does not need to know where it is stored in the sender context.
- As we will see in Chapter 8, we can analyze index sets to optimize the state machine and to produce efficient code.

After the reception of all indexed messages, we need to indicate the completion condition. Listing 5.7 shows the syntax that denotes broadcast termination. Using one the index variables, which has a corresponding index domain, we denote the condition "all messages have been received".

In addition to index domains with parallel semantics, we extend the language with parallel composition using the comma separator.

Listing 5.8 shows three parallel sending actions, denoting multiplication. Its respective answers are captured at state B and stored in parallel.

Combining object composition, data-parallel broadcasting and parallel composition of action expressions, we obtain a modular implementation of nested data parallelism paradigm on the

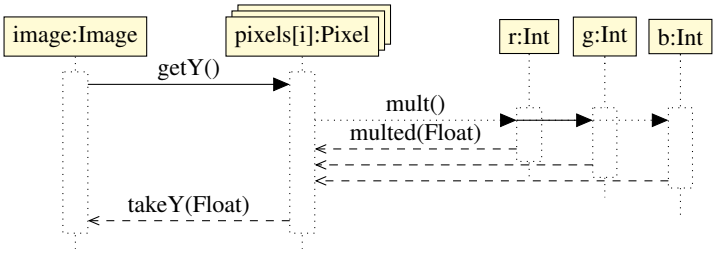


FIGURE 5.7: Nested parallel operations

context of modeling languages. The diagram of Figure 5.7 illustrates the nesting of sending actions. The Image sends `getY` indexed messages to `pixels[i]` for all $i \in [0..255]$. For each `Pixel` object, `getY` triggers three parallel send of `mult(Float)` messages. Therefore, parallelism is nested in a modular fashion.

Index Domains. In the context of broadcasting, we briefly introduced in the previous section the concept of index domains. They are composed by a list of iterators together with a constraint formula (see complete definition in Appendix A). Therefore, we can specify more elaborated iteration domains such as

- Modulo conditions: `{ i: i % 2 = 0 }`
- Column/row selection: `{ i, j: i = 3 and 0 <= j <= 255 }`
- Upper/lower triangle of a two dimensional association:
  `{ i, j: 0 <= i < j and 0 <= j <= 255 }`

We can iterate on both $\langle \text{HOE} \rangle^2$ actions, updates and sending. Iterators in sending actions have additional semantics. The defined index variables can be used later in receive expressions so that the same domain applies there. As shown in Listing 5.6, the index domain of `i` defined at state A remains the same at the receive clause of state B.

We limit index constraints to the representation of affine expressions. That is, constraints are defined over the set of presburger formulas [67].

Association Slicing. The action language supports slicing of associations. The programmer can specify a closed range to select a list of values from a multi-valued associations in a single expression. As an example, consider the `Image` creator of Listing 5.9.

The `Image` creates `pixel.len` pixels using creator `pixel` and passing it a slice of the association `rawImage` containing three values.

```
creator raw(rawImage: Int[768]) /
        { i: 0..255 } pixels[i] = new Pixel.RGB(rawImage[3*i..3*i+2])
        to GETY
```

LISTING 5.9: Association slicing of three integers

```
1  object Pixel
2    interface
3      on getY() -> takeY(Float)
4
5    has Int    as r, g, b
6    has Float  as y, u, v
7    sm PixelSM.
8      creator RGB(rgb: Int[3]) / r = rgb[0]
9                               , g = rgb[1]
10                              , b = rgb[2]
11                              to ComputeY
12     // Wait for getY and launch multiplications
13     state ComputeY. on getY() / : r.mult(0.299)
14                                 , g.mult(0.587)
15                                 , b.mult(0.114)
16                     to Multing
17     // Collect two multiplications and launch the addition
18     state Multing. on multed(v1: Float), multed(v2: Float) / : v1.fadd(v2)
19                    to MultAdding
20     // Collect last multiplications and launch another addition
21     state MultAdding. on multed(v3: Float), fadded(v4: Float) / : v3.add(v4)
22                       to Adding
23     // Reply to the sender of getY() by using keyword "initiator"
24     state Adding. on fadded(result: Float) / y = result: initiator.takeY(y)
25                   to ComputeY
```

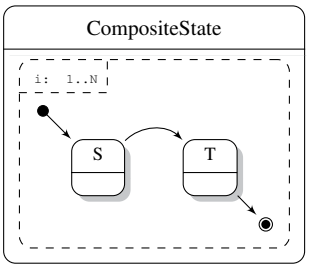
LISTING 5.10: Reply on initiator

5.5 Initiators

In section 5.4, we presented how to handle replies triggered by specific messages such as `getY` or `Add` and `mult(Int)` but we did not show how the corresponding replies were generated.

In ⟨HOE⟩^{2}, the receiver may send replies using the keyword **`initiator`**. For instance, consider the complete implementation of `Pixel` where the use of initiator is highlighted at line 24 of Listing 5.10.

The initiator allows objects to send replies without knowing the actual target. The initiator value is directly related to the interface definition. Only the input messages exposed by the interface will set the initiator variable with the corresponding sender. For instance, given the relation exposed in the interface of `Pixel`, we see that initiator corresponds to the sender of message `getY` at line 32. This setting is clearly control flow dependent. The compilation process via our intermediate representation, presented in Chapter 7, makes definitions explicit.




```
cstate CompositeState.
  region { i: 1..N }
    initial S
    state S. on / to T
    state T. endon /
  endregion
```

FIGURE 5.8: $\langle$HOE$\rangle^2$ indexed regions

5.6 Object Creation

Every object exposes a creation transition. For instance, consider the example of Listing 5.10 where `Pixel` defines a creation transition called `RGB` taking an association of three integers. Then, `Image` creates `Pixel` instances using the following syntax

```
pixels[i] = new Pixel.RGB(rgb[3*i..3*i+2])
```

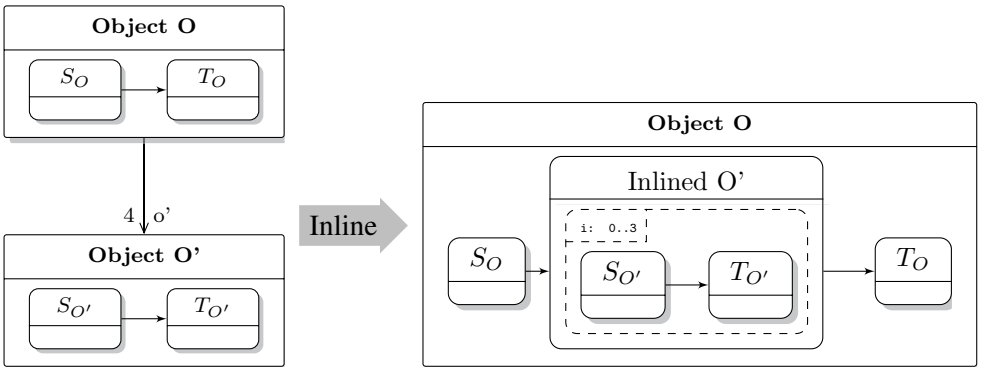
LISTING 5.11: Instantiating objects

5.7 Indexed Regions

Regions inside states represents parallel regions of code, modeled using state machines, hence the hierarchical composition. Whenever the state machine enters a composite state, it automatically jumps to the entry point of each of its internal regions.

We extend the HSM model with *indexed regions* where its graphical and textual notation is shown in Figure 5.8. It represents N parallel regions, each one indexed by i. The i-th region contains two states, S_i and T_i, where S_i is the initial one. Indexed regions model *forall* block expressions, found natively in most parallel languages or introduced as parallel extensions to sequential languages [9, 34].

We will show in Chapter 8 that the indexed region is a key concept to enable deep optimizations for efficient code generation, e.g. inlining of HSMs. As an illustration, consider objects O and O' related through a multi-valued association o' as shown in Figure 5.9(a). Under specific hypotheses for model and state machine transformations that we will introduce in detail in Chapter 8, we are able to inline object T into S featuring indexed regions as described by Figure 5.9(b). Note that the multiplicty range is compliant with the indexed region domain.



(a) Inlining pattern

(b) Inlined object using indexed regions

FIGURE 5.9: Inlining objects

5.8 Scalars

Every language provides a set of scalars as built-in types, e.g. **`int`** or **`float`** in C/C++ or Java. In addition, pure functional languages like Haskell allow us to represent algebraic properties on scalars. They share an intrinsic property across all programming languages: *immutability*. Furthermore, scalars usually denote the carrier set of some algebraic structure. Lots of compiler optimizations are built upon properties of scalars, e.g. constant propagation, strength reduction or value numbering [68].

On the other hand, the behavior of built-in types strongly influence 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 [69]. 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 [70].

Data-flow languages introduce another interesting example of built-in types [71, 72]. In the Kahn denotational semantics of data-flow languages [73], 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 allow 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

```
scalar Int
  interface
    on Add(Int)  -> Added(Int)  ~> AddOp
    on Mult(Int) -> Multed(Int) ~> MultOp
  model Z with (+): AddOp, (*): MultOp
```

LISTING 5.12: Integer scalar

```
scalar Float
  interface
    on Div(Float) -> Dived(Float) or DivedByZero() ~> DivOp
```

LISTING 5.13: Float scalar

setting, scalar values follow the same semantics as generic immutable objects. This has a lot of advantages in modeling languages.

For instance, consider again the state machine of `Pixel` object shown in listing 5.10. It extracts the luminance information from a `Pixel` in RGB format interacting with scalars. Note that interactions with scalars, at `ComputeY` for instance, are explicitly parallel and equivalent to generic objects from the modeling perspective. 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.

The scalar defines operations in the form of message exchanges. It has an underlying carrier set that provides an intrinsic meaning to the type and its operations in the form of message exchanges. For instance, consider the `Int` scalar of Listing 5.12. The denotation of scalar `Int` is the set $\mathbb{Z}$, which is built-in in the compiler and equivalent to $\mathbb{Z}$, together with the denotation of message exchanges (addition and multiplication). The interface has a new kind of entry that we call *operational transitions*. Int has two operational transitions `AddOp` and `MultOp`. They model the beginning of an operation and its ending in an asynchronous manner.

Besides a clean integration on the modeling framework, this representation has interesting implications. We will show in Chapter 8 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. Formally, one may simply think of `AddOp` as the function $AddOp : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$.

As another example, see the interface entry of scalar Float in Listing 5.13.

`DivOp` operator is a partial function (not defined when dividing by zero) but we can make it total by taking $DivOp : \mathbb{R} \times \mathbb{R} \to (Dived + DivedByZero)$, where $Dived \simeq \mathbb{R}$ and $DivedByZero$ models the return value on error condition. Therefore, message replies become values of the Scalar type being defined.

```
scalar Float
  interface
    on Div(Float) -> Dived(Float)
                  or NaN()
                  or Inf()
                  or nInf() ~> DivOp
```

LISTING 5.16: Extended Float scalar

For instance, let `f1` and `f2` be two Float scalars and, following $\langle$HOE$\rangle^2$ notations, the send action `f1.div(f2)`. Then, we should have the following transitions to collect all possible answers.

```
state Divided.
  on dived(f: Float) to NextState
  on divedByZero()   to ErrorState
```

LISTING 5.14: Message operations

Now, suppose we are able to find that send and receive expressions are actually related and that a language construct **`applyon`** which takes an object, an operation and a variable number of object parameters exists. In that case, we do not need to use message passing semantics and we can safely *apply* `DivOp` operator in-place as follows

```
f = applyon f1 DivOp f2
[...]
state Divided.
  [f is dived] to NextState
  [f is divedByZero] to ErrorState
```

LISTING 5.15: Inlined operation

where operator **`is`** would allow us to check whether the value of f has been properly divided or not. This construction can be seen as a pattern matching over set $Dived + DivedByZero$. The main difference is that f does not need to be projected in order to be used once it has been matched. If it is an error, it cannot not be used at any of the reachable states starting from `ErrorState` and if it is used, it is not an error. To see this claim, note that Listing 5.15 is the translation of Listing 5.14 after sends and receives have been folded. In the latter, no variable is defined when an error message is received and therefore, f does not exist at all states reachable from the second transition unless there is a path in the control flow which joins NextState and `ErrorState`, i.e., they share reachable states. If there is such a path and f is used, it must correspond to `dived(Float)` message.

Other interesting refinements can also be intuitively constructed based on this idea. For instance, the C implementation of floating-point numbers follow the standard IEEE754 [74] and can be translated as shown in Listing 5.16.

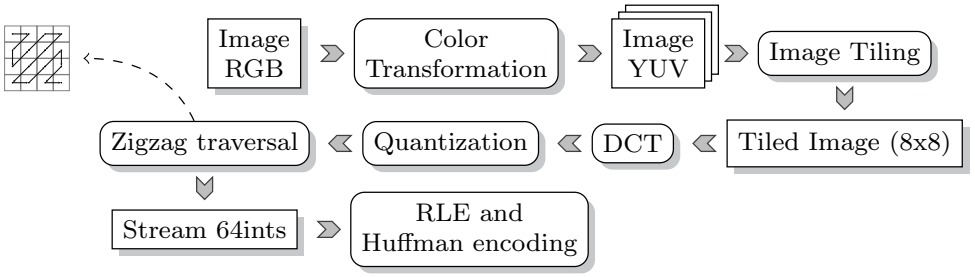


FIGURE 5.10: JPEG algorithm phases

Given that we may add new values (or reply types) to any scalar set and associate sum type as a return type of a given operator, could we have an operator such as $DivOp : \mathbb{R} + NaN + Inf \rightarrow \mathbb{R}$? The answer is no. The programmer cannot manipulate Inf or NaN values because there is no variable associated to them as they are the result of message matching at receive expressions — though we can send message named NaN if the scalar supports it.

In general, the operational transitions for any scalar T are of the form

⟨*optransition*⟩ ::= '**on**' ⟨*input*⟩ '**->**' ⟨*output*⟩ '**~>**' ⟨*opid*⟩

⟨*input*⟩ ::= ⟨*id*⟩ '**(**' ⟨*type*⟩* '**)**'

⟨*output*⟩ ::= ⟨*output*⟩ '**or**' ⟨*output*⟩ | ⟨*id*⟩ '**(**' [⟨*type*⟩] '**)**'

5.9 Modeling Applications

The set of proposed features allow us to model data intensive applications in an intuitive manner. In order to show the use of some of the proposed features, we consider the computational intensive flow of the JPEG compression algorithm shown in Figure 5.10.

Starting from an image on RGB format, the algorithm transforms it into the Luminance and Chrominance color metrics (YUV). The luminance values are then tiled with a tile size of 8x8. The Discrete Cosine Transform (DCT) is applied over all image blocks. Afterwards, the quantization phase divides each frequency value by specific values taken from an external quantization table. Finally, the Run Lenght Encoding (RLE) and Huffman encoding compress the resulting data, which is formatted into different sections to be stored.

Figure 5.11 shows the object model for the JPEG implementation of an image with 64x64 pixels. We briefly show here the proposed expressions of the ⟨HOE⟩^{2} language. We will cover it in depth at Chapter 9.

For instance, we can tile the image into 8x8 blocks via array slicing and launch parallel encoding using indexed messages after all `Block8x8` has been constructed. The tiling and encoding operations are implemented inside the `Image` state machine as follows

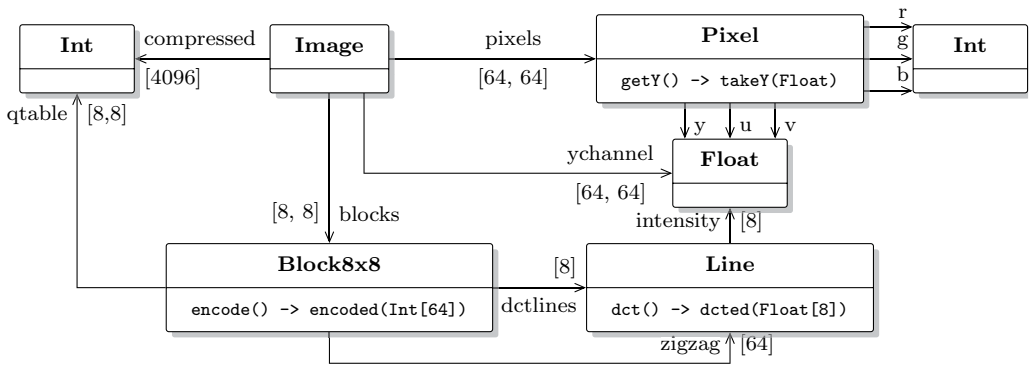


FIGURE 5.11: Image object model implementing JPEG

```
state Encode.
  on [] / {i, j: 0 <= i < 8 and 0 <= j < 8}
           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() // Parallel encoding
  to Encoding
```

After the encoded values has been received inside `Block8x8`, `Block8x8` launchs parallel divisions (quantization) relying on the scalar abstraction of ⟨HOE⟩^{2}.

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

As we can see, the parallel sending actions and the new scalar abstraction allow us to express parallelism in an uniform way.

5.10 Contributions

Following the structure of modeling languages, such as UML, we proposed several important features together with a new action language for the modeling of data-itensive applications. Our propositions can be classified into three categories: the action language, object related extensions and new state machine abstractions.

The ⟨HOE⟩^{2} action language. We proposed a minimal set of actions: update and send. Updates and sending actions are ordered sequentially where each one of them can be composed to form parallel expressions. Parallel expressions can be defined over an iteration domain and, more importantly, they allow the creation of indexed messages when it concerns sending actions. We also propose a well-defined reply action, the initiator semantics. Sending of messages

to the initiator object denotes a reply, which may be indexed if the initial message of the current transaction was also indexed.

We provide more expressive trigger specifications in order to capture indexed messages. Index values inside messages are closely related to the initiator semantics as it result from an implicit forwarding of index values from the initiating message. Therefore, we relate send and receive actions through specific index values.

We introduced join semantics using `all` conditions over specific indexes. Indeed, the index has an associated domain when defined for parallel sending actions. This condition allow us to wait for the broadcast to be completed.

More expressive models. We extended the multiplicity specification to support the modeling of n-dimensional associations. In addition to richer multiplicities, we defined an object interface that captures incoming and outcoming messages, together with precedence relations between them.

A key proposition is the $\langle$HOE$\rangle^2$ scalar. $\langle$HOE$\rangle^2$ scalars allows the programmer to consider scalar values as plain objects. The idea is followed by the modeling of simple arithmetics in terms of message passing. The combination of index domains to model parallel sending actions and the $\langle$HOE$\rangle^2$ scalar abstraction enabled the modeling of parallel operations.

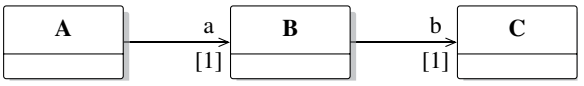
State machine abstractions. We extend the Statechart formalism with the concept of indexed regions. Indexed regions provide forall semantics to traditional regions via an indexed domain. It allows, among other things, the inlining of objects and fork/join fusioning, as we will see in the following chapters.

Chapter 6

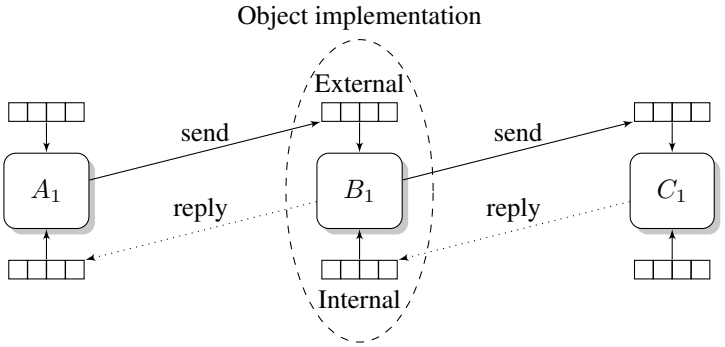
$\langle$HOE$\rangle^2$ Formal Semantics

Contents

In this chapter, we present the formalization of the Highly Heterogeneous, Object-Oriented, Efficient Engineering $\langle$HOE$\rangle^2$ language. We propose a hierarchical semantics decoupled into three operational parts: semantics of actions, transitions and configurations. We use a combination of structured operational [75] and denotational semantics techniques to handle non-determinism of parallel actions and meaning of single $\langle$HOE$\rangle^2$ statements, respectively. $\langle$HOE$\rangle^2$ semantics is strongly inspired from the informal semantics of Unified Modeling Language (UML) Statecharts [21].



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



(b) $\langle$HOE$\rangle^2$ Instance model with external and internal FIFOs

FIGURE 6.1: Functional view of communicating $\langle$HOE$\rangle^2$ objects

6.1 Introduction

Before entering into specific definitions, let us present the very general picture of communications among $\langle$HOE$\rangle^2$ objects. $\langle$HOE$\rangle^2$ is based on asynchronous state machines communicating through message passing. Each $\langle$HOE$\rangle^2$ object corresponds to a state machine implementing *external* and *internal* First In-First Out (FIFO) buffers. As mentioned in Chapter 5, the interface definition exposes "external" or visible messages to the object owners. The messages not shown at the interface level are not visible to the owners and hence they flow through an internal FIFO buffer. Figure 6.1 shows the proposed communication flow by expliciting external and internal FIFOs. In 6.1(a) we have a simple $\langle$HOE$\rangle^2$ object model and Figure 6.1(b) shows the communication flow of a given instance of such model. For instance, object B_1 will write to external C_1 FIFO using send primitives whereas all replies, triggered by `initiator`, from C_1 to B_1 will write to the internal B_1 FIFO. 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 message reception procedure will look at the 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.

In the following sections, we present required notations and definitions. Then we develop our hierarchical approach where the semantics of $\langle$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 6.2 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

$$\boxed{sys \in \mathcal{S}}$$

$$\Gamma_A \quad a \to a' \in \Gamma_A$$

$$\Gamma_T \quad \frac{a \to a' \in \Gamma_A \qquad t_1 \to t_1' \in \Gamma_T}{t_2 \to t_2' \in \Gamma_T}$$

$$\Gamma_K \quad \frac{t \to t' \in \Gamma_T \qquad k_1 \to k_1' \in \Gamma_K}{k_2 \to k_2' \in \Gamma_K}$$

FIGURE 6.2: Hierarchical semantics

separation allow us to deal with each particular problem, initiator semantics, message send/reception, message dropping policy, etc., mostly in isolation.

6.2 Domains

According to the $\langle$HOE$\rangle^2$ language presented in Chapter 5, the intuitive semantical domain is that of objects, references, array of references and indexes. Therefore, we define the following concepts:

- The finite set of references (or locations) $\mathcal{R}$, the undefined reference $Null$ together with the lifted version $\mathcal{R}_{Null}$.
- A store mapping references to object values $A = \bigcup_{R \subseteq \mathcal{R}} R \to O$.
- Indexes as integer values, $\mathbb{Z}$.
- The set of array values $\bar{A} = \bigcup_{a \in \mathbb{N}} [0, a) \to V$, where $[0, a)$ denotes the product of right-open interval and V the set of values.

Given above definitions, we construct the generalized set of values $V = \mathcal{R}_{Null} + \bar{A} + \mathbb{Z}$.

Definition 6.1. 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.

Definition 6.2. An object is defined as $O = (\langle sm \rangle, K, E, \Phi, I)$ where

- $\langle sm \rangle$ is the state machine.
- K is the current state machine configuration, which will be precisely defined later.

- E denotes the event list.
- $\Phi = \mathcal{V} \rightharpoonup V$ is a partial function corresponding to the object binding context where $\mathcal{V}$ is a set of variable identifiers.
- $I = \mathcal{R}_{Null}$ is the initiator reference.

Definition 6.3. The message $M = (\mathcal{R}, I, \mathcal{M}, \Psi, \mathbb{B})$ contains the sender reference, a sequence of index values $I = \overline{\mathbb{Z}}$, a valid message identifier, a mapping to its ordered parameter references $\Psi = \mathbb{N} \to V$ and a boolean value that is true if we have a reply message.[1]

Definition 6.4. We define the event pool $E = \overline{M}$ as a list of messages.

6.3 Semantics of Actions

Let us first introduce some useful notations.

- We note $sys^{[r \mapsto o]}$ the new system that maps reference r to object o and keeps everything else unchanged.
- We use α to range over the $\langle$HOE$\rangle^2$ syntax sets defined in Appendix A.1.
- List concatenation
- The function update is noted as $f' = [f \mid r' \mapsto o]$, which is equivalent to build a new function $f'(r) =$ if $r' = r$ then o else $f(r)$.
- We also define the conditional function update $f' = [f \mid r' \mapsto o]_p$ where $p \in \mathbb{B}$ is a condition such that $f' =$ if p then $[f \mid r' \mapsto o]$ else f
- We note projections with subscripts, e.g. A_{sys} is the mapping of system $sys \in \mathcal{S}$.
- The binding extension operator $\triangleright : \Phi \to \Phi \to \Phi$ is defined as

$$\phi \triangleright \phi' = \lambda v.\text{if } v \notin dom(\phi) \text{ then } \phi'(v) \text{ else } \phi(v)$$

  The implementation of $v \notin dom(\phi')$ is not shown here. For completeness, we model partial functions with functor $Partial(a) = a + Undefined$.[2]

- We will refer to an instance of a given syntax set S corresponding to the $\langle$HOE$\rangle^2$ grammar as α_S. For instance, α_{state} is an instance of $\langle$HOE$\rangle^2$.

[1] Reply messages resides in the internal object buffer, otherwise they belong to the external one. It is a particular implementation of our double FIFO communications.

[2] Note that $Partial \simeq Maybe$ using the well-known Haskell type.

- We use classical notation for lists $\overline{v} = [v_i]$ and note the concatenation of lists as $a \cdot b$.
- We note disjoint unions as $S = A + B$ and we use ι to describe injections into S. For instance, $\iota_A : A \to S$ and $\iota_B : B \to S$ represent the injections of S.

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

6.3.1 Sequential and Parallel Composition

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

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

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

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

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

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

$$\frac{(\alpha_{send}, \phi, sys) \to (\alpha'_{send}, \phi', sys')}{(\alpha_{send}, \ \alpha''_{send}, \phi, sys) \to (\alpha'_{send}, \ \alpha''_{send}, \phi', sys')} \text{ AParSL}$$

$$\frac{(\alpha_{send}, \phi, sys) \to (\alpha'_{send}, \phi', sys')}{(\alpha''_{send}, \ \alpha_{send}, \phi, sys) \to (\alpha''_{send}, \ \alpha'_{send}, \phi', sys')} \text{ AParSR}$$

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 [43].

$\langle$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 or it may also add new objects to the system, hence new references.

6.3.2 Update

We start by defining a denotational semantics of updates. Let us consider the simple one with the following evaluation rule

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

where we take a simpler definition of its grammar

⟨*supdate*⟩ ::= ⟨*var*⟩ '**=**' (⟨*var*⟩ | ⟨*new*⟩)

The left-hand side of updates, ⟨*var*⟩, are single or array variables while the right-hand side may also contain "new" expressions denoting object creation. We define $[\![]\!]_u : \langle update \rangle \rightarrow (\Phi \times \mathcal{S}) \rightarrow (\Phi \times \mathcal{S})$ for the particular case of single variables as

$$\begin{aligned}
[\![\mathbf{v} = \mathbf{v'}]\!]_u(\phi, sys) &= (\phi', sys^{[\hat{r} \mapsto \hat{o}']}) \\
&\text{where} \qquad r = (\phi \rhd \phi_{\hat{o}})(\mathbf{v'}) \\
&\qquad\quad\; \phi' = [\phi \mid \mathbf{v} \mapsto r]_{\mathbf{v} \in dom(\phi)} \\
&\qquad\quad\; \phi'_{\hat{o}} = [\phi_{\hat{o}} \mid \mathbf{v} \mapsto r]_{\mathbf{v} \notin dom(\phi)} \\
&\qquad\quad\; \hat{o}' = (sm_{\hat{o}}, k_{\hat{o}}, e_{\hat{o}}, \phi'_{\hat{o}})
\end{aligned}$$

We build binding contexts depending on scoping conditions. We obtain the reference value r and we update the local context ϕ iff the defining variable is local. Otherwise, we update the object binding context.[3] Then, we return a new local binding and system.

[3]Note that the conditional function update returns the same function if the condition is false.

As mentioned earlier, the **new** expression creates an object, and hence a new reference into the current system mapping. We define its denotation as follows

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

and $[\![]\!]_n : (\Phi \times \mathcal{S}) \rightarrow (\mathcal{R} \times \mathcal{S})$ creates a new object referenced by r' on sys', which is an extension of sys. We build the new bindings as before and update sys' with the updated object o. We detail hereafter the impliciations of the creation process.

Creation. As explained in Chapter 5, the creation transition points to the entry state of the state machine. We recall the definition of **new** and **creator**:

⟨*creator*⟩ ::= '**creator**' ⟨*id*⟩ '**(**' ⟨*param*⟩* '**)**' ['**/**' ⟨*update*⟩+] ⟨*to*⟩

⟨*new*⟩ ::= '**new**' ⟨*id*⟩ '**.**' ⟨*id*⟩ '**(**' ⟨*var*⟩* '**)**'

In order to create a new object following Definition 6.2, we have to define the following components of our initial object $o_0 = (\alpha_{sm}, k_0, e_0, \phi_0, r_0)$ where

1. The state machine implementation $\alpha_{sm} \in \langle sm \rangle$.
2. An initial configuration $k_0 \in K$
3. An empty event pool $e_0 = \epsilon$
4. An initial binding context $\phi_0(v)$, which initialize associations with the undefined reference $Null$.
5. The initiator is a null reference $r_0 = Null$

Note that $\alpha_{new} = (\alpha_{id}, \alpha'_{id}, \overline{\alpha}_{var})$ provides all these informations. The first identifier α_{id} corresponds to the object type to be created, from which we obtain the state machine implementation. The second one α'_{id} corresponds to the creator identifier to be evaluated. Finally, the input parameter list $\overline{\alpha}_{var}$ provide the local binding context to evaluate the update action (defined at the **creator** transition) according to the transition semantics defined over Γ_A.

The initial configuration k_0 concerns the state machine semantics, Γ_K, and will be introduced in Section 6.5. We will show precisely how to construct k_0 from the initial $\langle \text{HOE} \rangle^2$ state.

6.3.3 Send

Similar to updates, we define a denotational semantics of sends

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

where the grammar of simple sends is defined hereafter

⟨*ssend*⟩ ::= ⟨*var*⟩ ' **.** ' ⟨*msg*⟩

Let $push = \lambda o, m.(sm_o, k_o, e_o \cdot [m], \phi_o)$ be the function that pushes message m into object o, and $[\![]\!]_m : \langle msg \rangle \rightarrow \overline{\mathbb{Z}} \rightarrow \mathbb{B} \rightarrow (\Phi \rightarrow \mathcal{S}) \rightarrow M$ the function that takes a message definition, index values, a boolean indicating if it is a reply, a binding context and a system to give us a new message $m \in M$. Then, the denotation of send, $[\![]\!]_s : \langle send \rangle \rightarrow (\Phi \times \mathcal{S}) \rightarrow (\Phi \times \mathcal{S})$ which for the case of single variables is defined as

$$\begin{aligned} [\![\mathbf{v}.\alpha_{msg}]\!]_s(\phi, sys) &= (\phi, sys^{[r' \mapsto o']}) \\ &\text{where} \quad \phi' = \phi \rhd \phi_{\hat{o}} \\ &\qquad\quad r' = \phi'(\mathbf{v}) \\ &\qquad\quad o' = push(A_{sys}(r'), [\![\alpha_{msg}]\!]_m(\overline{0}, false, \phi', sys)) \end{aligned} \tag{6.1}$$

Unlike updates, sending actions will modify other objects in the system by pushing messages to their event pools. The message indexes are set to 0 because we are dealing with non-indexed sends. Notation $\overline{0}$ stands for a single element list of zeros.

Initiator. We also have an special case here, the initiator.

$$\begin{aligned} [\![\mathbf{initiator}.\alpha_{msg}]\!]_s(\phi, sys) &= (\phi', sys^{[r' \mapsto o']}) \\ &\text{where} \quad \phi' = \phi \rhd \phi_{\hat{o}} \\ &\qquad\quad o' = push(A_{sys}(I_{\hat{o}}), [\![\alpha_{msg}]\!]_m(\overline{0}, true, \phi', sys)) \end{aligned}$$

The sending action is done on the initiator reference $I_{\hat{o}}$ of the current object $\hat{o}$ and a reply message is instantiated.

6.3.4 Indexed Actions

Indexed actions concerns update or send expressions under a specific index set domain. For instance, consider the $\langle \text{HOE} \rangle^2$ syntax of indexed updates

$\langle iupdate \rangle$ ::= '{' $\langle indexset \rangle$ '}' $\langle supdate \rangle$

Instead of specify a denotational semantics similar to [76], we developed an operational view that takes advantage of the logical deduction on presburger formulas to derive a transition semantics.

Let $\mathbf{c} \in \langle int \rangle^n$ and $[\![]\!]_{ix} : \langle int \rangle \to \mathbb{Z}$ an vector of index constants and the denotation of index values, respectively. We also use the *alpha conversion* noted $\alpha_{indexset}[\mathtt{a}/\mathtt{b}]$, which replaces all occurrences of `a` by `b` into the $\langle \text{HOE} \rangle^2$ index set $\alpha_{indexset}$. In the following definition and in order to keep clear notations, we consider lifted versions of our index denotational function, alpha conversion and $\mapsto$ to n-dimensional sets.

$$\frac{\begin{array}{l}\alpha_{indexset}[\mathbf{i}/\mathbf{c}] \vdash \top \\ (\alpha_{update}, [\phi \mid \mathbf{i} \mapsto [\![\mathbf{c}]\!]_{ix}], sys) \to (\phi', sys') \\ \alpha_{indexset}, \mathbf{i} < \mathbf{c} \ \ \mathtt{or} \ \ \mathbf{c} < \mathbf{i} \vdash \alpha'_{indexset}\end{array}}{(\{\alpha_{indexset}\}\, \alpha_{update}, \phi, sys) \to (\{\alpha'_{indexset}\}\, \alpha_{update}, \phi', sys')} \text{ AForall}$$

The above rule gives an operational view of the forall expressions in the $\langle \text{HOE} \rangle^2$ language. Essentially, we choose a constant `c` that belongs to the index set, we bind the indexes to the constant and evaluate α_{update} under this new binding. After evaluation, the constant value should not belong to the set anymore. For this reason, we take advantage of the logical relation of presburger sets under new constraints to materialize the evaluation relation.

Indexed Messages. Indexed send expressions carry additional implications: indexed message instantiation. The surrounding index set gives the index values to the message to be instantiated (see Definition 6.3). Following above forall semantics, we instantiate a message with index value `c` by using function $[\![\alpha_{msg}]\!]_m([\![\mathtt{c}]\!]_{ix}, false, \phi', sys)$. Note that non-indexed send semantics sets index values to $\overline{0}$ (see Equation 6.1).

6.4 Semantics of Transitions

Receive expressions control the evaluation of actions. Let $t = (\alpha_{trigger}, \alpha_{guard}, \alpha_{action}, \alpha_{to}) \in \langle trn \rangle$, we define a relation on

$$\Gamma_T = \langle trn \rangle \times \mathcal{S} + \mathcal{S} \tag{6.2}$$

Let $a, a' \in \Gamma_A$, we note $a \rightarrow^* a'$ the transitive closure of relation $\rightarrow$ on Γ_A. We define rules that rely on the previous action semantics defined over Γ_A.

1. Transitions are evaluated unconditionally if there is no trigger provided that guard is true.

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

   where ϕ_ϵ is the empty binding.

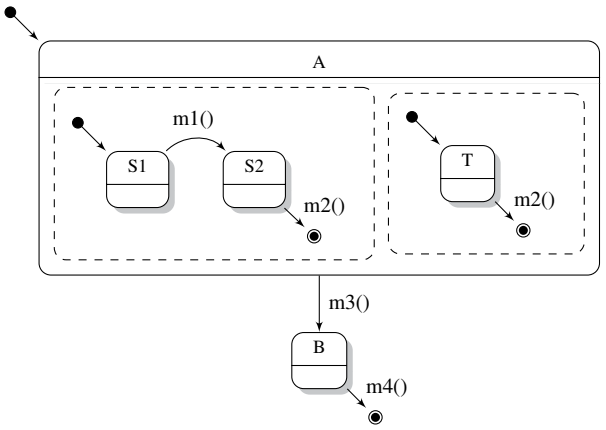
2. If there is a trigger, we check if the first message in the event pool matchs it ($match$) and guard evaluates to true under the local binding context, which is constructed from the input message by $mbind : M \rightarrow \langle trigger \rangle \rightarrow \Phi$.

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

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

   The message binding function $mbind$ takes also into account indexed triggers. Indexed triggers are of the form "`on msg{i}()`" where `i` captures the index value of the incoming message. Therefore, $mbind$ binds incoming index values from m to the corresponding index variables specified in $\alpha_{trigger}$.

Actually, rule TTrigger is richer than the presented one. There are still two important facts to consider, one refers to message consumption, i.e., from which FIFO we are going to pop a message, and the other concerns the initiator setting.



(a) $\langle$HOE$\rangle^2$ graphical state machine

```
sm SM.
  creator c() to A

  cstate A. // Composite state
      region // First region
        initial S1
        state S1. on m1() / to S2
        state S2. endon m2()
      endregion
      region // Second region
        initial T
        state T. endon m2()
      endregion
    on m3() / to B // Outgoing transition
                   // from composite state A
  state B. endon m4()
```

(b) $\langle$HOE$\rangle^2$ code

FIGURE 6.3: Non-Indexed model

External and Internal FIFO. We detailed in Figure 6.1 the functional view of writes to the external and internal FIFOs of the state machine. However, we did not specify how the state machine decides from which one it is going to read. As for the writes, the choice is based on its interface. Naturally, the reads follows similar conditions. More precisely, if $\alpha_{trigger}$ is exposed at the interface then we look for non-reply messages. Otherwise, we look for reply ones.

Initiator. According to Definition 6.2, the object has an initiator reference. We use it in the semantics of sending actions via the `initiator` keyword, as shown in Section 6.3.3. On the other hand, the message sender reference (as defined in 6.3) was never used yet. Let $m \in M$ be a non-reply message, the initiator reference $I_{\hat{o}}$ is set to the message sender reference $\mathcal{R}_m$ iff $\mathcal{M}_m$ belongs to the object interface. That is, the object receives an external message.

6.5 State Machine Evaluation

We start by considering the non-indexed configuration model. A configuration indicates the current *active* states down the hierarchy of the Hierarchical State Machine (HSM). Therefore, it must model simple, composite and final conditions. The hierarchical state machine configuration is a *functor* K defined as follows:

$$K(a) = a + a \times [K(a)] + End \tag{6.3}$$

with injections for simple $\iota_s : a \to K(a)$, composite $\iota_c : a \to [K(a)] \to K(a)$ 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 6.3 is:

$$k = \iota_c(\text{'A'}, [\iota_s(\text{'S1'}), \iota_s(\text{'T'})]) \tag{6.4}$$

Configuration (6.4) indicates that the state machine is at composite state A where its left region is at state S1 and its right one at state T.

We define the transition semantics of configurations as a relation on the lifted set $\langle$*state*$\rangle$ $\langle$*state*$\rangle$

$$\Gamma_K = K(\langle state \rangle) \times \mathcal{S} + \mathcal{S}$$

which we note $\rightsquigarrow$. Using the transition relation on (6.2), we introduce the set of evaluation rules shown in Figure 6.4 where α^i_{trn} is the i-th transition out of $s \in \langle state \rangle$ and the next state configuration function $\kappa n : \langle sm \rangle \to \langle id \rangle \to K(\langle state \rangle)$.

We defined two kind of inference rules and for each one of them we have two possibilities corresponding to the transition type to be evaluated. Rules KsExt and KsFinal handle simple configurations with external and final transitions, respectively. If applicable, KsExt evaluates transition α^i_{trn} and computes next configuration. KsFinal perform the same action on final transitions and evaluates to ι_{end}. Rules KcExt and KcFinal follows the same pattern for composite configurations. The key observation is that they evaluate the transition iff all its composite configurations have terminated (ι_{end}). Finally, KcPar evaluates parallel regions in case KcExt and KcFinal are not applicable.

On the expressiveness of K. The functorial definition allow us to lift any function into the functor structure. For instance, let $\pi_{trn} : \langle state \rangle \to [\langle trn \rangle]$ be the projection that takes a $\langle$HOE$\rangle^2$ state and returns the list of its outgoing transitions. Given that K is a functor, we have for free the lifted version of π_{trn} as $K(\pi_{trn}) : K(\langle state \rangle) \to K([\langle trn \rangle])$. That is, we can apply any function all over the hierarchal structure introduced by the functor.

In order to give a precise relation between $\langle state \rangle$ and $K(\langle state \rangle)$, we define the entry configurator $\kappa : \langle state \rangle \rightarrow K(\langle state \rangle)$ that given a state s returns the entry configuration down the hierarchy of states. Let $L(a)$ be the list functor, then we define κ as follows

$$\kappa(s) = \begin{cases} \iota_s(s) & s = \texttt{state}\ \ \alpha_{id}\texttt{.}\overline{\alpha}_{trn} \\ \iota_c(s, L(\kappa r)(\overline{\alpha}_{region})) & s = \texttt{cstate}\ \ \alpha_{id}\texttt{.}\overline{\alpha}_{region}\overline{\alpha}_{trn} \end{cases}$$

and

$$\begin{aligned} \kappa r(\texttt{region}\ \ \alpha_{initial}\ \overline{\alpha}_{state}) \quad &= \kappa(s_{initial}) \\ &\text{where } s_{initial} = lookInitial(\alpha_{initial}, \overline{\alpha}_{state}) \end{aligned}$$

Therefore, we build the entry configuration with two mutually recursive functions κ and κr. The entry configurator κ builds a simple configuration for simple states and a composite one, using κr to move across regions, in case of composite states. The function $lookInitial$ simply returns the initial state pointed by the `initial` keyword.

For instance, consider state A of Figure 6.3. Entering a composite state like A from any transition implies entering all the initial states of its corresponding regions. This action applies recursively on the initial state of each particular region. Using κ, we can formally build the sample configuration shown at (6.4)

$$\kappa(A) = \iota_c(A, [\iota_s(S1), \iota_s(T)])$$

The expressive power of K also comes into play when we want to take all hierarchical outgoing transitions from $\kappa(A)$:

$$K(\pi_{trn})(\kappa(A)) = \iota_c([t_{A,B}], [\iota_s([t_{S_1,S_2}]), \iota_s([t_T])])$$

Running example. As a running example, consider the state machine of Figure 6.3 at configuration (6.4). Let $e_{\hat{o}} = [m]$ be the message pool of the object under evaluation $\hat{o}$ where message m contains the identifier $M_m = \text{m1}$. 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 which evaluates to sys'.

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

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

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

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

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

FIGURE 6.4: Evaluation rules for configurations

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.

6.6 Indexed Configurations

We extend the hierarchical configuration K defined in (6.3) as follows

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

We model the indexed configuration $IK(a)$ as a disjoint union of the classical configuration, $K(a)$, and the indexed one, $\mathbb{Z}^n \to K(a)$. The indexed configuration is a function from indexes to configurations. The configuration value End allow us to view such function as a partial one.

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 6.3 to be indexed with index

```
sm SM.
  creator c() to A

  cstate A. // Composite state
      region // First non-indexed region
        initial S1
        state S1. on m1() / to S2
        state S2. endon m2()
      endregion
      region { i: 0 <= i <= 255 } // Indexed region
        initial T
        state T. endon m2()
      endregion
    on m3() / to B // Outgoing transtion from composite state A

  state B. endon m4()
```

LISTING 6.1: Indexed regions

```
scalar Int
  interface
    on Add(Int)  -> Added(Int)  ~> AddOp
    on Mult(Int) -> Multed(Int) ~> MultOp
  model Z with (+): AddOp, (*): MultOp
```

LISTING 6.2: Integer Scalar

set `[i: 0 <= i <= 255]` as show in Listing 6.1, we extend sample configuration (6.4)

$$k = \iota_c(\text{'A'}, [(\iota_k \circ \iota_s)(\text{'S'}), \iota_i(fr)]) \tag{6.5}$$

where

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

The indexed configuration (6.5) puts the state machine at state A, where its nested first region is at state S and the second indexed one is at $T_i, \forall i \in [0, 255]$.

6.7 Scalars

The semantics of scalars differs slightly from that of objects in the sense that they "hide" a concrete value belonging to a specified mathematical set, hence providing state machine semantics to scalar values. For instance, consider again the Int object shown at Listing 6.2.

In this particular case, the Int object wraps a $value \in \mathbb{Z}$ where $\mathbb{Z}$ is the denotation of `Z`. The operations `(+)` and `(*)` are known binary operations on $\mathbb{Z}$.

```
state Operating.
  on add(p: Int) / Int result = applyon addOp(p): initiator.added(result)
    to Operating
```

LISTING 6.3: Applyon usage inside state machine of Int

Definition 6.5. The scalar is a parametric structure $P(v) = (\langle sm \rangle, K, E, I, v)$ where v is instantiated to the mathematical model indicated by **`model`**.

Given the underlying model of `Int` and Definition 6.5, we denote scalars relying on such a model with $P(\mathbb{Z})$. In consequence, we must extend the store mapping defined as $A = \bigcup_{R \subseteq \mathcal{R}} R \to O$ to support scalar as well

$$A = \bigcup_{R \subseteq \mathcal{R}} R \to O + P$$

The operational transitions as described in the Int interface combined with the scalar model play a central role in the semantics of **`applyon`**.

6.7.1 Applyon Semantics

Although the **`applyon`** is defined inside the update action, we developed its semantics here as it is strongly related to the semantics of ⟨HOE⟩^{2} scalars. We recall its formal definition hereafter:

⟨*applyon*⟩ ::= '**`applyon`**' ⟨*id*⟩'**`(`**' ⟨*var*⟩* '**`)`**'

A valid instance of ⟨*applyon*⟩ inside `Int` is shown at Listing 6.3. The expression is only valid in the context of an scalar object, i.e., $\hat{o}$ is an scalar, and takes ⟨HOE⟩^{2} scalars as parameters. The operation $addOp$, which is formally related to addition operation on $\mathbb{Z}$, applies on the current object value $v_{\hat{o}}$ and its parameter value v_p where $p = \phi(\mathbf{p})$.[4] We define the applyon semantics such that it returns a new scalar value defined as

$$s_0 = (\alpha_{sm}, k_0, \epsilon, Null, v_{\hat{o}} + v_p)$$

In the example of Listing 6.3, this value is binded through a new reference in the local binding context of $\hat{o}$ and added to the system. It is then sent as a reply message to the corresponding **`initiator`**.

[4]Note that ϕ corresponds to the local binding context and not the object one.

6.8 Contributions

In this chapter, we presented the formalization of the $\langle \text{HOE} \rangle^2$ language in a layered and composable manner. Although not exhaustive, we showed how the approach let us formalize in a modular way almost all the particularities of an action language like $\langle \text{HOE} \rangle^2$ based on communicating HSMs. In particular, we did not specify array value accesses, error conditions and *all* semantics.

We considered parallel and sequential compositions at the action and state machine level. $\langle \text{HOE} \rangle^2$ supports object creation, detailed by our semantics at the action level. At this level, we also specified initiator semantics, which creates reply messages. The initiator is modified at the transition level. Such interaction between two stages demonstrates how the hierarchical approach enable the complete specification of an unique concept in a structured way.

The functor abstraction for the state machine configuration leads to a clear operational view of state machine steps. Moreover, the indexed region semantics turns into a natural and clear extension to the functor structure.

Chapter 7

Intermediate Representation

Contents

We present an Intermediate Representation (IR) suitable for the efficient compilation and optimization of network of communicating Hierarchical State Machines (HSMs).

In contrast to the Highly Heterogeneous, Object-Oriented, Efficient Engineering ⟨HOE⟩^{2} language, which is object-oriented, the IR disassociates type definitions from state machines. The IR structure is inspired from the low-level implementation of the object-oriented paradigm in languages such as C++. Following a similar path and considering the semantics of ⟨HOE⟩^{2} presented in chapter 6, we detach the HSM from the object definition and create independent

```
object Pixel {
  interface {
    takeY(Float y);
    getY() -> takeY(Float y);
  }
  associations {
    Int r, g, b;
    Float y;
  }
}
```

LISTING 7.1: IR type definition (Pixel Part I)

function-like state machines. The approach allow us to make a clear distinction between structural definitions and execution tasks that implement the state machine. The tasks use the structure instances to communicate with other tasks. It is high-level enough and implementation independent, similar to the $\langle$HOE$\rangle^2$ model itself, because it preserves the type definitions and the state machine hierarchy.

The IR exposes implicit information in $\langle$HOE$\rangle^2$ such as initiator definitions, message types and variables, the "this" variable pointing to the owner object of the current state machine, etc.. For instance, an $\langle$HOE$\rangle^2$ object cannot send messages to itself in an explicit manner. There is no such support as it may lead to undesirable conditions when considering actual implementations with bounded message channels. The IR, however, poses no restriction concerning self-sending actions as they are forbidden by the input language.

7.1 Overview

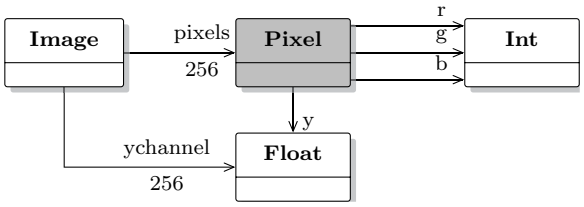
In order to present the general structure of the IR, we will introduce a simple – but still rich enough – translation example of the model shown in Figure 7.1 for the Pixel object.

The translation of Pixel type definition is shown in Listing 7.1. It is essentially the same with a slight change to the syntax of multi-valued association types.

While type constructors remain close to the $\langle$HOE$\rangle^2$ ones, the state machine is divided into two top level block expressions, **`fsm`** and **`creator`**, where both of them are composed by imperative statements. The statements preserve the structure of the HSM and $\langle$HOE$\rangle^2$ action language.

Creator

Consider the translation of Pixel creator RGB in Listing 7.2. The creator signature is preserved and implicit actions such as structural allocations (**`alloc`**) and state machine running are exposed



(a) Image model

```
sm PixelSM.
  creator RGB(rgb: Int[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.add(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
```

(b) Pixel state machine

FIGURE 7.1: Pixel model

```
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;
}
```

LISTING 7.2: IR creator (Pixel Part II)

```
fsm Pixel.PixelSM(Pixel this) {
INIT:
  wait this for (_m1_getY, Object src) = recv getY<>
    then when _m1_getY goto SBB_GET_Y;

SBB_GET_Y:
```

LISTING 7.3: IR state machine (Pixel Part III)

(**start**). Indeed, the creator job is to allocate, initialize the object and run its state machine. Note that we keep updates in parallel at the IR level as well.

```
SBB_GET_Y:
  Float rcst = create Float.float 0.299,
  Float gcst = create Float.float 0.587,
  Float bcst = create Float.float 0.114;

  sendfrom this this.r mult<Float> rcst,
  sendfrom this this.g mult<Float> gcst,
  sendfrom this this.b mult<Float> bcst;

  wait this for _m1_multed = recv Float'multed<Float>,
                _m2_multed = recv Float'multed<Float>
    then when _m1_multed, _m2_multed goto UBB_MULTING;

UBB_MULTING:
```

LISTING 7.4: Parallel expressions (Pixel Part IV)

Receiving

Listing 7.3 shows the entry point of the state machine. First of all, the state machine is introduced with **fsm** and its unique and mandatory argument plays the role of the "this" pointer found in other object-oriented programming languages (here it might have been called differently). The state machine contains a list of labeled statements. The first statement is a **wait** expression. It needs to know on which *channel* we want to wait together with a list of receives clauses, which in this case refers to the external message `getY`. In the IR context, we consider object as channels and use both terms indifferently. The receive clause allows us to extract all the information contained into the incoming message. It defines two variables `_m1_getY` and `src` which are of type `Message` and `Object`, respectively. The message is a struct-like variable enclosing all the formal parameters while `src` refers to the message sender. It follows a list of **then** test cases. In above state machine, we branch to label `SBB_GET_Y` if message `_m1_getY` has been received. We will develop more about each statement in the following sections.

Parallel update and send

To continue with the overview of this particular example, Listing 7.4 shows the start of the luminance computation. We found parallel definitions of variables (`rcst`, `gcst`, `bcst`) and parallel sending actions (**sendfrom**). In order to have a close equivalence to the $\langle \text{HOE} \rangle^2$ input language, the IR implements parallel statements that combine send and update actions. Note that dot notation no longer means message sending but structural access to object and message variables.

The statement **create** calls the creator of Float object with the corresponding value. Similar to the $\langle \text{HOE} \rangle^2$ implementation, it sends three multiplication messages in parallel and waits for two of them afterwards.

```
UBB_ADDING:
  this.y = _m5_added.res;
  reply this src takeY<Float> this.y;
  done this;
```

LISTING 7.5: Reply (Pixel Part V)

Replying

Once the computation is done, Pixel state machine replies to the sender providing it the result inside message `takeY` as shown in Listing 7.5.

It stores the final computation result coming inside message variable `_m5_added` (issued by a Float object) and, using **`reply`**, build message `takeY` and specifies sender (`this`), destination (`src`), message type and parameters.

In conclusion, the intermediate representation separates type definitions from state machines based on the implementation idea of object-oriented paradigm. It keeps interface and association definitions, exposes object and message variables where the former are seen as channels and the latter are struct-like variables grouping message formal parameters. We found specific instructions implementing communication primitives and, most importantly, the parallelism is preserved. In the following sections, we introduce the IR specifics with meaningful examples.

7.2 Structure

Formally, the IR has the following top-level statements

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

Import, object and scalar expressions follow the same constructions as their ⟨HOE⟩[2] counterparts. They introduce new type definitions, objects and scalars, implementing an interface and a set of associations and a denotational model, respectively.

⟨*object*⟩ ::= '**`object`**' ⟨*ID*⟩ '{' ⟨*interface*⟩ ⟨*associations*⟩ '}'

⟨*scalar*⟩ ::= '**`scalar`**' ⟨*ID*⟩ '{' ⟨*interface*⟩ ⟨*model*⟩ '}'

Because the IR handles explicit message variables, it also introduces *message types*.

Message types

Message types have a particular syntax. They are composed by an identifier followed by a list of parameter types. Two messages with same identifier but different parameter types are considered to be distinct. For instance, `mult<Float>` and `mult<Int>` are two different message types. The set of message types is defined as

⟨*msg_type*⟩ ::= ⟨*simple_msg_type*⟩
| ⟨*qualified_msg_type*⟩

⟨*simple_msg_type*⟩ ::= ⟨*ID*⟩ '<' ⟨*obj_type_list*⟩ '>'

⟨*qualified_msg_type*⟩ ::= ID '**'**' ⟨*simple_msg_type*⟩

Given that two objects may define same message types, qualified message types allow us to disambiguate such naming collision when necessary. For instance, `Float'mult<Float>` and `Int'mult<Float>` define the same message, which models multiplications, under different sets of values.

As discussed in previous section, we separated object type definitions from the state machine. The latter being further decoupled into two different blocks: ⟨*creator*⟩ and ⟨*fsm*⟩.

7.3 Creator

The creator, introduced by the keyword **`creator`**, must have an associated type and identifier and it may define a variable number of parameters.

⟨*creator*⟩ ::= '**`creator`**' ⟨*ID*⟩ '**`.`**' ⟨*ID*⟩ '**`(`**' ⟨*param_defs*⟩ '**`)`**' '{' ⟨*creator_stmt*⟩ '}'

The allowed statements inside creators are defined as follows

⟨*creator_stmt*⟩ ::= ⟨*creator_stmt*⟩ '**`;`**' ⟨*creator_stmt*⟩
| ⟨*parstmt*⟩
| ⟨*forall_block*⟩
| ⟨*start*⟩

⟨*parstmt*⟩ ::= ⟨*parstmt*⟩ '**`,`**' ⟨*parstmt*⟩
| ⟨*update_expr*⟩

⟨*forall_block*⟩ ::= '**`forall`**' '**`[`**' ⟨*indexset*⟩ '**`]`**' '{' ⟨*creator_stmt*⟩ '}'

⟨*start*⟩ ::= '**`start`**' ⟨*ID*⟩ '**`of`**' ⟨*ID*⟩

We may have parallel statements, a start directive and forall block expressions. No communication primitive is allowed inside creators. Its main purpose is preserved: allocation, initialization and object running. As described earlier, before initializing the object we need to allocate the structure using directive **`alloc`**. Once the object is allocated and initialized, **`start`** runs the indicated state machine. It takes two arguments, the state machine name to run and its associated structure.

The forall block defines an index domain. Similar to index domains in the ⟨HOE⟩^{2} language, it is composed by a list of index variables and a constraint formula. It is a handy statement when initializating multi-valued associations. The Listing 7.2 shows a simple creator instance.

In the following, we explain extensively update and send statements within the context of state machines.

7.4 Hierarchical State Machine

The state machine is introduced using the token **`fsm`**.

⟨*fsm*⟩ ::= '**`fsm`**' ⟨*ID*⟩ '**`.`**' ⟨*ID*⟩ '**`(`**' ⟨*vardecl*⟩ '**`)`**' '{' ⟨*fsmstmt*⟩ '}'

where its statements are

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

We have a list of, optionally labeled, imperative statements: parallel statements, variable declarations, wait and wait-in statements, unconditional branching ⟨*goto*⟩ and an object termination instruction ⟨*done*⟩.

7.4.1 Parallel Statements

In contrast to ⟨HOE⟩^{2}, the IR mixes in a single parallel expression send and updates. Therefore, it is defined as follows

```
sendfrom this this.r mult<Float> rcst,
sendfrom this this.g mult<Float> gcst,
sendfrom this this.b mult<Float> bcst;
```

LISTING 7.6: Parallel statement

```
forall[i: 0 <= i and i < 512]
  sendfrom[i] this this.pixels[i] getY<>;
```

LISTING 7.7: Indexed send

⟨*parstmt*⟩ ::= ⟨*parstmt*⟩ ’**,**’ ⟨*parstmt*⟩
| ⟨*send_expr*⟩
| ⟨*update_expr*⟩

⟨*send_expr*⟩ ::= ⟨*send*⟩
| ’**forall**’ ’**[**’ ⟨*indexset*⟩ ’**]**’ ⟨*send*⟩

⟨*update_expr*⟩ ::= ⟨*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 sets between brackets.

7.4.2 Send

There are three kind of sending commands: *single send*, *indexed send* and *reply*. They are defined as follows:

⟨*send*⟩ ::= ’**sendfrom**’ ⟨*varexpr*⟩ ⟨*varexpr*⟩ ⟨*msg_type*⟩ ⟨*param*⟩*
| ’**sendfrom**’ ’**[**’ ⟨*arith_exprs*⟩ ’**]**’⟨*varexpr*⟩ ⟨*varexpr*⟩ ⟨*msg_type*⟩ ⟨*param*⟩*
| ’**reply**’ ⟨*varexpr*⟩ ⟨*varexpr*⟩ ⟨*msg_type*⟩ ⟨*param*⟩*

In order to send a message, **sendfrom** needs source and target objects, a message type and its required parameters. The indexed send is similar to the first one plus a list of arithmetic expressions between brackets. This notation allows the initialization of indexes when building indexed messages.

At the overview, we presented three parallel sending that we recall in Listing 7.6. In order to broadcast messages over multi-value associations, we make use of forall expressions. Listing below shows a simple example, which is taken from the translation of Image to Pixel broadcasting shown at Listing 7.7.

We found an indexed send that builds a one-dimensional message with sender `this`, destination `this.pixels[i]`, message type `get<>` and index value `i`.

We send replies with command **`reply`** that serves to the implementation of $\langle \text{HOE} \rangle^2$ initiator semantics. In the IR context, reply must have as a target object a variable defined at some receive expression. For instance, consider the following reply already introduced in Listing 7.5

```
reply this src takeY<Float> this.y;
```

Note that `src` was defined at the top receive expression

```
wait this for (_m1_getY, Object src) = recv getY<>
  then when _m1_getY goto SBB_GET_Y;
```

7.4.3 Update

The update statements are defined as

⟨*update*⟩ ::= ⟨*var*⟩ '**=**' ⟨*varexpr*⟩
| ⟨*var*⟩ '**=**' '{' ⟨*param*⟩* '}'
| ⟨*var*⟩ '**=**' '**create**' ⟨*ID*⟩ '.' ⟨*ID*⟩ ⟨*param*⟩*
| ⟨*var*⟩ '**=**' '**applyon**' ⟨*varexpr*⟩ ⟨*applyon_type*⟩ ⟨*param*⟩*

The first one is a classical update operation between two variables. It is followed by an association initializer, an object creation directive and a new expression **`applyon`** that denotes "in-place" operations. Therefore, we can create pixel objects from initialized variables in the following way

```
Int|3| inraw_slice = { rgb[3*i] rgb[3*i+1] rgb[3*i+2] };
this.pixels[i]     = create Pixel.RGB inraw_slice;
```

LISTING 7.8: Variable initialization and object creation

As shown in chapter 5, the scalar object abstracts away scalar operations via operational transitions. They are materialized as "in-place" operations at the IR level using command **`applyon`**. It takes an object variable to apply a type of operation, which designed by the name of the operational transition, and its required parameters.

```
Int res_added = applyon i Int'add msg_add.a;
```

LISTING 7.9: Apply an operational transition

For instance, Listing 7.9 apply the operational transition `Int'add` on integer `i` with parameter coming from a certain received message called `msg_add`. We will discuss more about this instruction in chapter 8. Intuitively, given the type of operational transition, **`applyon`** is mapped directly to the corresponding operator that matchs the target language semantics at code generation time.

7.4.4 Branching

The IR supports conditional and unconditional branching.

⟨*goto*⟩ ::= '**goto**' ⟨*ID*⟩

⟨*wait*⟩ ::= '**wait**' ⟨*varexpr*⟩ [⟨*waitfor*⟩] ⟨*waitthen*⟩+

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

The expression **`goto`** unconditionally jumps to a labeled location. Instructions **`wait`** and **`waitin`** represent conditional branches with multiple target locations, differing from traditional intermediate representations for imperative languages where conditional branching schemes have two possible destinations. The **`wait`** clause requires an object to listen from, a list of accepted messages to wait for (⟨*waitfor*⟩) and a list of ⟨*waitthen*⟩ branches to jump to under specific conditions. This statement groups all the information concerning outgoing transitions of states. For instance, consider again the wait statements of Listing 7.3 and 7.4, which we present again below.

```
wait this for (_m1_getY, Object src) = recv getY<>
  then when _m1_getY goto SBB_GET_Y;

wait this for _m1_multed = recv Float'multed<Float>,
              _m2_multed = recv Float'multed<Float>
  then when _m1_multed, _m2_multed goto UBB_MULTING;
```

Both waits define message variables and have one **`then`** branch without guard. They wait on `this` until the indicated messages specified at when expressions are present. Additionally, the first one defines also a source variable object that represent the sender. After the object from which the wait statement is going to listen, we defined a comma-separated list of receive expressions as follows

⟨*waitfor*⟩ ::= '**for**' ⟨*recvexpr*⟩+

⟨*recvexpr*⟩ ::= ⟨*recvdef*⟩ '**=**' '**recv**' ['**[**' ⟨*indexset*⟩ '**]**'] ⟨*msg_type*⟩

⟨*recvdef*⟩ ::= ⟨*var*⟩ | '**(**' ⟨*var*⟩ '**,**' ⟨*var*⟩ '**)**'

Essentially, the receive expression provides a way to access grouped parameters, sender and indexes separately from the incoming message. Indexes are captured by *indexed receives*. We defined an optional ⟨*indexset*⟩ rule between brackets at receive expressions. This notation allows us to capture and bind indexes from the incoming message. For instance, consider the indexed reception shown at Listing 7.10

```
wait this for
  msg_takeY = recv[i: 0 <= i and i < 512] takeY<Float>
  then when msg_takeY goto UPDATE_CH
  then if msg_takeY.all goto FINAL;
```

LISTING 7.10: Indexed receive

It waits for the reception of a single message whos index value satisfies the specified constraint. The index variable `i` is available at all reachable statements. Message variables can only be declared and defined at receive expressions.

After introducing the list of receives, ⟨*waitthen*⟩ checks for message presence, evaluates a possible triggering condition and branches to the specified label.

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

Finally, **waitin** has the same structure as **wait** with additional support for parallel regions. We show in Section 7.5.5 an instance of this statement.

7.4.5 Regions

As mentioned earlier, the IR preseves the hiearchical structure of the original state machine. The statement **waitin** is composed by a list of regions, which can be either *simple* or *indexed*. They are defined as

⟨*region*⟩ ::= ⟨*sregion*⟩ | ⟨*iregion*⟩

⟨*sregion*⟩ ::= '{' ⟨*fsmstmt*⟩ '}'

⟨*iregion*⟩ ::= '**[**' ⟨*indexset*⟩ '**]**' '{' ⟨*fsmstmt*⟩ '}'

This definition closely follows the concept of indexed regions proposed by the ⟨HOE⟩^{2} language.

7.5 Translating $\langle$HOE$\rangle^2$

As we have shown in previous sections, the IR preserves most of the information coming from $\langle$HOE$\rangle^2$ while making them more explicit. We even found some equivalent constructions such as objects with only syntactic modifications. Indeed, objects together with its structural features are translated almost unchanged into object and/or scalar instances. In addition, container type constructors are also preserved. The fundamental change is found at the state machine level. We separate it into creator and fsm "functions". Most of these translations are relatively straightforward to perform as we have thoroughly shown with plenty of examples. However, some key characteristics of $\langle$HOE$\rangle^2$ need more precision on how the translation is to be done. Therefore, we consider several points needing further explanation

1. Compilation of states
2. Propagation of index domains
3. Defining initiators
4. "All" condition
5. Indexed regions

For the sake of clarity, we recall some notations already introduced in Chapter 6. We will refer to an instance of a given syntax set S corresponding to the $\langle$HOE$\rangle^2$ grammar as α_S. For instance, α_{state} is an instance of $\langle$HOE$\rangle^2$ state and its projection on transitions is noted $\pi_{trn}(\alpha_{state})$. We use classical notation for lists $\overline{v} = [v_i]$. Given a disjoint union set $S = A+B$, we use subscripted injections $\iota_A : A \to S$ and $\iota_B : B \to S$ to represent elements of S.

7.5.1 Compilation of States and its Transitions

We presented two statements that allow to check for incoming messages: **`wait`** and **`waitin`**. Both of them specify a group of messages to wait for and their corresponding **`then`** branches. $\langle$HOE$\rangle^2$ state translates into a wait statement and a set of sequential statements (basic blocks) concerning the different transition actions.

7.5.1.1 Actions

Let α_{state} be a state and $t \in \pi_{trn}(\alpha_{state})$, we build a set of basic blocks $BB_{state} = \{bb_t\}$ for each transition action $a \in \pi_{action}(t)$. We define the label of bb_t as $\#bb_t$, which given that $bb_t \in \langle$*fsmstmt*$\rangle$ it corresponds to the label of its first statement. BB_{state} will be the set of all

possible destination of **`then`** branches. All basic blocks resulting from external transition actions terminate with unconditional branches to target state identifiers. In case of final transitions, the basic block terminate with statement **`done`** `this`, which effectively finishes the current state machine.

7.5.1.2 Transition

We separate the translation of transitions into two steps: collection of triggers and computation of **`then`** branches.

Collecting triggers. We create a multiset from transition triggers $MSet(\pi_{trg}(t)) = (Trg, m)$, also noted $MSet_t$, where Trg is the set of triggers and $m : Trg \to \mathbb{N}$ the multiplicity of each trigger. We will refer to this multiset tuple as Trg_t and m_t. Then, we define a special multiset union where we take for each element the maximum number of elements between both sets. Its defintion is shown hereafter

$$\begin{aligned} MSet_t \uplus MSet_{t'} &= (Trg, m) \\ Trg &= Trg_t \cup Trg_{t'} \\ m(trg) &= max(m_t(trg), m_{t'}(trg)) \end{aligned}$$

Above definition allow us to create a common set of receive expressions. Indeed, we define the final multiset WR of wait triggers as $WR = \biguplus_{t \in \pi_{trn}(\alpha_{state})} MSet_t$. To each $wr \in WR$, we associate a message variable rv^i_{wr}, which is the i-th trigger kind of wait wr. From this set of variables, we build the receive expression list

$$\overline{recv} = [rv^i_{wr} \ \texttt{= recv} \ wr] \tag{7.1}$$

In order to build the message variable list for each transition t, we take $MSet_t$ and we proceed as before, i.e., building a list of message variables wv^i_{wr} for each trigger kind $wv \in MSet_t$, such that $wv^i_{wr} = rv^i_{wr}$. It yields same variable names at **`recv`** and **`when`** expressions. The when expressions for transition t is

$$when_t = \texttt{when} \ \ wv^i_{wr} \tag{7.2}$$

Compute "then" branches. Assuming a straightforward translation g_t of guard condition for transition t, we build the list of **`then`** expressions as

$$\overline{then} = [then_t] \tag{7.3}$$

$$then_t = \textbf{then}\ when_t\ \textbf{if}\ g_t\ \textbf{goto}\ \#bb_t \tag{7.4}$$

Collecting definitions (7.1),(7.2) and (7.3), the wait statement is defined as

$$waitstmt = \pi_{id}(\alpha_{state})\textbf{:}\quad \textbf{wait this for}\ \overline{recv}\ \overline{then} \tag{7.5}$$

We added a label corresponding to the state identifier such that the compilation of incoming transitions have consistent branch destinations. We did not take into account indexed receives but the translation procedure remains the same. Receive expressions coming from initiator conditions will be discussed later.

7.5.2 Propagation of Index Domains

There is a subtle difference between indexed receives at $\langle \text{HOE} \rangle^2$ and IR. In the former, index domains at send and receive are syntactically linked whereas in the latter we have decoupled them. For instance, consider the following $\langle \text{HOE} \rangle^2$ indexed send and receive pairs

```
1    state GETY.
2      on /: { i: 0..pixels.len - 1 } pixels[i].getY()
3      to GETTING_Y
4
5    state GETTING_Y.
6      on takeY{i}(y: Float) /
7           ychannel[i] = y to GETTING_Y
8      endon [i.all]
```

At the indexed receive of line 6, we know exactly the domain of `i`, which has been declared at line 2. In contrast to the IR, we introduce a new index variable with its associated domain each time we declare an indexed receive.

```
wait this for
  msg_takeY = recv[i: 0 <= i and i < 512] takeY<Float>
```

LISTING 7.11: Rebinding index domain

```
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) $\langle \text{HOE} \rangle^2$ Initiator

```
  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) Initiator-like IR code

FIGURE 7.2: Translation of initiators

Therefore, the compilation procedure will propagate send domains to the generated indexed receives as shown by Listing 7.11.

7.5.3 Defining Initiators

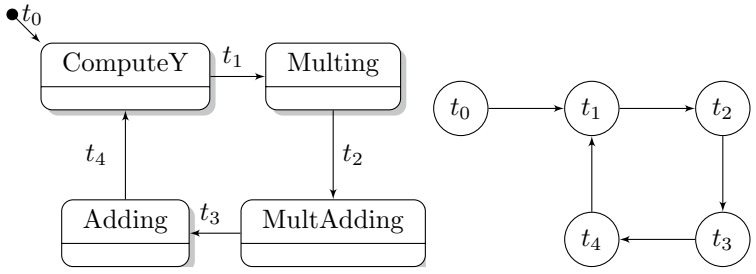
According to the $\langle \text{HOE} \rangle^2$ semantics, initiators are implicitly defined following the object interface. In the IR, we translate initiator semantics with concrete variable definitions as we informally presented in the overview 7.1. Figure 7.2 summarises such translation. It describes a receive expression that captures the sender `src` to perform a reply on it. The reply semantics has two important implications

a. Index-forwarding: By tracking which receive has defined the sender the IR compiler automatically transfers index values from the incoming message into the new reply message.

b. Given that sender variables can be defined using the same name at different receive expressions, we may have control-flow dependent senders.

A correct translation of initiator semantics asks for a data-flow analysis. The problem can be stated as follows

Reaching Definitions of Initiator. Given a HSM over the actions defined by the $\langle \text{HOE} \rangle^2$ language, a set of transitions defining the initiator and replying transitions (**`initiator`**), we need to "structurally" find the correct initiator def-use chain. By structurally, we mean a solution that takes into account nested state machines.



(a) Pixel state machine (b) Complement state machine

FIGURE 7.3: $\overline{SM}$ model of Pixel state machine

We have a special case of reaching definitions where we deal with only one variable [68]. In order to develop the underlying translation process, we introduce a simple state machine model $SM = (S, T)$, where S is the set of states with transitions $T \subseteq S \times S$. We define its complement $\overline{SM} = (T, U)$ where transitions become vertices of the graph and $(t, t') \in U$ iff $\forall s_1, s, s_2 \in S \mid t = (s_1, s) \wedge t' = (s, s_2)$. The predecessors of t is $preds(t)$. The complement state machine allow us to think of SM as a control-flow graph of statements. We introduce the reaching definition set on $\overline{SM}$ at the action of transition t as $rd_t \subseteq T$ and we formulate it in the following way.

$$rd_t = \begin{cases} \{t\} & \text{if } t \text{ reacts to an external message} \\ \bigcup\limits_{p \in preds(t)} rd_p & \text{otherwise} \end{cases} \tag{7.6}$$

That is, if transition t contains a trigger defined as an external input message by the interface then t will define the initiator, effectively "killing" any other incoming definition. Otherwise, we propagate the information. Clearly, the computation of rd_t converges since it is a monotone function over the complete-lattice given by the powerset of T.

For instance, the Pixel state machine at Listing 7.1 and its complemented version are described at Figure 7.3. Given that t_1 is the only transition that triggers on an external message, namely `getY()`, we obtain the following trivial solution

$$rd_{t_0} = \emptyset \quad rd_{t_1} = \{t_1\} \quad rd_{t_2} = \{t_1\} \quad rd_{t_3} = \{t_1\} \quad rd_{t_4} = \{t_1\}$$

Transition t_4 is the only transition to use on **`initiator`** and, by reachability condition, we should assign to the receive expression of `getY()` the same variable name as the reply statement generated from transition t_4. This constraint is satisfied with sender variable `src`, as shown by Figure 7.2.

In order to take into account the state machine hierarchy, we extend SM with hierarchical states, i.e., $S = SS + CS$ where $CS = [SM]$ (a list of nested state machines). A key observation is

that sender variables defined at nested states are *not* visible outside due to scoping. However, the converse is not true. We should be able to send replies on an initiator possibly defined at higher nesting levels. Roughly speaking, it means that our reaching definition can be applied per nesting level taking into consideration initializing conditions coming from upper ones.

Therefore, we redefine our complement state machine with hierarchical nodes $HT = ST + CT$ where $CT = [\overline{SM}]$, i.e., we preserve compositional information. We compute the complement as follows $\overline{SM} = (HT, U)$

- For each $t = (s, s') \in T$, we have $ht \in HT$ defined as

$$ht = \begin{cases} \iota_{CT}([\overline{sm}]) & \text{if } s' = \iota_{CS}([sm]) \\ \iota_{ST} & \text{if } s' = \iota_{SS} \end{cases}$$

  Note that $\iota_{CT}([\overline{sm}])$ means that we apply the computation recursively.

- We build the relation U as before: $(ht, ht') \in U$ iff $\forall s_1, s, s_2 \in S \mid t = (s_1, s) \wedge t' = (s, s_2)$

Let $\overline{sm}_i$ be some complemented state machine with initial transition ht_i and let $ht = \iota_{CT}([\overline{sm}_i])$. That is, we have a hierarchical node from another complement state machine $\overline{sm}$ on which $\overline{sm}_i$ is nested. Then, we add to the reaching definition formulation the initializing condition $rd_{ht_i} \triangleq rd_{ht}$ before solving into nested levels.

As a result, the proposed defuse chain solution allow the translation procedure to name sender variables correctly over hierarchical state machines.

7.5.4 "all" Condition

The "all" keyword refers to the $\langle \text{HOE} \rangle^2$ condition that is true when all messages have been received (see section 5.4). The IR preserves such information by attaching it to message variables instead of indexes. For instance, Figure 7.4 shows the translation of an indexed receive together with its **`all`** condition. According to the compilation of states presented earlier, the indexed receive at line 7 contains two possible branches, one to the basic block of the translated transition action and a second one to the final basic block. The compilation process attachs the all condition of second branch to the message variable associated to message `takeY<Float>`.

7.5.5 Indexed Regions

As pointed out early in this section, the hierarchy of HSM is provided by **`waitin`** statements. The equivalent IR expression of an indexed region is shown in Figure 7.5.

```
state GETTING_Y.
  on takeY{i}(y: Float) /
       ychannel[i] = y to GETTING_Y
  endon [i.all]
```

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

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

(b) Message variables with **all** condition

FIGURE 7.4: Translating **all** condition

```
cstate GETTING_Y. region { i: 0..255 }
                    [...]
                  endregion
                  on takeY(y: Float) to FINAL
```

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

```
GETTING_Y:
  wait this in
    [i: 0 <= i and i < 255] {
      [...]
    }
    for m_getY = recv takeY<Float>
    then when m_getY goto FINAL;
FINAL:
```

(b) IR Indexed Region

FIGURE 7.5: Translation of Regions

Throughout this section we informally showed the translation of $\langle \text{HOE} \rangle^2$ to our IR. In the next chapter, we will present the optimizing compiler chain and code generation strategy.

7.6 Contributions

The IR is designed to be compiler-friendly, i.e., suitable for analysis and optimizations. Indeed, important compiler structures such as the control-flow of basic blocks can be constructed directly from it. It also offers a cleaner path to code generation. No matter which runtime the language is based on, we need to make a distinction between object creation (introduction into the runtime environment) and its main behavior (thread).

It provides specific instructions for communication primitives with an explicit chain of variable definitions and uses to build incoming messages and extract all the needed information. Most importantly, the IR preserves the state machine hierarchical information. It is also highly expressive as it supports arrays, dynamic creation of state machines, iterators, provides fork/join

semantics in the same way $\langle \text{HOE} \rangle^2$ does, flexible communication primitives and indexed regions, among other features.

In order to give a precise idea of the $\langle \text{HOE} \rangle^2$ view at the IR level, we showed some equivalencies through program examples and completed it with specific translation procedures for more evolved concepts coming from the $\langle \text{HOE} \rangle^2$ language. We will show in the following chapter how to take advantage of the IR constructions to build an state machine optimizer.

Chapter 8

The Compiler: Analysis, Optimization and Code Generation

Contents

Based on the Intermediate Representation (IR) of Chapter 7, we developed the optimizing compiler. The compiler takes advantage of the proposed Highly Heterogeneous, Object-Oriented, Efficient Engineering $\langle \text{HOE} \rangle^2$ features, preserved at IR, to optimize the model and produce highly efficient code.

8.1 Challenges

In the context of optimizations for communicating automata, we face a certain number of difficulties. One challenge concerns *data dependencies*. Efficient code generation of data-intensive applications calls for an accurate knowledge of data dependencies. Unfortunately, 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 [77]. In [77], Peng and Puroshothaman formulate the problem of communicating automata as a set of recurrence equations over the domain of infinite streams of messages. The show 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 saying 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.

8.1.1 Instantaneous Reaction

We introduced in Chapter 5 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

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

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).

8.1.2 Mutability

Another issue is *mutability*, directly related to escaping conditions. When looking for optimization opportunities in presence of concurrent objects, say A and B, we would like to know at some instant the possible state(s) of B for a given state of A. Because data is wrapped inside messages, data-flow translates into concurrency. Many questions arise when compiling concurrent objects

a. Is A the only producer from B's perspective? In other words, is B a shared object?

b. Is A waiting for a response from B which is a direct consequence of a certain message m?

c. On which state is B after consuming message m from A?

We may reformulate (a) and ask whether object B *escapes* from the context of A. If it escapes, then B is a shared object otherwise it is not. Therefore, we must define when an object escapes from the context of another, which leads to the classical escape analysis heavily used in languages like Java to guide stack allocations, inlining and scalar replacements [78, 79]. Let us rather define when an object does not escape. In the context of $\langle \text{HOE} \rangle^2$, we have a set of particular conditions

1. The inlinee is an scalar object: according to properties introduced in Chapter 5, any action on scalar objects (communication for instance) does not have internal side-effects.

2. The inliner creates the inlinee and never shares it. Note that there are two ways to share an object in $\langle \text{HOE} \rangle^2$, either send it via message passing or create another object that accepts it as its creator parameter.

8.2 The Optimizing Compiler

The optimizing compiler chain is shown in Figure 8.1. The basic idea of the compilation flow is to transform the IR code to reach patterns for which we know that an efficient translation to the target language exists. Starting from the IR, the compiler builds a Hierarchical Control-Flow

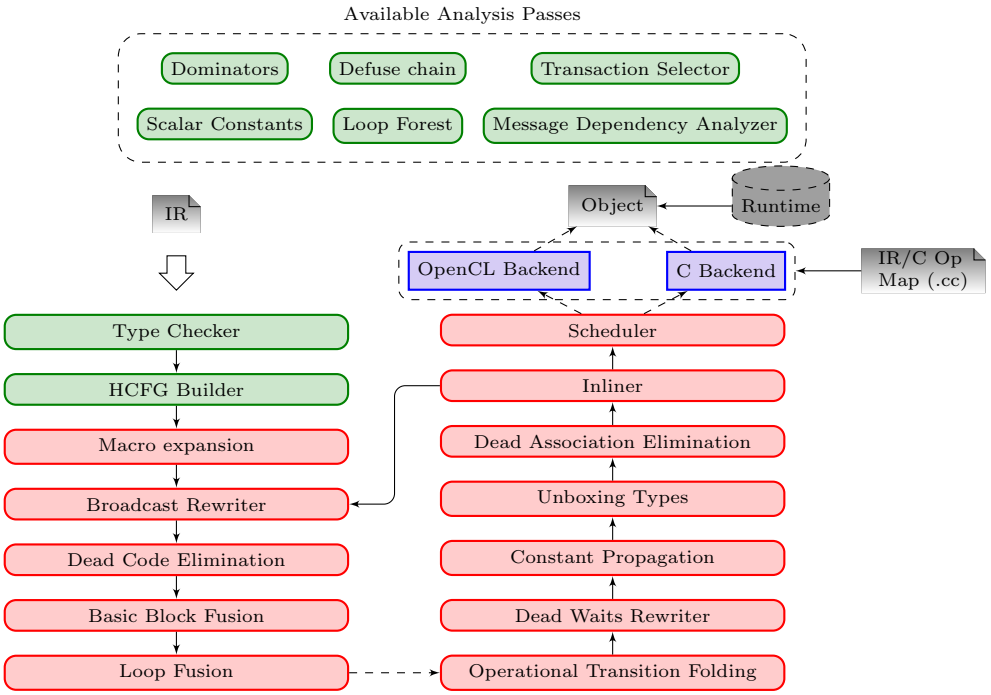


FIGURE 8.1: Compilation Flow

Graph (HCFG) out of the state machine following the branch kinds introduced in Chapter 7. 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 at Section 7.4.4). We apply all transformations recursively on the control-flow structure, i.e., across regions.

Once the model is fully optimized, the C/OpenCL backend generates code with specific runtime calls for object creation, termination, state machine running, send and receive. In addition to these runtime routines, it also needs to know how to translate native operations modeled as operational transitions. For this reason, a mapping between IR operations and target operations is provided separately (source.cc) as it concerns only code generation.

Before getting into the explanation of transformation passes, we introduce some terminology and notations used through this chapter.

Notations. We define a Hierarchical Control-Flow Graph as a graph $HCFG = (BB, E)$ where $BB \in \langle \textit{fsmstmt} \rangle$ and $E \subseteq BB \times BB$. We call it hierarchical because of composite statements such as **`waitin`** containing nested control-flow graphs. Branch instructions **`wait`**, **`waitin`** and **`goto`**, as well as **`done`** terminate basic blocks and thus we called them *terminators*.

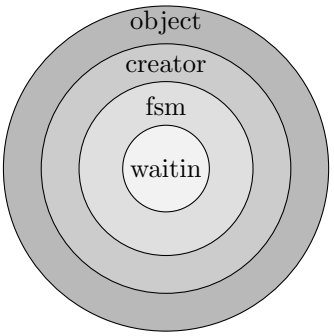


FIGURE 8.2: Definitions layers

8.3 Analyses

In this section, we present our specific analyses used by most of the transformations passes in the compilation chain.

8.3.1 DefUse Chain: Structured Analysis of Reachable Definitions

The variables in the IR can be local to the object, i.e., defined as object associations, or private, declared inside creators or state machines. By design choice, the declaration of arbritrary global variables is not allowed. Instead, they result from escaping conditions: shared local variables are considered to be global.

The object defines the local context of variables, or associations. Local variables allow the object to pass definitions from creator to its related state machine, which provides an initial set of definitions at the **`fsm`** entry. In the same way, the **`fsm`** provides an initialized set of definitions to its lower layers, or regions (**`waitin`**). Figure 8.2 describes pictorically the visibility layers of definitions with classical scoping constraints. Only local variables are visible at all levels of the hierarchy.

We obtain the chain of variable definitions and uses, namely the DefUse Chain, by solving the problem of reaching definitions in the form of data-flow equations. In order to show the structured approach of our analysis in the IR, we define a forward data-flow problem on the HCFG with a higher order transfer function ϕ:

$$ins(bb_i) = \bigcup_{p \in preds(bb_i)} outs(p) \tag{8.1}$$

$$outs(bb_i) = \phi(bb_i)(ins(bb_i)) \tag{8.2}$$

and we define ϕ inductively on basic blocks ($\alpha_{fsmstmt}$) as follows

$$\phi(\alpha_{fsmstmt} \ ; \ \alpha'_{fsmstmt})(d) = \phi(\alpha'_{fsmstmt}) \circ \phi(\alpha_{fsmstmt}) \tag{8.3}$$

$$\phi(\alpha_{fsmstmt} \ , \ \alpha'_{fsmstmt})(d) = \phi(\alpha_{fsmstmt})(d) \cup \phi(\alpha'_{fsmstmt})(d) \tag{8.4}$$

$$\phi(s)(d) = DEF_s \cup (d \setminus KILL_s) \tag{8.5}$$

In (8.5), we have the basic kernel of data-flow equations. We defined ϕ such that it returns our transfer function over the complete lattice of sets of defining statements for each statement s. DEF_s correspond to s if s is a definition and $KILL_s$ is composed by all other statement defining the same variable that s defines. On the way, we consider sequential composition (8.3) and, a missing term in classical formulations, parallel statements (8.4).

The structured approach comes into play when we want to account for the effect of structured statements such as **`waitin`** with nested regions. First of all, we need to be able to input any initial condition coming from upper layers. Therefore, we extend (8.1) as

$$ins(bb_i) = ins_0(bb_i) \cup \bigcup_{p \in preds(bb_i)} outs(p) \tag{8.6}$$

The term $ins_0(bb_i)$ accounts for initial conditions of each basic block, which can be seen as having another basic block entering bb_i.We note the data-flow solution under initial conditions I as $ins(bb)|_I$ and $outs(bb)|_I$.

A more subtle extension concerns Equation (8.5), which we extend as follows.

$$\phi(s)(d) = DEF_s(d) \cup (d \setminus KILL_s(d)) \tag{8.7}$$

It allow us to get a recursive data-flow solution for each statement when handling structured ones such as **`waitin`**.

Let $\overline{r} = [r]$ be the list of regions and $HCFG_r = (BB_r, E_r)$ the HCFG with entry basic block and the set of exit basic blocks ebb_r and $\widehat{BB}_r$, respectively. Let s be a **`waitin`** statement shown hereafter

$$s = \texttt{wait this in } [r] \ \overline{recv} \ \overline{then}$$

then we define its set of definitions as

$$DEF_s(d) = \bigcup_{r \in R} \bigcup_{bb_r \in \widehat{BB}_r} outs(bb_r)|_{I_r(d)} \tag{8.8}$$

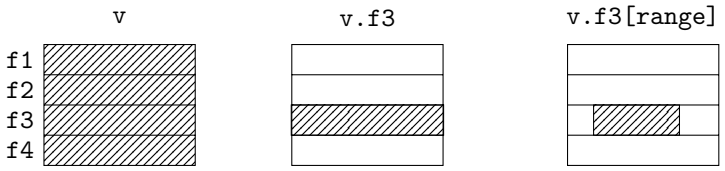


FIGURE 8.3: Granularity levels of variable definitions: object, association and element levels

where $I_r(d)$ is the initial conditions ins_0 on BB_r

$$ins_0(bb) = \begin{cases} d & \text{if } bb = ebb_r \\ \emptyset & \text{otherwise} \end{cases}$$

and $KILL_s$ corresponds to all other statements defining the same set of variables defined on $\overline{r}$. Finally, DEF_s and $KILL_s$ are constant functions for straight-forward definitions other than **`waitin`** statements.

The analysis takes into account variables within the following IR grammar

$$\begin{aligned} \langle varexpr \rangle &::= \langle varexpr \rangle \text{ '.' } \langle var \rangle \\ &\mid \langle var \rangle \\ \langle var \rangle &::= \langle id \rangle \text{ '[' } \langle arithexprs \rangle + \text{ ']'} \\ &\mid \langle id \rangle \end{aligned}$$

In the formulation, we handle different granularities of variable definitions. Considering fine grain definitions, i.e., inside object associations (or fields), allow us to optimize out dead definitions of associations when dealing with non-global variables. Figure 8.3 describes three granularity levels concerning variable definitions where we handle the first two ones while being pesimistic over the last one. That is, a definition of an element inside a multi-valued association, `v.f3[range]` for instance, is considered to kill any incoming definition of such association, `v.f3`, regardless of the indexing access. However, a definition of `v.f3` does not kill other definitions of `v`.

8.3.2 Message Dependency Analyzer

In Chapter 5, we presented the definition of interfaces where we indicate valid input and output messages as well as precedence relations between them. The IR preserves such information and the message dependency analyzer relies on it to build a set of related send and receive expressions inside the state machine. For instance, consider the running example of Listing 8.1.

```
1   sendfrom this this.r mult<Float> rcst,
2   sendfrom this this.g mult<Float> gcst,
3   sendfrom this this.b mult<Float> bcst;
4
5   wait this for _m1_multed = recv Float'multed<Float>,
6                _m2_multed = recv Float'multed<Float>
7     then when _m1_multed, _m2_multed goto UBB_MULTING;
8
9  UBB_MULTING:
10  sendfrom this _m1_multed.res add<Float> _m2_multed.res;
11  wait this for _m3_multed = recv Float'multed<Float>
12             , _m4_added  = recv Float'added<Float>
13    then when _m3_multed, _m4_added goto UBB_MULTADDING;
```

LISTING 8.1: Send-Receive relations: a running example

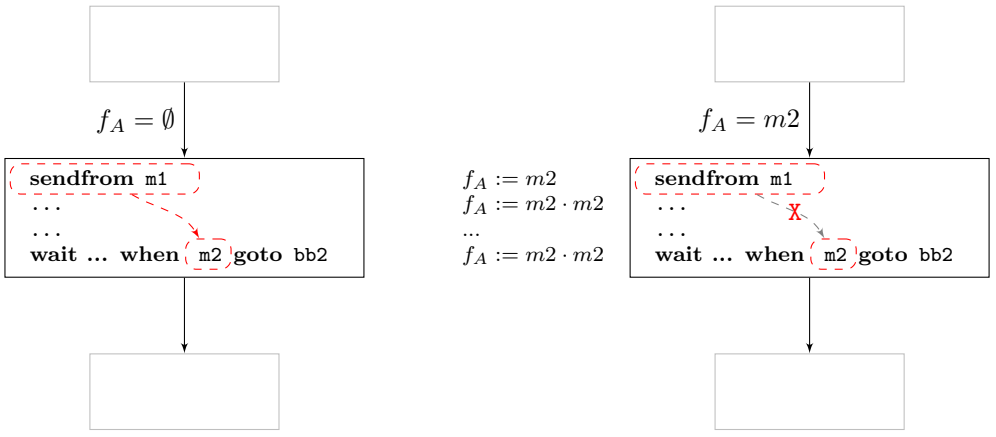
At Line 3, we have three parallel sending actions to scalar `Int` and according to its interface, it sends back three `multed<Float>` messages. Two of them are captured in **`wait`** expression at Line 6 and the last one is taken at Line 11. Which `mult<Int>`, and hence which sending expression, is related to which `multed<Float>` is not specified; it is actually not necessary to do so in the computation being modeled.

Another related question concerns incoming FIFO states. 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 8.4 shows the FIFO state problem when trying to relate send and receive expressions. It describes an $HCFG_A$ of object A communicating with S where $(m_1, m_2) \in \mathcal{T}_S$. In 8.4(a), the FIFO of A is an empty list and we can safely assume that **`sendfrom`** `m1` is related to the 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 another **`sendfrom`**, the described one is not going to be related to next **`wait`** anymore (see 8.4(b)). Therefore, whether **`sendfrom`** `m1` is related to the next receive expression or not will depend on the incoming FIFO state.

In order to relate send and receive expressions, we need to precisely track the FIFO state of two communicating objects. Given that it is an undecidable problem in general as explained earlier, we narrow the analysis to objects comunicating with other objects that specify precedence relations for all of its valid input messages.

We propose to build a set $\mathcal{F}$ of terms (or equations) for each branch of the $HCFG_A$ in the state machine such that each equation f represents a possible FIFO state. The set $\mathcal{F}$ describing such state is inductively defined as follows:

- Message $m \in \mathcal{F}$
- Variable $v \in \mathcal{F}$



(a) Empty FIFO (b) Initialized Incoming FIFO

FIGURE 8.4: Send-receive problem

- Concatenation: If $f, f' \in \mathcal{F}$ then $f \cdot f' \in \mathcal{F}$
- Alternation: If $f, f' \in \mathcal{F}$ then $f + f' \in \mathcal{F}$
- Multiset union: If $f, f' \in \mathcal{F}$ then $f \uplus f' \in \mathcal{F}$
- Receive: If $f, f' \in \mathcal{F}$ then $f \leftarrow f' \in \mathcal{F}$

The meaning of above operators is the standard one with the exception of operator receive, $f \leftarrow f'$. It is intended to represent an element-wise matching operation between two FIFO states where f represents a *waiting state* and f' the *input state*. Under the hypothesis of instantaneous communications introduced in section 8.1, we are going to consider **`wait`** and **`sendfrom`** statements as generating expressions of waiting lists of messages and input list of messages, respectively.

Definition 8.1. Given $\mathcal{F}$, we define a suffix partial order on FIFO states. An empty state is lower than any other: $\emptyset \leq f$. By induction, we define the ordering as follows $\forall f, f', m, m' \in \mathcal{F}$, $f \cdot m \leq f' \cdot m'$ iff $m = m'$ and $f \leq f'$.

Remark 8.2. The suffix partial order is preserved under right concatenation: $f \leq f' \Rightarrow f \cdot g \leq f' \cdot g$ for any g.

We start by defining an abstraction function working on basic blocks that looks only on sending actions $\psi : BB \rightarrow \mathcal{F}$

$$\psi(\alpha_{fsmstmt} \ ; \ \alpha'_{fsmstmt}) = \psi(\alpha_{fsmstmt}) \cdot \psi(\alpha'_{fsmstmt}) \tag{8.9}$$

$$\psi(\alpha_{fsmstmt} \ , \ \alpha'_{fsmstmt}) = \psi(\alpha_{fsmstmt}) \uplus \psi(\alpha'_{fsmstmt}) \tag{8.10}$$

$$\psi(\textbf{sendfrom} \ sender \ target \ m \ \overline{param}) = \{\overleftarrow{m}^S\} \tag{8.11}$$

$$\psi(s) = \emptyset \quad \text{otherwise} \tag{8.12}$$

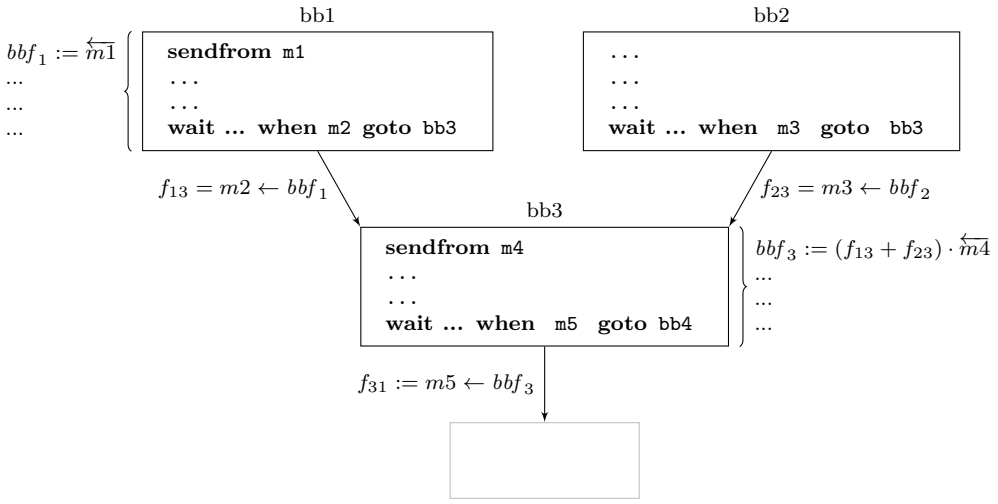


FIGURE 8.5: Message dependency analysis: FIFO equations

It translates a basic block by intepreting pushes into the FIFO (8.9) of parallel (unordered) (8.10) simple sends (8.11) while ignoring everything else that does not involve communication (8.12).

Then, consider a statement terminator **`wait`**

$$w = \textbf{wait this } \overline{recv}\ \overline{then}$$

where $\overline{then} = [then_i]$ and $when_i(w)$ gives a multiset of messages corresponding to the triggers listed in the **`when`** clause of $then_i$.

Given ψ and a wait terminator w_{bb_i} of basic block bb_i, we define a term for each $f_{ij} \in E_A$ such that

$$f_{ij} = when_j(w_{bb_i}) \leftarrow \sum_{bb_h \in preds(bb_i)} f_{hi} \cdot \psi(bb_i) \tag{8.13}$$

Definition (8.13) establishes the expression value of all output branches of basic block bb_i following the FIFO state changes that bb_i introduces by itself plus all possible incoming branches.

Figure 8.5 sketches the analysis. At the entry of bb3, we do not know the incoming FIFO state. Therefore, we consider alternatively both of them, f_{13} and f_{23}. The first **`sendfrom`** of bb3 will push the response to $m4$ to the current unknown state. The only output branch of bb3 will forward such state after having consumed message $m5$. We proceed this way for all edges of the control flow as described by (8.13).

The analysis is based on the idea that receive equations can be reduced. That is, we can *apply* the receive operator recursively according to the following rules

$$\begin{array}{rll} (m \cdot f) \leftarrow (m' \cdot f') & = f \leftarrow f' & \text{if } m = m' \\ (m \cdot f) \leftarrow (m' \cdot f') & = (m \cdot f) \leftarrow f' & \text{if } m \neq m' \\ \emptyset \leftarrow f' & = f' & \text{otherwise} \end{array} \tag{8.14}$$

The first rule of (8.14) matches head messages of waiting and input FIFO states. The second one implements the dropping semantics proposed in Chapter 6 and the last one says that if there is nothing to wait for then we forward the input state as is. We recall that each message has a generating expression, for instance **`sendfrom`** `m4` generates $\overleftarrow{m}_4$. A matching between two messages will imply a relation between two generating expressions. For the sake of clarity, generating expressions are not shown in the analysis. There is a missing rule not shown in (8.14): $f \leftarrow \emptyset$. It stands for *dead-lock* as we are waiting on a empty input state.

Remark 8.3. The partial function $f(x) = cst \leftarrow x$ is monotone and decreasing: $g = cst \leftarrow f' \Rightarrow g \leq f'$. It follows directly from its definition (8.14).

Algebraic rules. Some basic algebraic properties hold on $\mathcal{F}$

$$\begin{array}{ll} (a + b) \leftarrow c & = (a \leftarrow c) + (b \leftarrow c) \\ c \leftarrow (a + b) & = (c \leftarrow a) + (c \leftarrow b) \\ (a + b) \cdot c & = (a \cdot c) + (b \cdot c) \\ c \cdot (a + b) & = (c \cdot a) + (c \cdot b) \end{array} \tag{8.15}$$

meaning that the alternation can be pushed upwards in order to reach a canonical form.

Lemma 8.4. *We note $\overline{\mathcal{F}}$ to the set $\mathcal{F}$ not containing terms of the form $f + f'$. Then, from the system of equations given by* (8.13) *and using rules* (8.15)*, f_{ij} can be rewritten to the following form*

$$c = \sum_i \overline{f}_i$$

called canonical such that $\overline{f}_i \in \overline{\mathcal{F}}$.

Proof. By applying the second rule on (8.13), we directly obtain the canonical form of f_{ij}

$$\begin{aligned} f_{ij} = when_j(w_{bb_i}) &\leftarrow \sum_{bb_h \in preds(bb_i)} f_{hi} \cdot \psi(bb_i) \\ &= \sum_{bb_h \in preds(bb_i)} when_j(w_{bb_i}) \leftarrow f_{hi} \cdot \psi(bb_i) \end{aligned}$$

□

Lemma 8.5. *The canonical form is preserved under variable replacement.*

Proof. We replace f_{hi} by its defining equation into (8.13)

$$f_{ij} = when_j(w_{bb_i}) \leftarrow \sum_{bb_h \in preds(bb_i)} \left(when_i(w_{bb_h}) \leftarrow \sum_{bb_g \in preds(bb_h)} f_{gh} \cdot \psi(bb_h) \right) \cdot \psi(bb_i)$$

By applying the second rule, we obtain

$$f_{ij} = \sum_{bb_h \in preds(bb_i)} when_j(w_{bb_i}) \leftarrow \left(\sum_{bb_g \in preds(bb_h)} when_i(w_{bb_h}) \leftarrow f_{gh} \cdot \psi(bb_h) \right) \cdot \psi(bb_i)$$

and third rule give us

$$f_{ij} = \sum_{bb_h \in preds(bb_i)} when_j(w_{bb_i}) \leftarrow \sum_{bb_g \in preds(bb_h)} (when_i(w_{bb_h}) \leftarrow f_{gh} \cdot \psi(bb_h)) \cdot \psi(bb_i)$$

to finally apply second rule again

$$f_{ij} = \sum_{bb_h \in preds(bb_i)} \sum_{bb_g \in preds(bb_h)} when_j(w_{bb_i}) \leftarrow (when_i(w_{bb_h}) \leftarrow f_{gh} \cdot \psi(bb_h)) \cdot \psi(bb_i)$$

By induction hypothesis we have that f_{gh} has also a canonical form under variable replacement. Following the same reasoning as before, we achieve again a canonical form. Therefore, f_{ij} has a canonical form under variable replacement. □

An additional rule holds under the guarantee of state machine progress

$$(f \leftarrow f') \cdot f'' = f \leftarrow (f' \cdot f'')$$

Assuming progress we have $g = f \leftarrow f'$ such that $g \leq f'$ (suffix partial order) because receive applies on head messages of f' only, by receive rules (8.14). Then $f \leftarrow (f' \cdot f'')$ is also going to apply on head messages of f' only and both expressions will yield the same result, $g \cdot f''$.

There are still two important questions around this analysis. The first one is related to type correctness of equations. The last one concerns the resolution of $F = \{f_i\}$ equations in presence of control-flow loops, which implies solving of recurrence equations.

Type correct receives. According to equations (8.9) and (8.10), ψ should output a list of a multiset of messages. However, the application rule of receive operator is defined to work on lists of messages. Therefore, we transform multisets of messages to the combinatorial alternation of lists before evaluation.

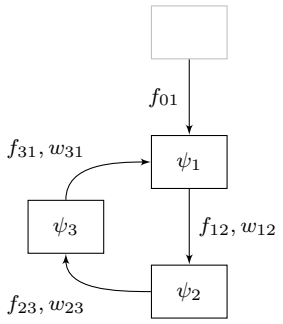


FIGURE 8.6: Loop in the control-flow and its FIFO equations

Recurrence in FIFO states. Given the system of equations $F = \{f_{ij}\}$ obtained from the $HCFG$, we solve F in topological order if there is no loop in the control-flow. Otherwise we need to deal with recurrence equations and initialization issues.

Consider a rewritten version of equation (8.13)

$$f_{ij} = w_{ij} \leftarrow \sum_{bb_h \in preds(bb_i)} f_{hi} \cdot \psi_i$$

where $w_{ij} = when_j(w_{bb_i})$ and $\psi_i = \psi(bb_i)$. Figure 8.6 shows a simple looping instance where we get the following set of equations

$$\begin{aligned} f_{12} &= w_{12} \leftarrow (f_{31} + f_{01}) \cdot \psi_1 = w_{12} \leftarrow f_{31} \cdot \psi_1 + w_{12} \leftarrow f_{01} \cdot \psi_1 \\ f_{23} &= w_{23} \leftarrow f_{12} \cdot \psi_2 \\ f_{31} &= w_{31} \leftarrow f_{23} \cdot \psi_3 \end{aligned} \tag{8.16}$$

Solving for f_{31}, we have

$$f_{31} = w_{31} \leftarrow (w_{32} \leftarrow (w_{12} \leftarrow (f_{31} + f_{01}) \cdot \psi_1) \cdot \psi_2) \cdot \psi_3$$

and using rules (8.15) to move up alternation

$$\begin{aligned} ep &= w_{31} \leftarrow (w_{32} \leftarrow (w_{12} \leftarrow f_{01} \cdot \psi_1) \cdot \psi_2) \cdot \psi_3 \\ lp &= w_{31} \leftarrow (w_{32} \leftarrow (w_{12} \leftarrow f_{31} \cdot \psi_1) \cdot \psi_2) \cdot \psi_3 \\ f_{31} &= ep + lp \end{aligned}$$

Naturally, f_{31} accounts for both paths in the loop, the loop itself lp and its entry ep. To solve the looping term lp, we have to find an initialisation value for f_{31}. The initalising value cannot be randomly taken. Both terms impose a set of constraints on the values of f_{31}. In general, we will always obtain the alternation form under variable replacement as shown by lemma (8.5).

By using remark (8.3), we deduce from (8.16) the following constraints

$$f_{31} < f_{23} \cdot \psi_3 \quad f_{23} < f_{12} \cdot \psi_2 \quad f_{12} < f_{31} \cdot \psi_1 \quad f_{12} < f_{01} \cdot \psi_1$$

Applying (8.2) to $f_{23} < f_{12} \cdot \psi_2$, we obtain

$$f_{23} \cdot \psi_3 < f_{12} \cdot \psi_2 \cdot \psi_3$$

and given that $f_{31} < f_{23} \cdot \psi_3$ we deduce that

$$f_{31} < f_{12} \cdot \psi_2 \cdot \psi_3$$

Following this reasoning, we get

$$f_{31} < f_{31} \cdot \psi_2 \cdot \psi_3 \cdot \psi_1$$
$$f_{31} < f_{01} \cdot \psi_2 \cdot \psi_3 \cdot \psi_1$$

The first inequality corresponds to the loop path and the last one to its entry. We can see that the former holds for any $n \geq 1$ such that

$$f_{31} < \psi_{231}^n$$

where $\psi_{231} = \psi_2 \cdot \psi_3 \cdot \psi_1$.

We note the initial value of f as f^0. If f_{31} is less than both terms, i.e., $f_{31} < f_{31}\psi_{231}^n \wedge f_{31} < f_{01}\psi_{231}$, then it is necessarily lower than its greater lower bound on the semi-lattice of suffix partially ordered FIFO states

$$f_{31}^0 = \psi_{231}^n \sqcap f_{01}\psi_{231}$$

Note that f_{01} is a constant value. Finally, we can safely take this value as its initial one to solve F.

Generalizing this result, we define the initialization value f_{ij}^0 as

$$f_{ij}^0 = \psi_p^n \sqcap \prod_{h \in preds(j) \wedge h \neq i} f_{hj} \cdot \psi_p$$

where p is a specific loop path of basic blocks in the backwards direction starting at bb_j.

In order to show a running instance of the analysis, we consider the control-flow of Pixel shown in Figure 8.7 where we highlighted all communication primitives.

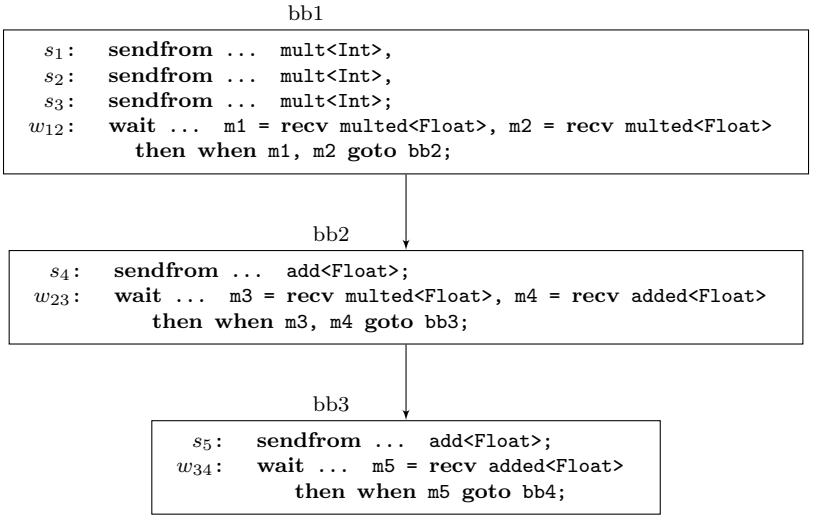


FIGURE 8.7: Pixel control-flow with communication primitives only

We extract the following constant definitions where we denote lists with square brackets

$$\begin{aligned}
\psi_1 &= [\{(s_1, multed), (s_2, multed), (s_3, multed)\}] & \psi_3 &= [\{(s_5, added)\}] \\
\psi_2 &= [\{(s_4, added)\}] \\
w_{12} &= [\{(r_1, multed), (r_2, multed)\}] & w_{34} &= [\{(r_5, added)\}] \\
w_{23} &= [\{(r_3, multed), (r_4, added)\}]
\end{aligned}$$

Here we expose generating expressions, which are not shown in the analysis to facilitate its presentation. For instance, we have three parallel sends s_1, s_2 and s_3, in basic block bb1 and the next two ones s_4 and s_5 in basic blocks bb2 and bb3, respectively. Writing down receive expressions (see (8.13)), it follows

$$f_{12} = w_{12} \leftarrow \psi_1 \quad f_{23} = w_{23} \leftarrow f_{12} \cdot \psi_2 \quad f_{34} = w_{34} \leftarrow f_{23} \cdot \psi_3$$

Before solving, we need to make type correct receives. Given that operands of a receive operator must be a list of messages and we have instead a list of multisets of messages, we convert them to corresponding combinatorial alternation of message lists as discussed earlier. For instance, w_{23} is rewritten to

$$w_{23} = (r_4, multed) \cdot (r_5, added) + (r_5, added) \cdot (multed, r_4)$$

We show hereafter an alternative obtained by rewriting w_{12} and ψ_1 in f_{12}

$$\begin{aligned}
f_{12} &= [(r_1, multed), (r_2, multed)] \leftarrow [(s_2, multed), (s_1, multed), (s_3, multed)] \\
&= [(s_3, multed)]
\end{aligned}$$

By matching messages, we deduce the following set SR of related expressions from f_{12}

$$SR = \{(s_2, r_1), (s_1, r_2)\}$$

and we get the remaining input messages $f_{12} = [(s_3, multed)]$ as an input to solve f_{23}. In this example, the resolution gives us the following set

$$SR = \{s_1, s_2, s_3\} \times \{r_1, r_2\} \cup \{s_1, s_2, s_3\} \times \{r_3\} \cup \{(s_4, r_4), (s_5, r_5)\} \tag{8.17}$$

SR materialize all possible related send and receive expressions. It is the work of the *Transaction Selector* to choose a certain relation over another one. For instance, from SR we know that $(s_1, r_1) \in SR$ will relate first **`sendfrom`** to the first **`recv`** expression in bb1. But we also have $(s_1, r_3) \in SR$, which is also valid because a parallel send operation does not specify any arrival order of responses — even under the hypothesis of instantaneous communications — and we may receive the message triggered by s_1 at receive expression of basic block bb2.

8.3.3 Transaction Selector

The transaction selector uses the result of our message dependency analyzer to make a deterministic choice between non-deterministic relations of send-receives. We briefly discussed the result of the analysis on the example of Figure 8.7. Due to non-determinstic properties of parallel send, we may have to choose between a certain subset of relations. Consider again the result (8.17). We observe that there are receive expressions related to more than one sending actions, s_1 for instance, but according to Pixel code flow s_1 can only be related to only one receive if we need to generate code with deterministic behavior. However, in some cases keeping all relations from sends to receives may be a valid situation. In the example of Figure 8.8, we assume $SR' = \{(s'_1, r'_1), (s'_2, r'_1)\}$ where s'_1 and s'_2 correpond to **`sendfrom`** at bb1 and bb2, respectively, and r'_1 represents the receive at bb3. In this particular case, even if we have a single receive related to two sending actions, we do not need to choose between both pairs as they are control-flow dependent. For this reason, we refer to control-based dependencies in the traditional Single Static Assignment (SSA) sense, i.e., as a ϕ dependency, and to dependencies introduced by non-determinism as a π *dependency*. Note that the converse situation may also exist in a control-flow dependent manner, i.e., sending expressions with multiple related receives, which we described in Figure 8.9.

Therefore, the Transaction Selector needs to decide whether a subset of relations are control-dependent or not. If they are, we must preserve them, otherwise we need to choose between one of them to produce a deterministic behavior.

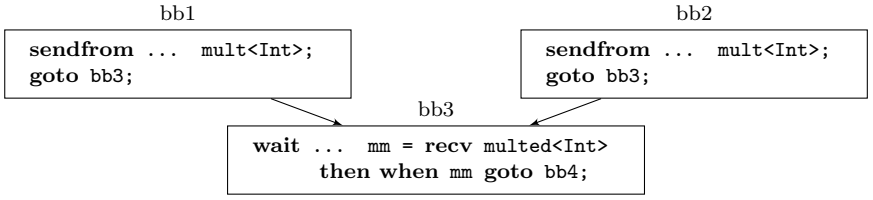


FIGURE 8.8: ϕ dependency of sending-receive pairs

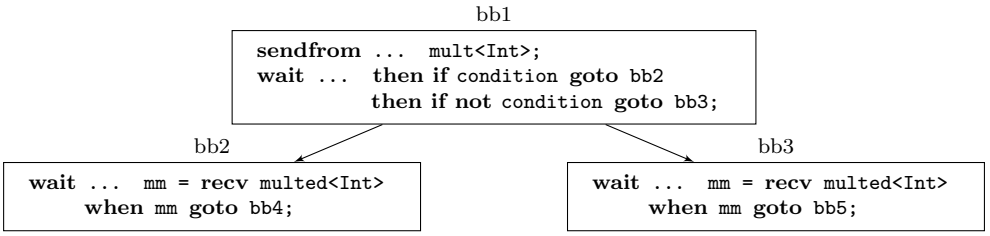


FIGURE 8.9: Splitting of sending actions

Let $M = (S \uplus R, T)$ be the output graph of our message dependency analyzer on the set of send S and receives R where $T \subseteq S \times R$.

Lemma 8.6. *A receive related to more than one send necessarily represents ϕ or π dependencies.*

Proof. The only way for a receive to be related to more than one send is to have a term of the form

$$f = r \leftarrow (s + s')$$

where the alternation can only come from (8.13), i.e. control-flow dependent, or the combinatorial expansion of multisets of messages introduced by parallel sends in (8.10), thus a non-determinstic relation (see Type correctness of receives). □

From the proof of Lemma (8.6), we deduce the following important corollary

Corollary 8.7. *Let $(s, r), (s', r) \in T$ where s and s' are parallel sends then $r = \pi(s, s')$.*

Therefore, given any M we are able to discover π dependencies and as a consequence of Lemma (8.6) we have ϕ dependencies as well. The following lemma will help us expose the structure of M among parallel sends.

Lemma 8.8. *Let $r \in R$ and $s, s' \in S$ two parallel sends such that (s, r), $(s', r) \in T$ and $\exists r' \mid (s, r') \in T$ then $(s', r') \in T$.*

Proof. If s and s' are parallel and related to r, then they produce the same reply message m which is captured by r. Because they are parallel, we will have an alternation expression $m_s +$

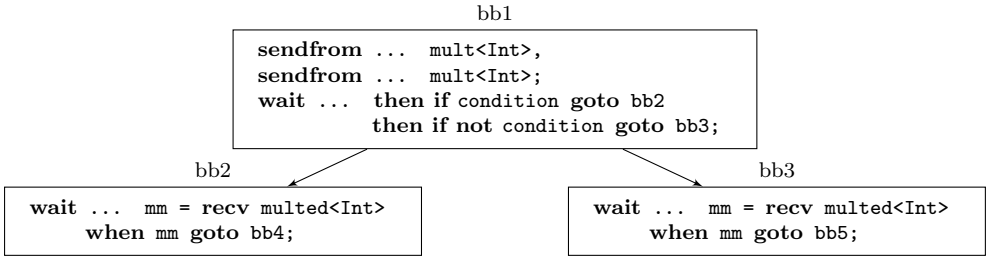


FIGURE 8.10: Shared send action

$m_{s'}$ according to (8.10). If there exists an r' on message m related to s, by the alternation term and our algebraic rules (8.15), it must also be related to s'. □

Following this lemma, we have a complete bipartite graph between any set of parallel sends and its respective receives.

From Corollary (8.7), we can see each particular receive as a "shared" resource among parallel sends that form a π dependency. However, it does not say anything about the other direction, i.e., a single send related to more than one receive. If we come back to the example of Figure 8.7, we see that there must be a bijective relation between senders expressions of `mult<Int>` and receivers of `multed<Float>` because each reply message, consequence of a particular send, becomes unavailable once it has been consumed by one receive. In other words, receivers do not *share* sending actions.

On the other hand, Figure 8.10 shows the converse situation, which is a slightly different version of Figure 8.9. We have an additional sending action s_2 in parallel with the first one s_1. Assuming an empty FIFO state at bb1, we have that both receives at bb2 and bb3, r_1 and r_2, are π related to s_1 and s_2, respectively. Here, we can let receives connected to the same sending action after having removed π dependencies. Thus, s_1 and s_2 are shared between them. In conclusion, we need to distinguish between what we called *concurrent* receives from the sending perspective. We introduce this notion in the following definition.

Definition 8.9. Let $r_1, r_2 \in R$ be two different receives. They are *concurrent* iff $r_2 \in dom(r_1) \vee r_1 \in dom(r_2)$ where $dom : R \to 2^R$ is the dominance set of receives.

We say that a receive r dominates r' if all control-flow paths from the state machine root to the wait statement that contains receive r', pass through the wait of r. Clearly, if one receive dominates the other then common parallel sending actions related to both receives cannot be shared as the first one will pick an available send and make it unavailable for the following one.

Based on Corollary (8.7) and given Lemma (8.8) combined with Definition (8.9), we implemented a simple algorithm to eliminate concurrent dependencies from any send-receive graph

M. Let us first define the equivalence class of parallel sends and concurrent receives as follows

$$\|s\| = \{s' \in S \mid parallel(s, s')\}$$
$$\langle r \rangle = \{r' \in R \mid r' \in dom(r) \vee r \in dom(r')\}$$

We show in Algorithm 1 the implementation of the π dependency elimination. Basically, for each receive $r \in R$, we take its incoming parallel send classes and we choose one send for each class. Then, we remove concurrent relations according to Corollary (8.7) and Definition (8.9).

Algorithm 1: Remove π dependencies

Data: Send-Receive graph $M = (S \uplus R, T)$
Result: Graph M without π dependencies
for $r \in R$ **do**
  $Parallels \leftarrow \{\|s\| \mid (s, r) \in T\}$;
  for $P \in Parallels$ **do**
    if $|P| \leq 1$ **then**
      /* No π dependencies here */
      continue;
    end
    $s \leftarrow choice(P)$;
    /* From Corollary (8.7), $P \times \{r\}$ are π dependencies, hence we must remove relations from r to any other concurrent send s' in P */
    $CS \leftarrow \{(s', r) \mid s' \in P \wedge s \neq s'\}$;
    /* From Lemma (8.8) and Definition (8.9), we need to remove concurrent receives to the chosen s */
    $CR \leftarrow \{(s, r') \in T \mid r' \in \langle r \rangle \wedge r \neq r'\}$;
    $T \leftarrow T \setminus (CS \cup CR)$;
  end
end

We show a running example in Table 8.1 where each step corresponds to one receive (and its associated parallel sends) of results (8.17). At the first step, we pick receive r_1 with related parallel sending s_1, s_2 and s_3, we choose relation (s_1, r_1) and remove (s_2, r_1), (s_3, r_1). Afterwards, we look at r_2, choose (s_3, r_2) and remove (s_3, r_3). Finally, r_3 gets related to the last send s_2. Note that the final set of relations is a one-to-one relation between send and receives because

8.3.4 Scalar Constants

As introduced in Chapter 5, $\langle \text{HOE} \rangle^2$ scalars are objects that communicate through message-passing in the same manner that user-defined objects. The scalar constant analysis tracks constant associations wherever they are defined, either inside **`fsm`** or **`creator`**. It simply detects **`create`** expressions of scalar objects as the following

Receive / ParSend	T
r_1 / (s_1, s_2, s_3)	$\{(s_1, r_1)\} \cup \{s_2, s_3\} \times \{r_2, r_3\} \cup \{(s_4, r_4), (s_5, r_5)\}$
r_2 / (s_2, s_3)	$\{(s_1, r_1), (s_3, r_2)\} \cup \{s_2\} \times \{r_3\} \cup \{(s_4, r_4), (s_5, r_5)\}$
r_3 / (s_2)	$\{(s_1, r_1), (s_3, r_2), (s_2, r_3)\} \cup \{(s_4, r_4), (s_5, r_5)\}$
r_4 /	$\{(s_1, r_1), (s_3, r_2), (s_2, r_3)\} \cup \{(s_4, r_4), (s_5, r_5)\}$
r_5 /	$\{(s_1, r_1), (s_3, r_2), (s_2, r_3)\} \cup \{(s_4, r_4), (s_5, r_5)\}$

TABLE 8.1: Selecting between non-deterministic send-receive relations

```
this.cst = create Int.int 10;
```

We show in the next section that constants can be made local to the state machine and automatically unboxed if possible. Using local variables instead of associations may produce unused associations that can be optimized out afterwards.

8.4 Transformations

Based on the analyses presented in the previous section, we built a set of transformations to achieve IR patterns that are known to have efficient code translations. Before introducing our transformations, where most of them rely on the presented analyses, let us start by presenting required mathematical tools and notations concerning the representation of $\langle \text{HOE} \rangle^2$ index sets.

8.4.1 Index Sets

We define the denotation of an $\langle indexset \rangle$, $[\![.]\!] : \langle indexset \rangle \to \Sigma \to S$ where $S = \bigcup_i S_i$ is the finite union of labeled parametric polyhedral sets $S_i : \mathbb{Z}^n \to \left(\Sigma \times 2^{\mathbb{Z}^d}\right)$ represented as follows

$$S_i(\mathbf{s}) = \{\ell_i(\mathbf{x}) \mid \mathbf{x} \in \mathbb{Z}^d \wedge \exists \mathbf{z} \in \mathbb{Z}^e : A\mathbf{x} + B\mathbf{s} + D\mathbf{z} \geq \mathbf{c}\}$$

with $A \in \mathbb{Z}^{m \times d}$, $B \in \mathbb{Z}^{m \times n}$, $D \in \mathbb{Z}^{m \times e}$, $\mathbf{c} \in \mathbb{Z}^m$ and $2^{\mathbb{Z}^d}$ the power set of $\mathbb{Z}^d$ and Σ a finite set of labels. That is, we have m inequations with parameters $\mathbf{s}$, existentially quantified variables $\mathbf{z}$ and constants $\mathbf{c}$.

We define also the polyhedral relation $R = \bigcup_i R_i$ where $R : \mathbb{Z}^n \to \left(\Sigma \times 2^{\mathbb{Z}^{d_1}}\right) \times \left(\Sigma \times 2^{\mathbb{Z}^{d_2}}\right)$, which is the union of basic relations R_i

$$R_i(\mathbf{s}) = \{\ell_1(\mathbf{x_1}) \to \ell_2(\mathbf{x_2}) \mid \mathbf{x_1} \in \mathbb{Z}^{d_1}, \mathbf{x_2} \in \mathbb{Z}^{d_2}, \exists \mathbf{z} \in \mathbb{Z}^e : A_1\mathbf{x_1} + A_2\mathbf{x_2} + B\mathbf{s} + D\mathbf{z} \geq \mathbf{c}\}$$

Using polyhedral sets, we define the index domain of certain IR statements. The index domain of a parallel IR statement $\mathcal{D} : \langle parstmt \rangle \rightarrow S$ is defined as

$$\begin{aligned}
\mathcal{D}(\alpha_{parstmt}\texttt{ , }\alpha'_{parstmt}) &= \mathcal{D}(\alpha_{parstmt}) \cup \mathcal{D}(\alpha'_{parstmt}) \\
\mathcal{D}(\texttt{forall[}\alpha_{indexset}\texttt{] }\alpha_{send}) &= [\![\alpha_{indexset}]\!]_{\alpha_{send}} \\
\mathcal{D}(\texttt{forall[}\alpha_{indexset}\texttt{] }\alpha_{update}) &= [\![\alpha_{indexset}]\!]_{\alpha_{update}} \\
\mathcal{D}(\alpha) &= \emptyset \quad \text{otherwise}
\end{aligned}$$

using a shorthand notation $[\![.]\!]_\ell$ for the denotational function. Additionally, we define the polyhedral union set of a **waitin** statement as the union of all its indexed region domains. Let w be a waitin statement of the form

$$w = \texttt{wait this in } \overline{region}\ \overline{recv}\ \overline{then}$$

and we note $\mathcal{D}_r \in S$ the polyhedral set of region $r \in \overline{region}$ such that $\mathcal{D}_r(\texttt{[}\alpha_{indexset}\texttt{]}\ \{\alpha_{fsmstmt}\}) = [\![\alpha_{indexset}]\!]_r$ if we have an indexed region, or $\mathcal{D}_r = \emptyset$ otherwise.

In addition, we model read and write accesses using polyhedral relations for send, receive and update statements following theirs operands. An access refers to a certain IR variable. Therefore, let us consider again the grammar of variables

⟨*varexpr*⟩ ::= ⟨*varexpr*⟩ '**.**' ⟨*var*⟩
| ⟨*var*⟩

⟨*var*⟩ ::= ⟨*id*⟩ '**[**' ⟨*arithexpr*⟩+ '**]**'
| ⟨*id*⟩

Let s be an statement with associated index domain $\mathcal{D}(s) = S(\mathbf{x})$ and $R^i : \mathbb{Z}^n \rightarrow \prod_i \left(\Sigma \times 2^{\mathbb{Z}^{d_1}}\right)$, we define an interpretation $\mathcal{I} : \langle var\_expr \rangle \rightarrow R^i$ of accessed variables such that the relation between s and the access is the labeled polyhedral n-ary relation $A = S(\mathbf{x}) \rightarrow \mathcal{I}(v)$.

$$\mathcal{I}(\mathbf{v}) = \{\mathbf{v}()\} \tag{8.18}$$

$$\mathcal{I}(\mathbf{v}\texttt{[}\alpha_{arith\_exprs}\texttt{]}) = \{\mathbf{v}(\mathbf{x}) \mid \mathbf{x} = \alpha_{arith\_exprs}\} \tag{8.19}$$

$$\mathcal{I}(\alpha_{var\_expr}\texttt{ . }\alpha_{var}) = \mathcal{I}(\alpha_{var\_expr}) \times \mathcal{I}(\alpha_{var}) \tag{8.20}$$

Definitions (8.18) and (8.19) are known access models in existing polyhedral tools [80, 81], while (8.20) allow us to model structural accesses. We extract read and write accesses out of **fsm** statements in terms of labeled polyhedral relations. For instance, let s be the following statement

```
forall[i: 1 <= i and i < 32]
  sendfrom this this.a[i-1] mult<Int> this.a[i];
```

then we have that $\mathcal{D}(s) = \{\mathbf{s}(i) : 1 \leq i \wedge i < 32\}$ with read associations

$$A_1 = \mathcal{D}(s) \rightarrow \mathcal{I}(\texttt{this}) = \{\mathbf{s}(i) \rightarrow \mathbf{this}() \mid 1 \leq i \wedge i < 32\}$$
$$A_2 = \mathcal{D}(s) \rightarrow \mathcal{I}(\texttt{this.a[i-1]}) = \{\mathbf{s}(i) \rightarrow \mathbf{this}() \rightarrow \mathbf{a}(i_0) \mid 1 \leq i \wedge i < 32 \wedge i_0 = i - 1\}$$
$$A_3 = \mathcal{D}(s) \rightarrow \mathcal{I}(\texttt{this.a[i]}) = \{\mathbf{s}(i) \rightarrow \mathbf{this}() \rightarrow \mathbf{a}(i) \mid 1 \leq i \wedge i < 32\}$$

For the manipulation of polyhedral sets and relations and its operations, we used the Integer Set Library (ISL) [82]. Structural accesses like A_2 or A_3 can be modeled in ISL using nested spaces.

8.4.2 Dead Code Elimination

This pass removes unused definitions which are either performed at the **creator** or the **fsm** scope, regardless of their declaration scope (associations or local variables). Note that the only way to pass definitions from **creator** to **fsm** is through associations. Therefore, we should be able to follow definitions and its corresponding uses at different scopes. Data dependencies between different hierarchical scopes are handled by our data-flow analysis shown in Section 8.3.1. Given that we also take into account structural accesses, we can easily detect unused associations by consulting the DefUse chain analysis. The encapsulation property of $\langle \text{HOE} \rangle^2$ objects guarantees the correctness of this transformation. Associations are not exported to external objects which means that they cannot be used outside the context of the object under analysis.

8.4.3 Dead Associations Elimination

In this pass, we remove undefined associations from the object type definition. We rely on the same fact as before, object associations cannot be defined at the outside of the current object because of visibility constraints.

```
  forall[i: 0 <= i and i < 512]
    sendfrom[i] this this.pixels[i] getY<>;
  goto GETTING_Y;

GETTING_Y:
  wait this for
    msg_takeY = recv[i: 0 <= i and i < 512] takeY<Float>
    then when msg_takeY goto UPDATE_CH
    then if msg_takeY.all goto FINAL;

UPDATE_CH:
  this.ychannel[i] = msg_takeY.y;
  goto GETTING_Y;
```

LISTING 8.2: Wait-all Loop

```
  forall[i: 0 <= i and i < 512]
    sendfrom[i] this this.pixels[i] getY<>;
  goto GETTING_Y;
GETTING_Y:
  wait this in [i: 0 <= i and i < 512]
  {
      wait this for msg_takeY = recv[i] takeY<Float>
        then when msg_takeY goto UPDATE_CH;
    UPDATE_CH:
      this.ychannel[i] = msg_takeY.y;
  } then goto FINAL;
```

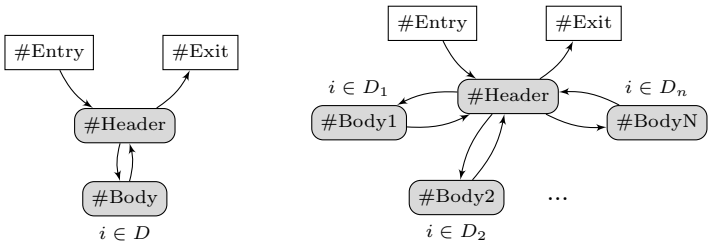
LISTING 8.3: Indexed Region

8.4.4 Rewriting Broadcasts

This pass transforms what we call *wait-all* loops into indexed regions. Wait-all loops are loops that wait for all indexed messages under a certain domain to arrive and quit the main loop afterwards. For instance, consider the loop of Listing 8.2 taken from the Image example. The loop groups basic blocks `GETTING_Y` and `UPDATE_CH` and it forms a wait-all loop. The wait branch at Line 7 waits for all `takeY<>` messages on the specified range of index values. Given that the arrival order of messages is not specified, the loop can be "parallelized" into an indexed region. Listing 8.3 illustrates the transformation where a new indexed region is introduced at Line 5. Note that parallel regions inside composite states receive their own copy of the incoming message. The message dropping semantics guarantees that regions not concerned by messages with wrong index values will drop them.

The loop pattern is shown in Figure 8.11(a) where D is the integer set denoted by the index domain at receive expression. We rapidly see that it can be extended to loops with multiple bodies as shown in Figure 8.11(b) under the following condition

$$\bigcap_i D_i = \emptyset$$



(a) Wait-all with single body

(b) Wait-all with multiple body

FIGURE 8.11: General case of wait-all loops

```
  forall [i: 0 <= i and i < 32]
    sendfrom[i] this this.table[i] Float'fadd<Float> this.cst10;
  goto DIVIDING;

DIVIDING:
  wait this for m = recv[i: 0 <= i and i < 32] Float'fadded<Float>
    then when m if i < 16  goto UPDATE_L
    then when m if i >= 16 goto UPDATE_R
    then if m.all goto FINISHED;

UPDATE_L:
  this.new_table[0, i] = m.res;
  goto DIVIDING;

UPDATE_R:
  this.new_table[1, i - 16] = m.res;
  goto DIVIDING;
```

LISTING 8.4: Guarded branching

where D_i is the intersection of reception index domain and the corresponding transition guard leading to $Body_i$.

To illustrate the general case, we show a simple looping example in Listing 8.4. Here we jump to different basic blocks, `UPDATE_L` and `UPDATE_R`, according to guards of **`then`** branches and store the message content on an association called `new_table`. The receive gives us a context domain $C = \{i \mid i \leq 0 \wedge i < 32\}$, then it follows two branching domains $W_1 = \{i \mid i < 16\}$, $W_2 = \{i \mid i \geq 16\}$, from which we can deduce the following indexed region domains

$$D_1 = C \cap W_1 \quad D_2 = C \cap W_2$$

Additionally, we have that $D_1 \cap D_2 = \emptyset$ as required by the transformation. Therefore, we transform Listing 8.4 into two disjoint indexed regions with domains D_1 and D_2, respectively.

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


```
GETTING_Y:
wait img in [i: 0 <= i and i < 512] {
⇒sendfrom[i] img img.pixels[i] getY<>;
  wait img for msg_takeY = recv[i] takeY<Float>
    then when msg_takeY goto UPDATE_CH;
  UPDATE_CH:
  img.ychannel[i] = msg_takeY.y;
} then goto FINAL;
FINAL:
```

FIGURE 8.12: Loop fusion: Fusioning broadcast into indexed region

8.4.5 Basic Block Fusion

In this pass we merge two basic blocks linked through an unconditional jump. Given two basic blocks BB_1 and BB_2, we merge them iff $preds(BB_2) = \{BB_1\}$ and BB_1 jump unconditionally to BB_2, i.e., via a **goto** statement.

8.4.6 Loop Fusion

The loop fusion pass moves indexed statements into indexed regions. In order to move an indexed send we must be sure that, among other contraints discussed later, its domain is equivalent to the one of the indexed region. Let s_1 be an indexed send with iteration domain $\mathcal{D}(s_1)$. Assume, we have a **waitin** w_1 statement with only one indexed region

$$w_1 = \textbf{wait this in } r_1 \ \overline{recv} \ \overline{then}$$

with iteration domain $\mathcal{D}(r_1)$ and a basic block $bb = s_1; \ w_1$. Then, we should be able to move s_1 into r_1 iff $\mathcal{D}(s_1) \subseteq \mathcal{D}(r_1) \wedge \mathcal{D}(r_1) \subseteq \mathcal{D}(s_1)$. As an example, consider the transformation shown at Figure 8.12. Given that the marked sending is under the same index domain as the next indexed region, then we are able to move the former into the latter.

Applicability of the transformation

If this transformation seems straighforward a priori, in general we need to handle **`waitin`** statements with multiple indexed regions and different kinds of statements to move into. In addition to the number of statements to handle and given our computational model, an important fact to consider is concurrency. Let s be a statement containing only one indexed expression and a **`waitin`** statement with two indexed regions r, r' such that $\mathcal{D}_s = \mathcal{D}_r$ and $\mathcal{D}_r \cap \mathcal{D}_{r'} = \emptyset$. In this case, we should be able to move s into region r. The indexed expression s is concurrent by itself (forall semantics) and it will remain as such even inside region r. The point is that before the transformation, s and r, r' were sequentially ordered but after the transformation, we put s in concurrency with all statements inside r'. Moreover, each particular instance of region r, say r_i, might also run into concurrency problems. This one is more subtle and relies on the fact that we do not specify any particular execution order between different instances of an indexed region, i.e., they are composed asynchronously. Thus *inter-iteration data dependencies* may not be safe in general without explicit synchronization through message passing. Assuming synchronized inter-iteration dependencies on s and r, we must guarantee that s_i does not introduce non-synchronized inter-iteration dependencies. To illustrate the discussion with an example, consider the invalid transformation shown in Figure 8.13. The indexed region makes a simple parallel copy between two arrays, `a` and `b`. Note that if we naïvely insert the first **`sendfrom`** into the following indexed region, as shown in the transformed code, we may run into concurrency problems because **`sendfrom`** introduces an inter-iteration dependency, $i \mapsto i - 1$. If region, say 2, is faster than 3, then we endup by sending message `mult<Int>` to the new value of `a[2]` in region 3, which is `b[2]`, because the update statement in region 2 occured before sending in region 3. On the other hand, the region execution order $i \mapsto 31 - i$ is guaranteed to preserve the initial behavior but we do not support scheduling hints into our current framework. In conclusion, we must check for inter-iteration dependencies before proceeding with the transformation to avoid race conditions.

Given the access relations introduced in Section 8.4.1, we can decide if two statements s_1, s_2 have inter-iteration dependencies by computing, $R_{12} = A_{s_1}^{-1}(A_{s_2})$ and constructing the new *delta* set $\Delta R_{12} = \{\mathbf{x} - \mathbf{y} \mid \forall \mathbf{x} \rightarrow \mathbf{y} \in R_{12}\}$. For instance, consider the read access `this.a[i-1]` of s_1 and the write one `this.a[i]` in s_2. From both accesses, we have

$$A_{s_1} = \{\mathbf{s_1}(i) \rightarrow \mathbf{this}() \rightarrow \mathbf{a}(i_0) \mid 1 \leq i < 32 \wedge i_0 = i - 1\}$$
$$A_{s_2} = \{\mathbf{s_2}(i) \rightarrow \mathbf{this}() \rightarrow \mathbf{a}(i) \mid 1 \leq i < 32\}$$

Then we apply $A_{s_1}^{-1}(A_{s_2}) = RW_{12} = \{\mathbf{s_2}(i) \rightarrow \mathbf{s_1}(i_0) \mid 1 \leq i < 32 \wedge i_0 = i + 1\}$ and compute the delta set $\Delta RW_{12} = \{1\}$, which provides a proof of an inter-iteration dependency.

```
forall[i: 1 <= i and i < 32]
  sendfrom this this.a[i-1] mult<Int> this.a[i];
wait this in [i: 1 <= i and i < 32] {
  this.a[i] = this.b[i];
} then goto NextState;
```


```
wait this in [i: 1 <= i and i < 32] {
  sendfrom this this.a[i-1] mult<Int> this.a[i];
  this.a[i] = this.b[i];
} then goto NextState;
```

FIGURE 8.13: Invalid loop fusion because of concurrent inter-iteration dependencies

Algorithm 2: Move indexed expressions into indexed regions

Data: Current basic block BB with waitin branch
Result: Modified BB

```
/* Split BB into BB' and its waiting branch w'                    */
```

$BB', w' \leftarrow BB$;
$R \leftarrow$ regionsOf(w');
for $stmt \in reverse(BB')$ **do**
  if $stmt \notin \langle parstmt \rangle$ **then**
    break;
  end
  if $\exists \alpha \in stmt$ *such that* α *is not an indexed expression* **then**
    break;
  end
  for $r \in R$ **do**
    if $\mathcal{D}_r \nsubseteq \mathcal{D}_{stmt}$ **then**
      continue;
    end
    $exprs \leftarrow \mathcal{D}_{stmt} \cap \mathcal{D}_r$;
    copy $exprs$ into region r;
  end
  remove $stmt$ from BB
end

Implementation

Our current implementation has the following limitations

1. We look for statements that fit perfectly into the list of indexed regions.

2. The statement is composed only by indexed expressions.

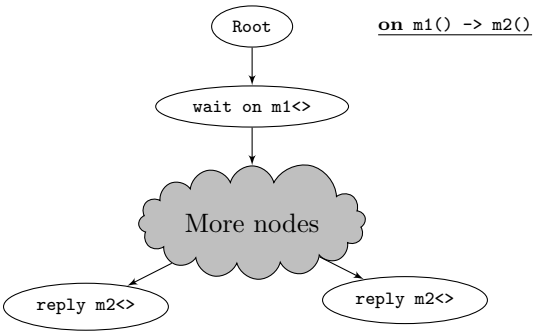


FIGURE 8.14: Wait-reply paths

Let us consider a parallel statement as a list of indexed sends or updates $\alpha_{fsmstmt} = [\alpha^i_{iexpr}]$. Then, the first constraint can be formalized as follows

$$\bigcup_{r \in \overline{region}} \mathcal{D}_r = \mathcal{D}(\alpha^i_{iexpr})$$

for all α^i_{iexpr}. Then, each parallel expression can be distributed among all the indexed regions assuming that it does not introduce any inter-iteration dependency among them. Algorithm 2 describes the copy of parallel statements into multiple indexed regions.

We show in Figure 8.12 a running example of the loop fusion transformation.

8.4.7 Inlining

This pass inlines one object state machine into another, called *inlinee* and *inliner*. In the context of state machines, the inlining procedure depends on multiple factors: precise knowledge of the current state of the inlinee from the inliner perspective, known send-receive relations and well delimited sub-state machine to inline.

Well delimited sub-state machine. Let A be a IR object with an input-output message relation at its interface level: `on m1() -> m2()`. Necessarily, there is at least one closed *sub-state machine* of A containing a **`wait`** statement with a receive expression on m_1 and a reachable **`reply`** statement on m_2. We call such path a *wait-reply* path. Note that it is the role of the interface to indirectly guarantee its existence. Figure 8.14 sketches the case of a single wait reaching multiple replies. We see that the wait *dominates* both receives, and hence it defines a well-delimited sub-state machine that encloses the actions needed to fulfill the exposed transaction. Although closely related, the general case concerning arbitrary wait-reply paths is a problem of interface verification rather than inlining. We did not go further into it and limit our current implementation to one wait and one reply paths.

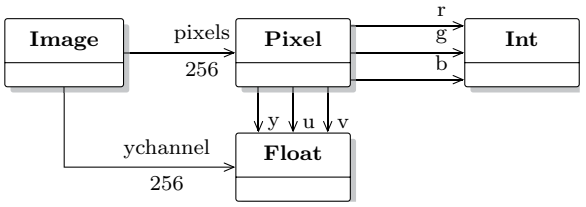


FIGURE 8.15: Image object model

Inlinee object never escapes. This constraint guarantees that the inliner is the only producer and inlinee its only consumer. Therefore, if the inlinee waits on a given message, which is exposed on its interface, it will only react to the inliner sending actions and no one else. This provides us a precise knowledge of the inlinee state at the beginning of the transaction.

The inliner completes the transaction. The inliner candidate must contain some send-receive relation. We rely on the Transaction Selector (see Section 8.3.3), which relies on the Message Dependency Analyzer, to collect those relations and following the above constraints we decide to inline the state machine.

For instance, consider again the Image object model of Figure 8.15 where the interface of Pixel object is defined as

```
object Pixel
  interface
    on getY() -> takeY(Float)
```

If the Image object has a completing transaction with Pixel objects and meets the other presented constraints as well, then we inline Pixel state machine into the Image one. The pass computes the region of states enclosed by reception of `getY<>` till reply of `takeY<Float>` in order to inline it into Image. The running transformation is shown in Figure 8.16.

8.4.8 Folding of Operational Transitions

In Chapter 5, we presented the $\langle \text{HOE} \rangle^2$ operational transitions and mentioned that they can be transformed into "in-place" operations. We built around this idea the $\langle \text{HOE} \rangle^2$ scalar operations. This pass concretizes such idea with the help of the Transaction Selector by folding operational transitions. We have shown in the precedent section that send-receive relations can be inlined and we show here that we can materialize them using the **`applyon`** operator if possible, joining the idea of the operational view of scalar message exchanges introduced in Chapter 5. Consider the extension to scalar Int interface of Listing 8.5. The interface says that such send-receive relation can be folded into an abstract operation called `multOp` which takes an Int and a Float

```
GETTING_Y:
wait img in [i: 0 <= i and i < 512] {
⇒sendfrom[i] img img.pixels[i] getY<>;
⇒wait img for msg_takeY = recv[i] takeY<Float>
    then when msg_takeY goto UPDATE_CH;
  UPDATE_CH:
  img.ychannel[i] = msg_takeY.y;
} then goto FINAL;
FINAL:
```


```
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;
```

FIGURE 8.16: Inlining Pixel object into Image

```
scalar Integer
  interface
    on Mult(Float) -> Multed(Float) ~> multOp
```

LISTING 8.5: New interface entry for Int objects

```
wait this in [i: 0 <= i and i < 512] {
⇒Float __new_var_2 = applyon this.pixels[i].r multOp this.pixels[i].rcst;
  wait this.pixels[i]
    then goto UBB_MULTING;
  UBB_MULTING:
  Float __new_var_1 = __new_var_2;
  [...]
} then goto Final;
```

LISTING 8.6: Transaction Folding

and gives us a Float object. In order to actually fold it, we transform it into an in-place operation by means of the abstract operation **`applyon`**. For instance, from the output of the inlining transformation shown in Figure 8.16, we see that the first **`sendfrom`** and the next **`wait`** are related. The Message Dependecy Analyzer spots this relation and the Transaction Selector forward such information, which is exploited by this pass in order to finally fold it and transform the code into the one shown in Listing 8.6.

8.4.9 Dead Wait Rewriter

The Transaction Folding pass removes send and receives statements and adds the corresponding in-place operation. As a consequence, the transformation may produce *dead waits*. Dead waits have only one `then` branch without receive nor guard specifications. We transform them into its equivalent, and simpler, statement **`goto`** as shown hereafter.

```
wait this then goto NextBB;  ➡  goto NextBB;
```

Afterwards, the basic block fusion pass may eliminate them if the Control-Flow Graph (CFG) meet the necessary conditions. More optimization possibilities may be discovered after fusioning basic blocks, such as Loop Fusion.

8.4.10 Unboxing Types

We say that $\langle \text{HOE} \rangle^2$ scalars represents *boxed* primitive types because they provide state machine semantics, equivalent to other objects, to simple scalar values. Boxed scalar types guarantees a clean and homogeneous interaction within the language with other objects. However, it implies a state machine for every single scalar value in the language runtime implementation, which is clearly too cumbersome if we want to improve $\langle \text{HOE} \rangle^2$ application performances. In general, we aim to have as much unboxed scalars and operations as possible in the final state machine model.[2] In our IR, boxing and unboxing operations are implicit and concern exclusively **`applyon`** operations. That is, the **`applyon`** operator requires unboxed types as operands and returns unboxed types. Thus, their operands are implicitly unboxed before applying the operation, and its output is boxed depending on the type of the target variable.

This pass is decoupled into two steps: Scalar Constant Unboxing and Unboxing of Definitions. First of all, we try to use unboxed constants if possible. For instance, the scalar creation

```
Int cst = create Int.int 10;
```

is unboxed to

```
int cst = 10;
```

iff all uses of `cst` require an unboxed scalar, i.e., they correspond to operands of **`applyon`**. Here, we rely on two key analyses: the DefUse chain (Section 8.3.1) and Scalar Constants (Section 8.3.4).

The next step is more general. We look for definitions where the left-hand side is an unboxed type but its right-hand side is not. The constraints remain the same, we verify that all uses of the

[2] In case there is no native support, either at the language or architecture level, for the message passing semantics.

applyon output require an implicit unboxing by consulting our DefUse chain analysis. We are able to unbox definitions such as those presented in Figure 8.6 and rewritten hereafter.

```
Float __new_var_2 = applyon this.pixels[i].r multOp this.pixels[i].rcst;
Float __new_var_1 = __new_var_2;
```

into

```
float __new_var_2 = applyon this.pixels[i].r multOp this.pixels[i].rcst;
float __new_var_1 = __new_var_2;
```

iff all uses `__new_var_1` and `__new_var_2` turn to be unboxed.

8.5 Contributions

We presented the IR optimizing compiler, its analyses and optimizations. The goal of our optimizing compiler is to achieve specific IR patterns, which are known to have efficient implementations in the target language. For this purpose, we introduced a set of analyses and transformations with some of them as extensions to already known analysis in the compilation domain, e.g. the structural reaching definition analysis. This analysis solves the reaching definition problem over structural, or hierarchical, statements. It allows us to keep the hierarchical information coming from the frontend language, $\langle \text{HOE} \rangle^2$, while enabling classical optimizations along the hierarchy of the state machine such as dead code and association elimination, automatic unboxing.

Based on the interface concept introduced in the $\langle \text{HOE} \rangle^2$ language, we developed a new analysis to relate send and receive communication primitives in a modular manner. The analysis takes into account the specification of many messages per transition, of different kind and multiplicity. On the way, it also allows us to detect dead-lock conditions under well-defined hypotheses. We also exposed its algebraic properties, on which we root the next analysis called the Transaction Selector. The Transaction Selector detects non-deterministic relations between send and receives in order to choose one valid deterministic behavior. We introduced the equivalent concept of ϕ nodes at the message exchange level and a new dependency kind called π to maintain non-determinsitic relations.

One of the main contributions of the $\langle \text{HOE} \rangle^2$ language is the introduction of indexed messages and regions. Using polyhedral analysis, we developed transformations to rewrite traditional $\langle \text{HOE} \rangle^2$ broadcasts made of wait-all loops into indexed regions. Afterwards, we study the problem of loop fusioning in the context of asynchronous composition of indexed regions. Enabling loop fusion allowed us to optimize further and fold indexed message exchanges into in-place

operations, according to the operational view of transactions presented in Chapter 5. We also developed an automatic unboxing pass on the context of the IR language.

In the following chapter, we validate our optimizing compiler and show experimental results.

Part III

Validation

Chapter 9

Experimental Results

Contents

In order to provide a clear interpretation of our experimental results, we first describe the underlying message-passing runtime and its interaction with Highly Heterogeneous, Object-Oriented, Efficient Engineering $\langle$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 one. In the process, we will consider multiple optimization levels to highlight the impact of the optimizing compiler, and to demonstrate the generated code quality. We conclude our experiences with an adaptation of our optimizing chain to support General-Purpose computing on Graphics Processing Units (GPGPU) code generation.

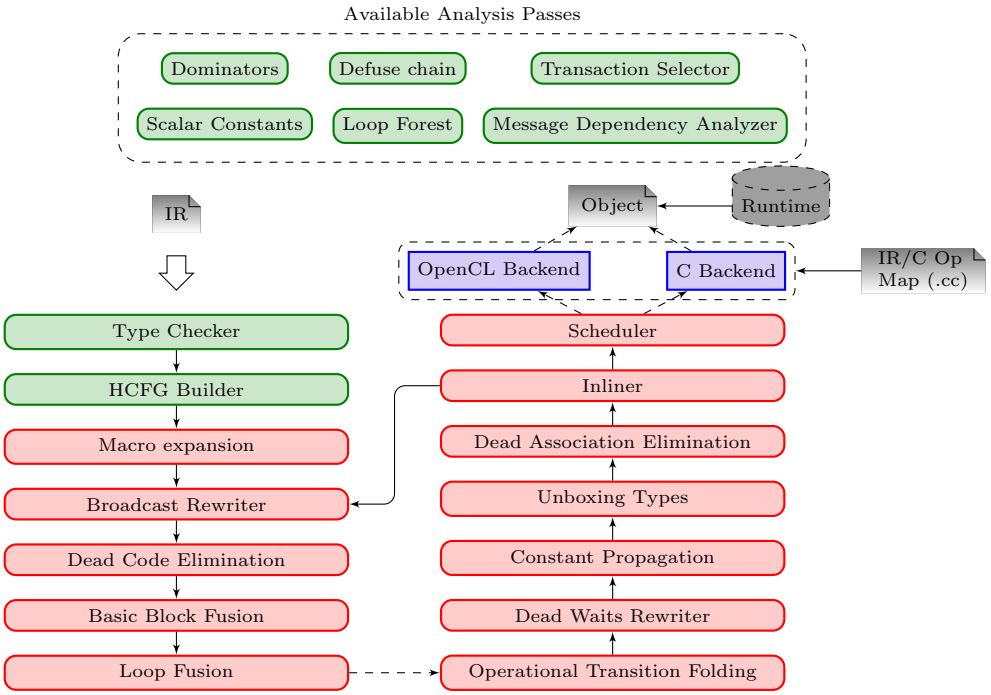


FIGURE 9.1: Optimizing pipeline

9.1 Code Generation Strategy

We recall the optimizing pipeline presented at Chapter 9 in Figure 9.1. Two backends are currently available: C and OpenCL. Each backend translates the Intermediate Representation (IR) of Chapter 7 into C/OpenCL code relying in a custom runtime support we developed.

9.1.1 Runtime

The compiler toolchain emits one C file per $\langle$HOE$\rangle^2$ object and this file embodies the optimized code of this object. The generated C code can call back the runtime for multiple purposes, like creating or destroying objects, sending messages, waiting for incoming messages, and initialization and termination of objects. Listing 9.1 shows the main callback routines. Our runtime implementation is based on a widely available thread library called *QThreads* [83]. Qthreads is a lightweight locality-aware user-level threading runtime. Each $\langle$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.

The runtime starts the application by calling the `main` constructor of a root $\langle$HOE$\rangle^2$ object defined by the programmer. This root object represents the system that creates all the initial objects of the $\langle$HOE$\rangle^2$ application. The basic template for the entry point of an $\langle$HOE$\rangle^2$ application is shown in Listing 9.2. Then, an object providing a `main` creator is called by the runtime entry to start the application. We associate to each $\langle$HOE$\rangle^2$ object a simple C structure that exposes

```
/* Runtime initialization, application running and finalization */
void hoe2_rt_init(hoe2_rt_options);
void hoe2_rt_run(hoe2_object *);
void hoe2_rt_finish();

/* Object creation and destruction */
void hoe2_rt_new_object (hoe2_object *obj);
void hoe2_rt_done_object(hoe2_object *obj);

/* Send and receive primitives */
void hoe2_rt_send( hoe2_object  *src
                 , hoe2_object  *dest
                 , hoe2_message *msg
                 , size_t        msg_size
                 );

hoe2_message *hoe2_rt_recv(hoe2_object *src);

hoe2_message *hoe2_rt_recv_in( hoe2_object *obj
                             , hoe2_region *regions[]
                             , size_t       regions_size
                             );
```

LISTING 9.1: $\langle$HOE$\rangle^2$ runtime

```
/* External object constructor */
struct hoe2_object *main_obj_ctr();

int main(int argc, char **argv) {
  hoe2_rt_options opt;
  parse_args(argc, argv, &opt);

  hoe2_rt_init(opt);
  /* Call the external creator of the main object */
  struct hoe2_object *main_obj = main_obj_ctr();
  /* Run it */
  hoe2_rt_run(main_obj);
  return 0;
}
```

LISTING 9.2: $\langle$HOE$\rangle^2$ application entry point

a (circular) First In-First Out (FIFO) buffer for communications, a main thread pointer for the state machine and its corresponding locks (see Listing 9.3).

9.1.2 Code Generation

Most IR statements have fixed translation rules, such as updates, object creation, message send and receive, except for **`applyon`** expressions. $\langle$HOE$\rangle^2$ models defining operational transitions must provide a mapping between **`applyon`** operations and their corresponding translation. The mapping is controlled by the programmer via a "cc" mapping file (see Figure 9.1). For instance, the following $\langle$HOE$\rangle^2$ expression

```
struct hoe2_object {
  char name[127];
  struct hoe2_fifo *__fifo;
  aligned_t (*fsm)(struct hoe2_object *);
  aligned_t step;
  aligned_t region_step;
};
```

LISTING 9.3: $\langle$HOE$\rangle^2$ object: C implementation

```
  Int result = applyon addOp(p)
```

is translated into the IR statement as

```
  Int result = applyon this Int'addOp p
```

which is then mapped to a native operation, addition in this particular case, that works on the unboxed type.[1]

```
  Int result = __obj_new_Int_int(this->value + p->value);
```

Indexed Regions: AST Generation from Polyhedra. $\langle$HOE$\rangle^2$ implements asynchronous indexed regions using affine constraints. We consider a coarse-grain asynchronous composition of regions, i.e., each region is viewed as an atomic indexed statement, which means that there is no specific execution order of each i*th* regions. Then, we generate for-loops in a lexicographic order, which is a valid order among all possible ones under asynchronous composition. Such scheduling is valid under a strong hypothesis: *no explicit inter-iteration synchronization*. In other words, the indexed region does not contain inter-iteration message dependences imposing a communication-dependent order. Thereby, if we have an indexed region such as

```
wait this in
  [i, j: 0 <= i and i < 256 and 0 <= j and j < 256] { // Indexed region
    [...]
  } then goto NEXT;
```

We choose to emit

```
// Lexicographically scheduled indexed region
for (int i = 0; i < 256; i+= 1) {
  for (int j = 0; j < 256; j+= 1) {
    [...]
  }
}
goto NEXT:;
```

[1]The boxed operation is optimized out by the Unboxing pass.

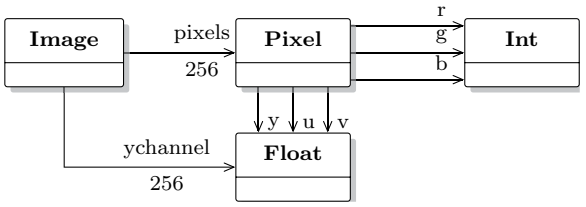


FIGURE 9.2: Image object model

Above translation is a lexicographic scheduling of the indexed region. Faulty loop carrying data-dependencies due to concurrency does not concern the code generator and should be detected beforehand.

While the generation of a for loop from index domain $\{(i,j) : 0 \leq i < 256 \wedge 0 \leq j < 256\}$ is straightforward because loop bounds and stride are given, we may expect more complex expressions. For instance, consider the domain given by the zigzag traversal (discussed later in this chapter)

$$\{(i,j) : 2 \left\lfloor \frac{i+j}{2} \right\rfloor = i + j \wedge i + j \leq 7 \wedge 0 \leq i < 8 \wedge 0 \leq j < 8\}$$

where $\lfloor . \rfloor$ returns the integer part of its argument. This domain comes from a modulo constraint between i and j, `i + j % 2 = 0`.

In consequence, we implemented a simple version of the algorithm presented by Bastoul in [84]. Recent advances in code generation from polyhedra would be of great benefit to improve detection of loop bounds and strides and, hence avoiding complex control statements in our translation of indexed regions [85].

9.2 Exercising the Optimizing Flow

We exercised the flow over our motivating model example shown at Figure 9.2 where Listing 9.4 shows its actual implementation. The Image object creates RGB pixels (Line 8) and uses parallel sending actions to get all gray values (Line 12). Underneath, the Pixel implements the actual gray conversion starting at creator `RGB` (Line 27). 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 9.5.

We can see how optimizations explained at Chapter 8 are applied:

(a) The message-passing arithmetics are replaced by *in-place* operations — Section 8.4.8.

```
1 object Image
2   has [512] Pixel as pixels
3   has [512] Float as ychannel
4
5   sm ImageSM.
6     creator RGB(rgb: Int[1536]) /
7         { i: 0..pixels.len - 1 }
8         pixels[i] = new Pixel.RGB(rgb[3*i..3*i+2])
9       to GETY
10
11    state GETY.
12      on /: { i: 0..pixels.len - 1 } pixels[i].getY()
13      to GETTING_Y
14
15    state GETTING_Y.
16      on takeY{i}(y: Float) /
17           ychannel[i] = y to GETTING_Y
18      endon [i.all]
19
20 object Pixel
21   interface
22     on getY() -> takeY(Float)
23
24   has Int    as r, g, b
25   has Float  as y, u, v
26   sm PixelSM.
27     creator RGB(rgb: Int[3]) / r = rgb[0]
28                              , g = rgb[1]
29                              , b = rgb[2]
30                              to ComputeY
31     // Wait for getY and launch multiplications
32     state ComputeY. on getY() / : r.mult(0.299)
33                                 , g.mult(0.587)
34                                 , b.mult(0.114)
35                    to Multing
36     // Collect two multiplications and launch the addition
37     state Multing. on multed(v1: Float), multed(v2: Float) / : v1.fadd(v2)
38                   to MultAdding
39     // Collect last multiplications and launch another addition
40     state MultAdding. on multed(v3: Float), fadded(v4: Float) / : v3.add(v4)
41                      to Adding
42     // Reply to the sender of getY() by using keyword "initiator"
43     state Adding. on fadded(result: Float) / y = result: initiator.takeY(y)
44                  to ComputeY
```

LISTING 9.4: Image model

```
1 aligned_t __obj_Image_ImageSM(struct Image *this) {
2   for (int i = 0; ((i >= 0) && (i < 512)); (i+=1)) {
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 9.5: Image object model: Generated C code

(*b*) The broadcast sending of `getY()` to all pixels is translated into an indexed region — Section 8.4.4.

(*c*) The **`sendfrom`** operation is inserted into the new indexed region — Section 8.4.6.

(*d*) The `Pixel`'s state machine are inlined into each indexed region thanks to the interface entry `Pixel`, **`on`** `getY() -> takeY(Float)` — Section 8.4.7.

(*e*) Finally, the indexed region created at (*b*) is translated into an efficient C for-loop.

Given that `Image` is the main object of our running program, its associations are preserved (considered as side-effect actions) and we found remaining boxing and unboxing operations of scalar objects at assignement and computation points of associations, respectively. As defined in Chapter 8, we say that $\langle$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 $\langle$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.

9.3 Metrics and Results

Our optimizing flow aims at optimizing-out as much messages as can be: If we achieve 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 $\langle$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

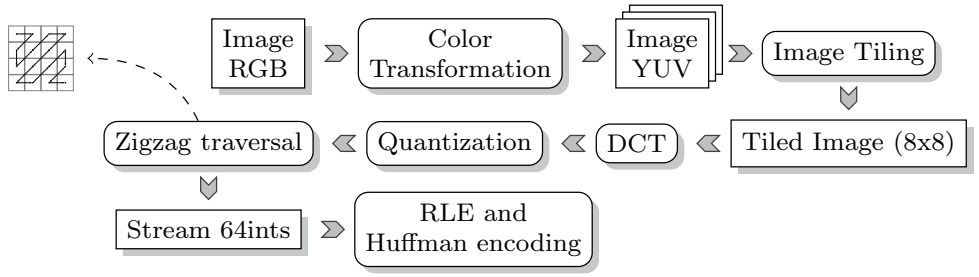


FIGURE 9.3: Phases of the JPEG algorithm

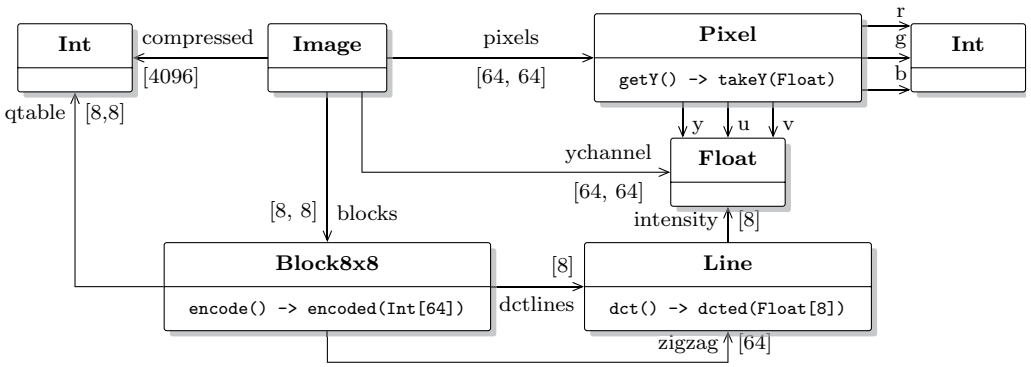


FIGURE 9.4: Image object model implementing JPEG till zigzag traversal

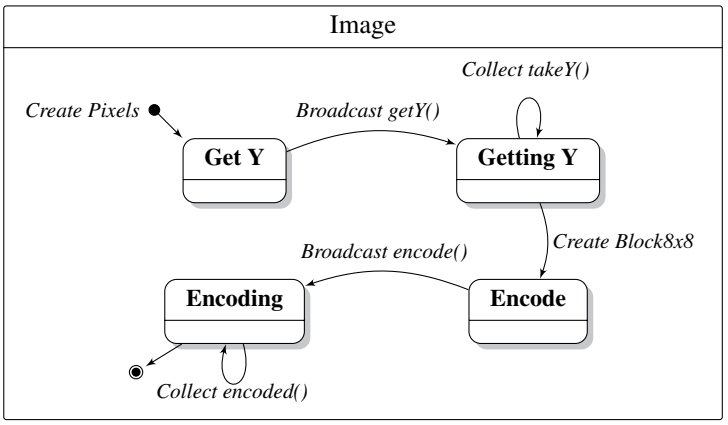
```
state Encode.
  on [] / {i, j: 0 <= i < 8 and 0 <= j < 8}
            blocks[i, j] = new Block8x8(ychannel[8*i..8*i+7, 8*j..8*j+7])
        : {i, j: 0 <= i < 8 and 0 <= j < 8}
            blocks[i, j].encode() // Parallel encoding
  to Encoding
```

LISTING 9.6: Parallel encoding of image blocks

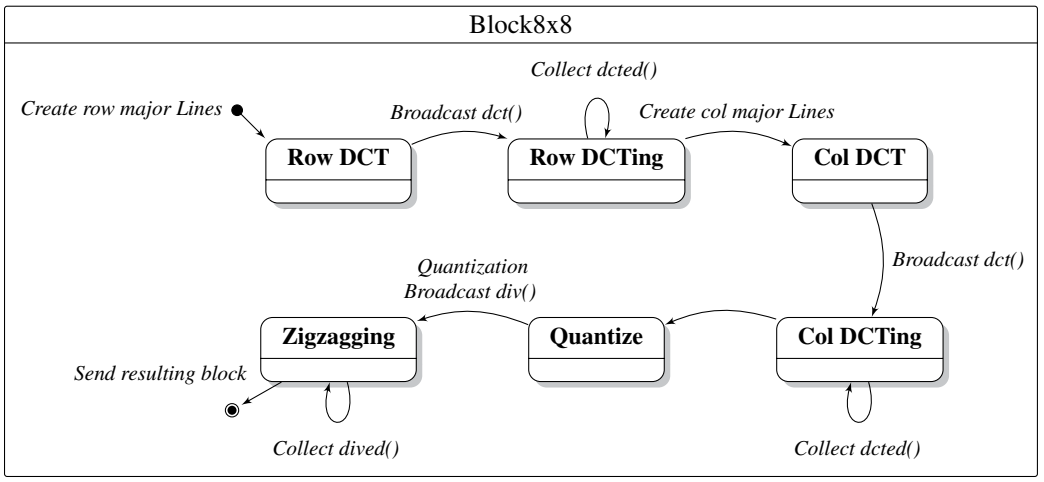
with inlining. Unoptimized code generation gives us good insights on the operations involved into the application and helps us to quantify them. Further optimization levels aim for efficient code generation.

To stress out the toolchain under these optimization levels, we modeled a chain of image transformations taken from the JPEG algorithm shown in Figure 9.3. The model of an image of 64 by 64 pixels is presented in Figure 9.4. After converting RGB pixels to its luminance component, the image needs to be tiled in blocks of 8x8 luminance values. The Discrete Cosine Transform (DCT) is applied in parallel to all blocks following our broadcasting semantics (see Listing 9.6). Descriptive state machines of the Image and Block8x8 object models are shown in Figure 9.5.

We show in Listing 9.6 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,



(a) Image state machine



(b) Block8x8 state machine

FIGURE 9.5: Descriptive state machines inside objects of the JPEG model

as described in [86].[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 RowDCT. on / : {i: 0 <= i < 8} dctlines[i].dct() to RowDCTing
```

Once the DCT is finished, `Block8x8` divides all the resulted 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]) // Parallel division
                to Zigzagging
```

[2]The most used implementation among JPEG encoders.

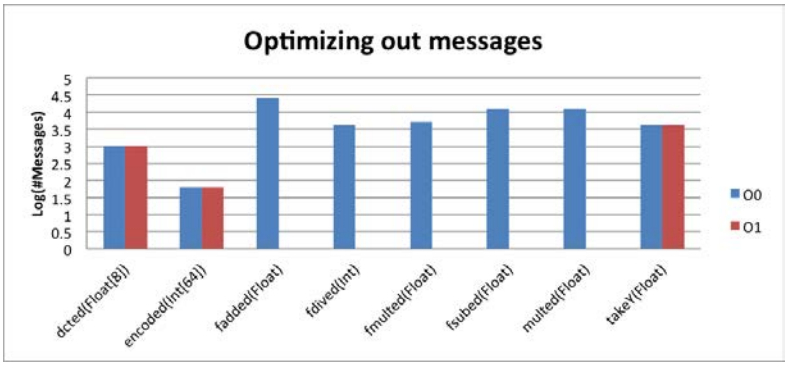


FIGURE 9.6: Number of messages at optimization levels O0 and O1 (O2 eliminates all messages)

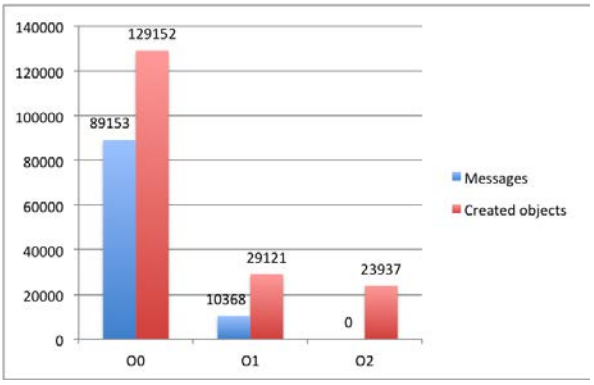


FIGURE 9.7: Number of messages and created objects messages at optimization levels O0, O1 and O2

```
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 9.7: Zigzag traversal state

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 9.7 shows its implementation.

For this extensive implementation, the number of exchanged messages at optimization levels O0 and O1 are shown in Figure 9.6 (logarithmic scale). Level O0 generates naive code, preserving all message exchanges. At O1, the specialization of indexed regions and folding messages into

```
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
                                                          );
}}}}}
```

LISTING 9.8: Slice of the generated C code from zigzag model

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, from only the modeled quantization and zigzag storage, the compiler produces the optimized C code shown at Listing 9.8

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. We use polyhedral code generation to produce C for-loops from our index domains [84]. To get an idea on how powerful high-level models allow the programmer to handle the complexity of deeply nested operations, we show in Listing 9.9 the C for loop structure of the application model automatically generated by the compiler.

Even though the number of messages has been aggressively optimized, the compiler failed to unbox all scalar types. Figure 9.7 compare both results. The results show that there are still too many object creations even at the highest optimization level: 129152 objects at O0, 29121 objects at O1 and 23937 objects at O2. If we take a ratio between different optimizations, we note that O1 improves O0 in terms of object creation by 77% and O2 improves O1 by 17%. The first improvement is due to in-place computations. After send/receive operations has been folded, we create **`applyon`** operations that may enable type unboxing aftewards, thus fewer creation of objects. The 17% at O2 comes from the inlining operation. Indeed, if we inline objects then we do not need to create them anymore. Still more than 23k objects are inserted into the runtime with no-communication at all. Taking a closer look at the problem, we found the following generated code snippet:

```
aligned_t __obj_Image_ImageSM(struct Image *this) {
  for (int i = 0; (i <= 63); (i += 1)) {
    for (int j = 0; (j <= 63); (j += 1)) {
      /* Compute ychannel[i][j] from pixels[i][j] */
  }}
  for (int i = 0; (i <= 7); (i += 1)) {
    for (int j = 0; (j <= 7); (j += 1)) {
      struct Float * block[8][8];
      for (int k = 0; ((k >= 0) && (k < 8)); (k+=1)) {
        for (int l = 0; ((l >= 0) && (l < 8)); (l+=1)) {
          /* Tile ychannel in 8x8 and store to block[k][l] */
      }}
      for (int p = 0; (p <= 7); (p += 1)) {
        struct Float * row[8];
        for (int q = 0; ((q >= 0) && (q < 8)); (q+=1)) {
          /* Extract row p from block and set row*/
        }
        /* Copy from row[0..7] to blocks[i][j].dctlines[p].intensity */
      }
      for (int r = 0; ((r >= 0) && (r < 8)); (r+=1)) {
        for (int s = 0; ((s >= 0) && (s < 8)); (s+=1)) {
          /* Initialize blocks[i][j].qtable[r][s] */
      }}
  }}
  /* Start DCT */
  for (int a = 0; (a <= 7); (a += 1)) {
    for (int b = 0; (b <= 7); (b += 1)) {
      struct Float * dctblock[8][8];
      for (int m = 0; (m <= 7); (m += 1)) {
        /* Compute 1-D DCT on blocks[a][b].dctlines[r].intensity */
        /* and set dctblock[0..7][m] */
      }
      for (int p = 0; (p <= 7); (p += 1)) {
        struct Float * lines[8];
        for (int q = 0; ((q >= 0) && (q < 8)); (q+=1)) {
          /* Extract line p from dctblock and store it into lines */
        }
        /* Copy to blocks[a][b].dctlines[p].intensity from line */
      }
      for (int r = 0; (r <= 7); (r += 1)) {
        /* Compute 1-D DCT on blocks[a][b].dctlines[r].intensity */
        for (int n = 0; ((n >= 0) && (n < 8)); (n+=1)) {
          /* Copy blocks[a][b].dctlines[r].intensity to dctblock[n][r] */
      }}
      /* Quantize and zigzag storage */
      struct Int * zigzag[64];
      for (int u = 1; (u <= 7); (u += 1)) {
        for (int v = (8 - u); (v <= 7); (v += 1)) {
          if ((((-(u) + v) % 2) == 0)) {
            /* Divide and set zigzag */
      }}}
      for (int u = 2; (u <= 7); (u += 1)) {
        for (int v = 7; (v <= 7); (v += 2)) {
          /* Divide and set zigzag */
      }}
      for (int u = 0; (u <= 7); (u += 1)) {
        for (int v = 1; (v <= (7 - u)); (v += 2)) {
          /* Divide and set zigzag */
      }}
      for (int u = 0; (u <= 6); (u += 1)) {
        for (int v = 0; (v <= (6 - u)); (v += 1)) {
          if ((((-(u) + v) % 2) == 0)) {
            /* Divide and set zigzag */
      }}}
      for (int k = 0; ((k >= 0) && (k < 64)); (k+=1)) {
        /* Store zigzag to encoded */
      }
  }}
}
```

LISTING 9.9: For loop structure of the generated JPEG model

```
1 for (int a = 0; (a <= 7); (a += 1)) {
2   for (int b = 0; (b <= 7); (b += 1)) {
3     [...]
4     for (int m = 0; (m <= 7); (m += 1)) {
5       [...]
6       float tmp12 = __new_var_14;
7       float __new_var_33 = tmp10 + tmp11;
8       this->blocks[a][b]->dctlines[m]->d0 = __obj_new_Float_float(
9                                               __new_var_33
10                                            );
11      [...]
12
```

Line 6 is a definition coming from a folded send/receive combined with an automatic unboxing. The next line is translated from an **`applyon`** operation, which is implemented as a C addition. Then, we have a boxing operation to store the final value in the association. This is necessary because we did not unbox association variables yet. That is, we do not modify the object definition itself, excepting the pass that removes dead associations. Ideally, we should be able to greatly reduce the number of created objects if we follow the idea of unboxed associations. Thereby, an object with unboxed associations could be used unboxed as well by composition. In conclusion, the automatic unboxing optimization is fundamental in our language in order to pass from message-driven to in-place operations with efficient structure accesses.

9.4 Application Development: The $\langle$HOE$\rangle^2$ Approach

In order to give the Model-Driven Development (MDD) aspect on which our $\langle$HOE$\rangle^2$ language is embedded, we introduce the $\langle$HOE$\rangle^2$ development process and methodology. The $\langle$HOE$\rangle^2$ development process was extensively studied in [19]. It is composed by four phases: Requirement Analysis, System Analysis, Design and Implementation. Figure 9.8 shows the $\langle$HOE$\rangle^2$ methodology and describes how these phases are combined together following a full MDD flow with an innovative, recursive usage of modern Model-Driven Architecture (MDA) techniques. Several meta-models support each phase, thoroughly explained in [19]. From a language perspective, which is the preferred view in this thesis, Hili proposes different grammar concepts to support concurrent co-design, i.e., hardware and software, of embedded systems.

The system requirement analysis consists in the specification of several use cases of the system, or *scenarios*. As a descriptive example of the methodology, we defined a color converter system with a single use case shown in Figure 9.9. The "User" is intended to model, through message exchanges (and its respective state machine), the interactions with the system, *it is an $\langle$HOE$\rangle^2$ objet by itself*. In other words, it might be seen as the root object of our $\langle$HOE$\rangle^2$ application, which is equivalent to the main entry in widely known languages like C/C++ or Java. Given

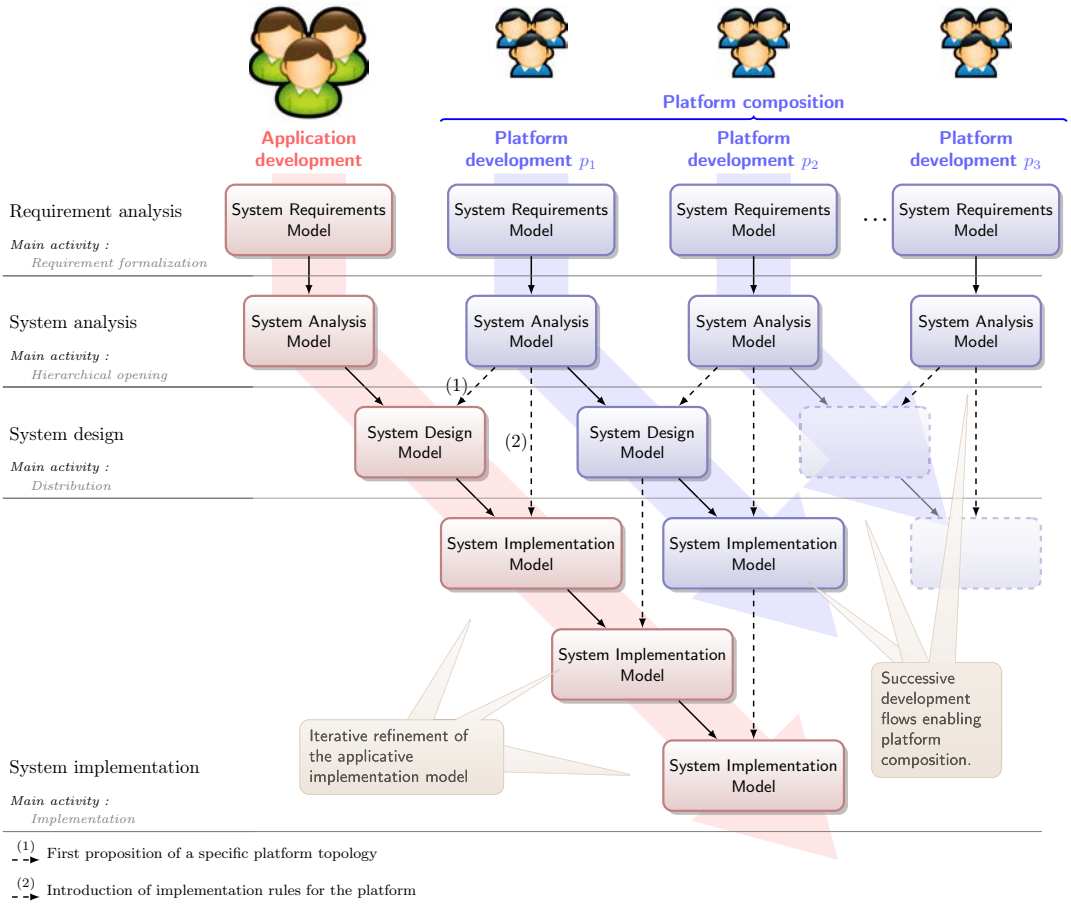


FIGURE 9.8: The $\langle \text{HOE} \rangle^2$ methodology for the development of embedded systems

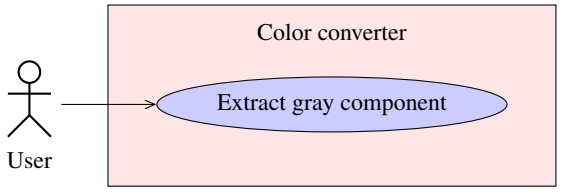


FIGURE 9.9: System requirements: Color converter use case

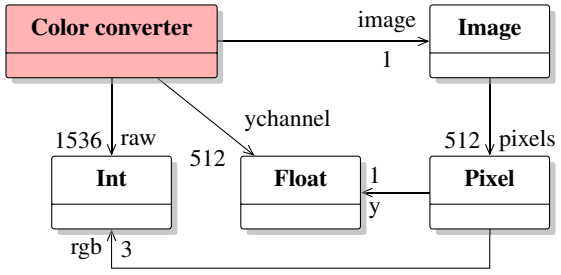
that User and System are $\langle \text{HOE} \rangle^2$ objects, they form more generally a *closed system* until future system releases. We concretized the system requirement view in Listing 9.10.

The system analysis phase involves a *hierarchical opening of the system*. That is, after defining a set of convenient use cases, the system designer concentrates on the required objects to meet such requirements. We show in Figure 9.10(a) a particular hierarchical opening of the system.

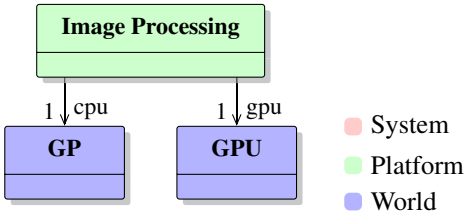
In the next phase, we proceed to the system design where its main activity concerns *object distribution*. The application is distributed over a predefined platform model. In order to show the main idea, we propose a simplistic model of a GPGPU platform depicted in Figure 9.10(b), and its equivalent source code view shown at Listing 9.11. We have two new meta-model concepts

```
import Converter

object User
  has Converter as system

  sm UserSM
    /* Main application entry: Create a "default" system */
    creator main() / system = new Converter.default() to BootSystem

    /* Use case "Extract gray component" */
    state ExtractGrayComponent. on / : system.convert()
                                to ExtractingGrayComponent

    state ExtractingGrayComponent. endon converted() /
```

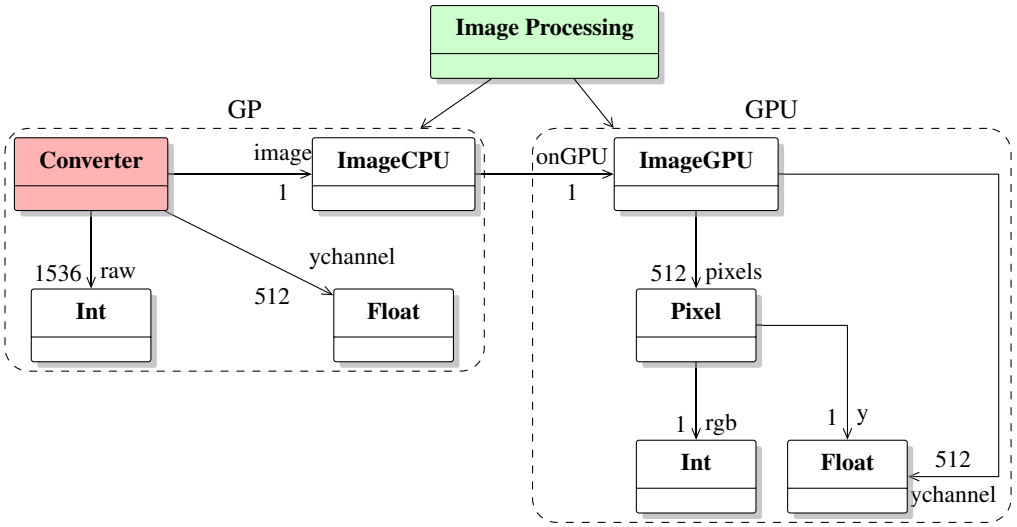
LISTING 9.10: Color converter source code



(a) Converter system analysis: Hierarchical Opening



(b) Platform model: GPGPU case



(c) System design: Object distribution

FIGURE 9.10: ⟨HOE⟩^{2} methodology: Distribution

called **platform** and **world**. A platform may contain multiple worlds, which are capable of hosting any applicative object. A particular distribution is given in Figure 9.10(c). We split object `Image` into two distinct objects, `ImageCPU` and `ImageGPU`, related by a new association, `onGPU`, in such a way that users and uses of `Image` may keep the same behavior as before. Clearly, the split object need to be rewritten according to the new implementation after distribution. Currently, the distribution and object splitting are ad-hoc tasks, guided by a good knowledge of the

```
world GP
world GPU

platform ImageProcessing
  has GP  as gp
  has GPU as gpu
```

LISTING 9.11: Platform description

underlying platform and the application to be distributed.

The implementation phase concerns platform-specific model transformations to ensure efficient code generation. From the language point of view of models, model transformations stands for $\langle \text{HOE} \rangle^2$ source code transformations. The optimizing compiler proposed in this thesis tries to accomplish such work automatically at the IR level, or at the very least provide a set of well-founded transformations to enable efficient code generation.

9.5 Towards GPGPU Code Generation

The distribution phase introduces a new interaction between objects of differents worlds. The state machine rewriting is guided by the $\langle \text{HOE} \rangle^2$ method. In our particular case, the object at the CPU side will send a `run()` request to the GPU object, which will then answer with a `completed(Float[512])` message carrying the computed result. Figure 9.11 describes both scenarios, before and after the rewriting, and Listing 9.12 shows the state machine of `ImageGPU` where we highlighted the increment with respect to `Image` object. As we can see, all the work has been transfered to the GPU and its CPU counterpart is there to preserve the interface among its users. Currently, this transformation is not performed automatically and needs user intervention on the model.

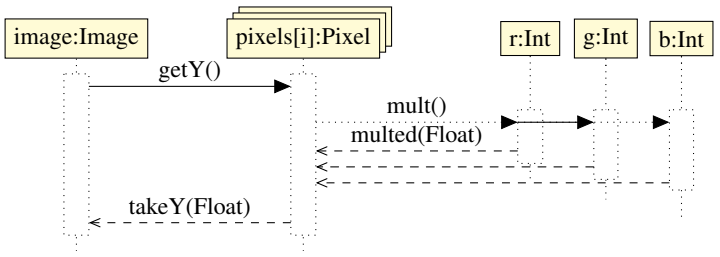
The GPU world is more constrained than the CPU one, hence we need to follow certain platform constraints.

Platform constraints

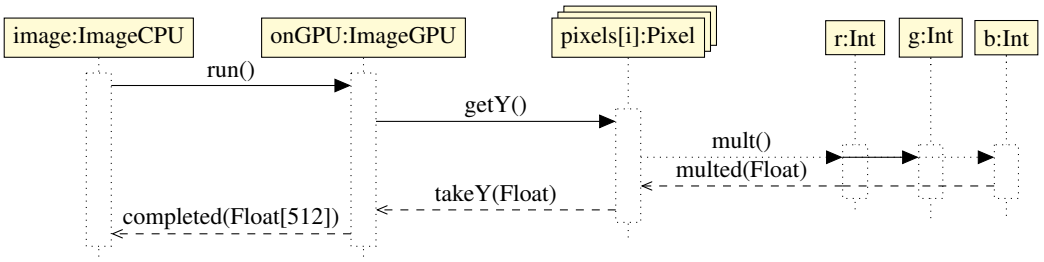
(1) We cannot follow pointers inside the structure

(2) No support for a $\langle \text{HOE} \rangle^2$ runtime implementation

Constraint *(1)* implies a code generation of unboxed generic objects. Similar to scalar unboxing, unboxed $\langle \text{HOE} \rangle^2$ objects cannot communicate. This leads us to the second constraint because no runtime implementation implies no communications and no object creations inside `ImageGPU`. Hopefully, the current set of compiler transformations allows us to automatically fulfill such



(a) Before distribution



(b) After distribution

FIGURE 9.11: New interaction after distribution

```
object ImageGPU
  interface
    on  run() -> completed(Float[512])

  has [512] Pixel as pixels
  has [512] Float as ychannel

  sm ImageSM.
    creator RGB(rgb: Int[1536]) /
        { i: 0..pixels.len - 1 }
        pixels[i] = new Pixel.RGB(rgb[3*i..3*i+2])
      to GETY

    state GETY.
      on  run() / : { i: 0..pixels.len - 1 } pixels[i].getY()
      to GETTING_Y

    state GETTING_Y.
      on takeY{i}(y: Float) / ychannel[i] = y to GETTING_Y
      endon [i.all] / : initiator.completed(ychannel)
```

LISTING 9.12: ImageGPU object model after split of Image object

requirements. Using the most agressive optimization level, we compile `ImageGPU` such that all arithmetic operations are folded and `Pixel` objects are inlined into `ImageGPU`, yielding no communications at all (as shown by our previous results). However, compiling arbritrary IR patterns into GPU kernels might be an overly complex task. Thus, we define a target set of IR patterns to reach via compiler transformations before entering the OpenCL backend.

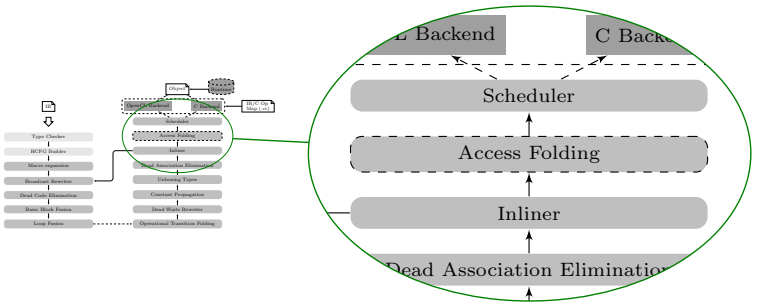


FIGURE 9.12: OpenCL-specific extension of the optimization pipeline

OpenCL Backend: Input constraints

(*a*) The root GPU object must contain an indexed region that perform the data-intensive work

(*b*) The indexed region must be enclosed by a well-defined transaction

(*c*) The transaction must be exposed at the interface

Given that the optimization phases favor indexed regions over broadcasts and in-place operations over communications, (*a*) is achieved automatically by the compiler. Constraints (*b*) and (*c*) are handled at the distribution activity.

Having fulfilled all the backend requirements, the OpenCL backend generates a kernel that performs the computation modeled at `ImageGPU` and, according to (*2*), *it does not communicate*. Therefore, objects hosted by the CPU world cannot communicate with the GPU ones anymore. As a consequence, the scenario shown at Figure 9.11(b) is to be transformed.

9.5.1 Extending the Optimizing Chain

We call objects like `ImageCPU` *OpenCL master* objects. Instead of communicating, the OpenCL master need to access directly the corresponding association(s) returned by the GPU root object. Based on constraint (*c*) and the message dependency analyzer presented in Section 8.3.2, we build a new transformation pass that replaces the send/receive relation by an in-place access.[3] For instance, `ImageGPU` in our example (see Listing 9.12) stores the computation result in association `ychannel` and sends its content to `ImageCPU` via message `completed(Float[512])`. After the transformation, the latter will access directly association `ychannel` of the former. We call this pass the *Access Folding* transformation. Figure 9.12 show the integration on the generic optimizing chain.

[3] So far, we only replaced send/receive dependencies by in-place operations.

9.5.2 OpenCL Code Generation

We show the generated host and OpenCL kernel code in Listing 9.13. In this implementation, we can see how the `ImageGPU` creator at Line 32 sets up the OpenCL kernel, run it and read its results from device memory. It is worth to note the interface between $\langle \text{HOE} \rangle^2$ objects and unboxed ones. Because `ImageGPU` is still created from an array of $\langle \text{HOE} \rangle^2$ objects, we found unboxing operations to set GPU associations (see Line 41).

The kernel corresponds to the indexed region introduced by the compiler transformations as required by *(a)* and it runs on the unboxed `ImageGPU` object (see Line 1). The OpenCL backend maintains the same unboxed type at both worlds, the kernel and the host one (though not shown here), respectively. Note that the creator returns the host-side structure such that `ImageCPU` can access it safely thanks to the Access Folding transformation introduced in the precedent section.

9.6 Conclusions

Under a custom runtime environment for communicating state machines, we stressed the proposed optimizing chain. We started by testing it on a simple object model where we showed encouraging results in terms of generated code quality. We managed to produce efficient C for loops from very high-level models taking into account many different notions of the $\langle \text{HOE} \rangle^2$ language: scalars, indexed regions, indexed messages, interface specification and replies.

We stressed the flow at different optimization levels against a complex model that implements part of the JPEG algorithm. The most aggressive optimization level achieves interesting results in term of messages: they are all optimized out. However, the number of live objects in the application runtime remains relatively high. The compiler did not manage to remove some boxing operations, though 77% of objects are removed from O0 to O1 and 17% from O1 to O2 resulting in a total of 88% eliminated boxing operations. Indeed, the store of an unboxed object into an association requires a boxing operation. There is still work to be done in automatic unboxing of associations in order to obtain yet more efficient implementations.

Despite of the number of objects, we showed that the language expressivity and modularity allow the programmer to write very high-level and composable expressions while obtaining efficient implementations with deep loop nesting levels computing over hierarchical structures.

We continue our experimentations on the methodology foundations of the $\langle \text{HOE} \rangle^2$ language. We presented the $\langle \text{HOE} \rangle^2$ methodology, which have been thoroughly studied in [19], and based on it we show the flexibility of our optimizing compiler. The GPGPU code generation would not have been possible in an semi-automatic fashion without the analyses developed at Chapter 8 and, technically very important, the compiler infrastructure. On the other hand, the number of

```
1 char *iregion_4594355024_src = "struct Pixel {"
2 "       int r;"
3 "       int g;"
4 "       int b;"
5 "       float y;"
6 "       float rcst;"
7 "       float gcst;"
8 "       float bcst;"
9 "};"
10 "typedef struct Pixel Pixel;"
11 "struct ImageGPU {"
12 "       struct Pixel pixels[512];"
13 "       float ychannel[512];"
14 "};"
15 "typedef struct ImageGPU ImageGPU;"
16 "kernel void iregion_4594355024(global struct ImageGPU *this)"
17 "{"
18 "       int i = get_global_id(0);"
19 "       float __new_var_6 = 0.299;"
20 "       float __new_var_7 = 0.587;"
21 "       float __new_var_8 = 0.114;"
22 "       float __new_var_5 = this->pixels[i].r * __new_var_6;"
23 "       float __new_var_3 = this->pixels[i].g * __new_var_7;"
24 "       float __new_var_4 = this->pixels[i].b * __new_var_8;"
25 "       float __new_var_1 = __new_var_4 + __new_var_3;"
26 "       float __new_var_2 = __new_var_5 + __new_var_1;"
27 "       this->pixels[i].y = __new_var_2;"
28 "       this->ychannel[i] = this->pixels[i].y;"
29 "}"
30 ""
31 ;
32 struct ImageGPU * __obj_new_ImageGPU_RGB(struct Int * rgb[1536])
33 {
34         struct ImageGPU *this = (struct ImageGPU *)
35                                 malloc(sizeof(struct ImageGPU));
36         for (int i = 0; ((i >= 0) && (i < 512)); (i+=1)) {
37                 struct Int * inraw_slice[3] = { rgb[(3 * i)]
38                                               , rgb[((3 * i) + 1)]
39                                               , rgb[((3 * i) + 2)]
40                                               };
41                 this->pixels[i].r = inraw_slice[0]->value;
42                 this->pixels[i].g = inraw_slice[1]->value;
43                 this->pixels[i].b = inraw_slice[2]->value;
44         }
45         hoe2cl_platform *plt = hoe2cl_platform_new();
46         hoe2cl_kernel *kernel = hoe2cl_kernel_new_from_source(
47                                   iregion_4594355024_src,
48                                   "iregion_4594355024",
49                                   plt
50                                 );
51         hoe2cl_kernel_write( kernel, &kernel->input
52                            , this, sizeof(struct ImageGPU));
53         hoe2cl_kernel_run(kernel, 512);
54         hoe2cl_kernel_read( kernel, &kernel->input
55                           , this, sizeof(struct ImageGPU));
56         return this;
57 }
```

LISTING 9.13: GPGPU generated code

constraints introduced to get GPGPU code generation may wrongly seem to lack of genericity or, in other words, driven by the specific example. It remains to develop more interesting model examples to run on GPGPU and show that the approach applies broadly.

Chapter 10

Conclusions and Perspectives

Contents

In the domain of Model-Driven Engineering (MDE), we studied in this thesis a new path for the modeling of parallel computations based on communicating Statecharts. The path led us to specific parallel extensions of the Statecharts formalism well-suited for data-intensive applications on recent parallel architectures.

10.1 Main Results and Contributions

This thesis is embedded into model-driven approaches for the design and development of Embedded System (ES). On this context, we highlighted interesting works pointing the need of new methodologies and supporting languages to exploit the parallelism offered in recent parallel architectures.

Object Modeling & Statecharts. Our first contribution addressed data parallel issues inside the Statechart itself, a widely known formalism to model behavior in the MDE community. In order to model parallel operations, we propose an original view of scalars as communicating state machines and its arithmetics operations being modeled by means of message passing semantics. Also, the parallelism is directly exposed in the model structure through multi-valued associations, over which parallel messages sending — thus denoting arithmetic operations —

is possible. We combined this idea with interesting Statechart extensions and parallel constructions in the action language: *indexed messages* and *indexed regions*, *parallel updates* and *parallel sending actions*.

The message-based arithmetics exposes a different view of scalars, making them functionally equivalent to generic objects. From the MDE point of view, the modeling activity becomes homogeneous, i.e., there is no need for different languages in the modeling framework to express control and data driven operations. This set of features allowed us to address data-intensive applications (JPEG) in a Statechart-based modeling style.

Statechart Formal Semantics. The second contribution is the specification of a formal semantics for Statecharts following a hierarchical and modular structure. This formalization takes into account the action language, the Statechart structure and communications between them in an object-oriented setting. We avoid semantical complications due to non-constructive semantics of Statecharts while still supporting asynchronous communications and composition of state machines. If not exhaustive, we showed how the formalization approach allows us to separate different semantical points in order to handle them layer by layer. Thus reducing the complexity of a flattened formalization as we have seen in existing approaches.

Intermediate Representation. The next contribution concerns the compilation of Statecharts. We proposed a compilation chain based on an expressive intermediate representation. This representation preserves most of the information coming from the front-end language, notably the state machine hierarchy. We support dynamic Statecharts, asynchronous communications, arrays, struct-like accesses and indexed domains, among other features. From this representation, we can reason about the hierarchy and the asynchrounous communications between different objects. Given the nature of index domains, we enabled powerful polyhedral analyses for automatic parallelization and code generation.

Static Analyses and Transformations. On the context communicating Statecharts, we extended the static analysis of reaching definitions to take into account the nesting structure of regions in the state machine. In the set of static analyses, we also proposed two complementary passes, *Message Dependency Analyzer* and *Transaction Selector*, that find send/receive relations enabling the folding of transitions and the inlining of objects.

In the Transaction Selector analysis, we introduced the notion of concurrent dependencies called π dependency. A π dependency exposes concurrent data-dependencies introduced by send and receive relations corresponding to a transaction between two objects. It allowed us to safely propose a deterministic behavior for code generation.

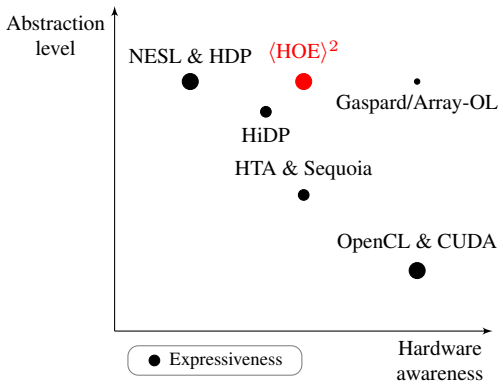


FIGURE 10.1: The ⟨HOE⟩² approach

Compiler for Embedded Architectures. The generation of efficient code from high-level models is a challenging task. We showed results concerning the quality of the generated code from a simple example and we extended it to cover a part of the JPEG algorithm. It resulted in complex generated code showing how powerful our model abstractions are to handle hierarchical implementations in case of data-intensive applications. The produced code is competitive with respect to hand-coded implementations. Moreover, we proved the flexibility of our compilation flow by adpating it to target parallel embedded architectures (General-Purpose computing on Graphics Processing Units (GPGPU) platforms).

10.2 A Look Back to the Thesis Motivations

This work started on the basis of multiple challenges concerning parallel programming of many-core embedded architectures. We observed that modern parallel languages and its corresponding compilers cannot follow complex constraints of embedded platforms anymore. We certainly need abstractions to handle this complexity. The iterative integration of platform constraints into a given application makes of MDE techniques an attractive approach to consider them in an abstract manner.

We addressed parallelism, formal semantics and code generation issues of Statecharts as the necessary and non-trivial steps towards a well-founded Statechart-based modeling language. As shown by our results, the parallel action language we proposed allowed us to generate efficient code and target parallel architectures such as GPGPU platforms.

However, the compiling challenge of taking into account complex platform constraints in an MDE fashion is still open. Platform-specific operators and its corresponding types, cache sizes, scheduling and mapping constraints are some of the performance factors that we should integrate into the optimizer via new model abstractions.

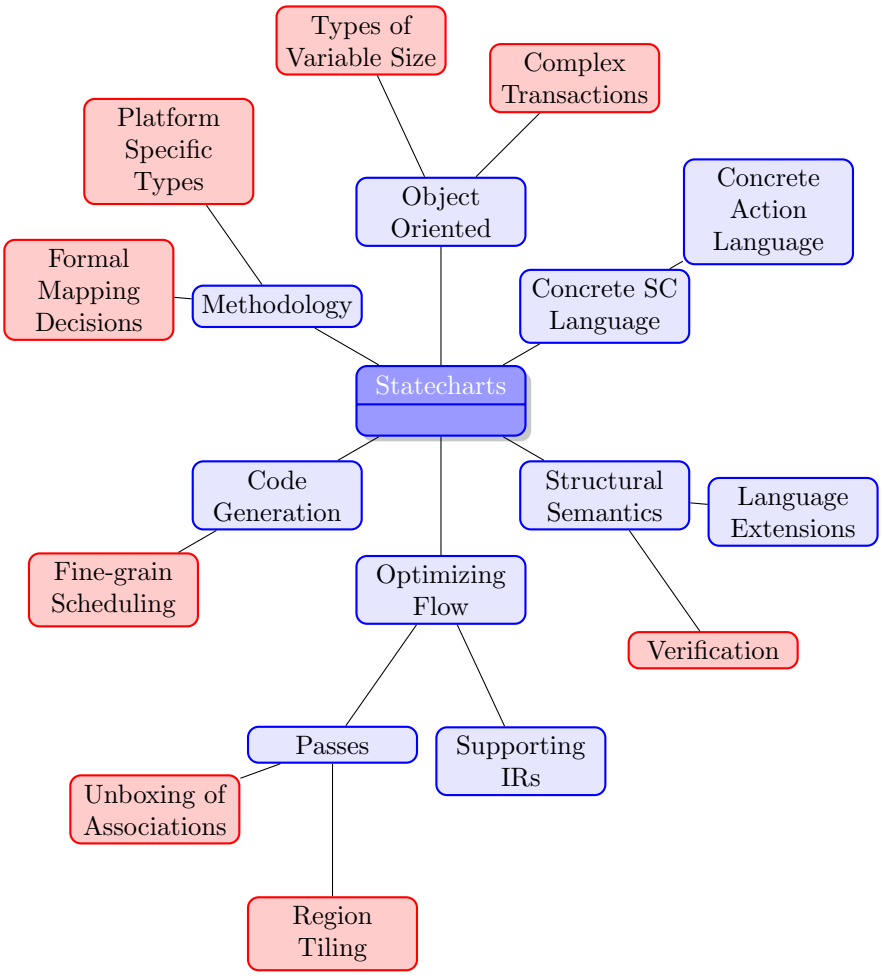


FIGURE 10.2: Future directions and extensions (in red)

In conclusion, we proposed a modern view of object-oriented Statecharts adapted for parallel programming. Figure 10.1 shows a comparison of our approach to reviewed parallel languages of Chapter 2. We remark the lack of formal platform constraints on the modeling approach we propose. However, it offers interesting abstractions in an unified setting and, most importantly, shows higher expressivity than other languages (dynamic Statecharts, array-like accesses, index domains).

As we will show in the following section, we distinguish several research directions gravitating around Statecharts that may guide new approaches to handle platform specific optimizations from a MDE perspective.

10.3 Future Research Directions

In this thesis, we lay the ground of a modeling language entirely based on Statecharts, even for arithmetic operations, with parallel support for data and control driven actions. However, there

is still a lot of work to be done in this direction.

From methodology to final code generation issues, we highlighted in Figure 10.2 several interesting points to develop further concerning different Statechart aspects.

Methodology. We place at the methodology process, the formalization of multiple hardware specific constraints. As shown in Chapter 9, the current state of the Highly Heterogeneous, Object-Oriented, Efficient Engineering $\langle \text{HOE} \rangle^2$ methodology allows us to take user-driven decisions concerning the distribution of our application. However, we need specific metrics to be able to reason about different application distribution strategies. Platform specific types can be abstracted using models. The compiler optimizations should be able to feedback the designer to spot relevant uses of platform-specific types. For instance, it may perform automatic indexed region tiling to find vectorization opportunities if the platform supports vector types and its respective operations.

Object-Oriented. The object oriented support of $\langle \text{HOE} \rangle^2$ leverages modular and composable designs. Modularity is the key for strong optimizations such as Statechart inlining. During our work, we proposed an external interface of objects that exposes message interactions between the object and its owners — closely related to initiator semantics. The Message Dependency Analyzer pass currently supports simple transaction entries. In order to capture send/receive relations with complex reply expressions, the analysis must be extended. Concerning association kinds, the compiler handles fixed size multiplicities. An extension to variable sized multiplicities of associations is necessary to widen the spectrum of supported input models. Moreover, we believe that types can be *refined*, i.e., from variable sized or unbounded to fixed or bounded ones, through specific static analyses.

Semantics. A natural future direction of the formal semantics concerns *verification*. Indeed, existing semantics of Statecharts are followed by formal verification methods, frequently based on *model checking*.

Transformation Passes. As shown in Chapter 9, the automatic type unboxing is the key to avoid unnecessary $\langle \text{HOE} \rangle^2$ object creation. Currently, the type unboxer do not unbox association types, which is the root of many boxing operations and hence the large number of live objects.

Code Generation. The current code generation strategy of parallel and indexed regions is based on a lexicographic scheduling through a classical polyhedral code generation algorithm. We make sure that no inter-iteration and no inter-region dependencies exists in order to safely

generate the scanning code. Cross regions synchronization dependencies should be taken into account to feed the polyhedral model and produce safe schedules.

Every single branch of Figure 10.2 opens a great number of questions and lead to specific research works. Several branches may be also related at some extent. For instance, certain optimizations passes could discover new opportunities and feedback methodological related actions. The set of optimizations are necessary driven by a specific semantics of the object-oriented Statecharts.

Glossary

$\langle$**HOE**$\rangle^2$ Highly Heterogeneous, Object-Oriented, Efficient Engineering.

Array-OL Array Oriented Language.

CFG Control-Flow Graph.

DCT Discrete Cosine Transform.
DSL Domain-Specific Language.

EMF Eclipse Metamodeling Framework.
ES Embedded System.
ETL Epsilon Transformation Language.

FIFO First In-First Out.
fUML foundational UML.

GEMM General Matrix Multiplication.
GPGPU General-Purpose computing on Graphics Processing Units.
GPU Graphical Processing Unit.

HCFG Hierarchical Control-Flow Graph.
HDP Haskell Data-Parallel.
HiDP Hierarchical Data Parallel Language.
HSM Hierarchical State Machine.
HTA Hierarchically Tiled Arrays.

IR Intermediate Representation.
ISL Integer Set Library.

LTS Labeled Transition System.

M2M Model-to-Model.
MBD Model Based Design.
MDA Model-Driven Architecture.
MDD Model-Driven Development.

MDE Model-Driven Engineering.
MoC Model of Computation.

NDP Nested Data Parallel.
NESL A nested data-parallel programming language.

OMG Object Management Group.
OOP Object-Oriented Programming.

PDM Platform-Dependent Model.
PIM Platform-Indepedent Model.
PMH Parallel Memory Hierarchy.
PSM Platform-Specific Model.

QVT Query/View/Transformation.

RLE Run Lenght Encoding.
RTC Run-To-Completion.

SoC System on Chip.
SSA Single Static Assignment.
STT State Table Transition.

UML Unified Modeling Language.

Appendix A

Syntax

We present in the following sections the syntax of the $\langle$HOE$\rangle^2$ language and its Intermediate Representation (IR). We use traditional grammar operators to denote zero or many elements $*$, one or many $+$ and optional elements $[]$.

A.1 $\langle$HOE$\rangle^2$

⟨*object*⟩ ::= '**object**' ⟨*id*⟩ ⟨*interface*⟩ ⟨*associations*⟩ ⟨*sm*⟩

⟨*sm*⟩ ::= '**sm**' ⟨*id*⟩ '**.**' ⟨*creator*⟩+ ⟨*state*⟩+

⟨*creator*⟩ ::= '**creator**' ⟨*id*⟩ '**(**' ⟨*param*⟩* '**)**' ['**/**' ⟨*update*⟩+] '**to**' ⟨*id*⟩

⟨*state*⟩ ::= ⟨*sstate*⟩ | ⟨*cstate*⟩

⟨*sstate*⟩ ::= '**state**' ⟨*id*⟩ '**.**' ⟨*trn*⟩+

⟨*trn*⟩ ::= ⟨*external*⟩ | ⟨*final*⟩

⟨*external*⟩ ::= '**on**' ⟨*trigger*⟩* ['**[**' ⟨*guard*⟩ '**]**'] '**/**' [⟨*action*⟩] '**to**' ⟨*id*⟩

⟨*final*⟩ ::= '**endon**' ⟨*trigger*⟩* ['**[**' ⟨*guard*⟩ '**]**'] ['**/**' ⟨*action*⟩]

⟨*cstate*⟩ ::= '**cstate**' ⟨*id*⟩ '**.**' ⟨*region*⟩+ ⟨*trn*⟩+

⟨*region*⟩ ::= '**region**' ['{' ⟨*indexset*⟩ '}'] ⟨*initial*⟩ ⟨*state*⟩+
'**endregion**'

⟨*initial*⟩ ::= '**initial**' ⟨*id*⟩

⟨*action*⟩ ::= ⟨*update*⟩* ['**:**' ⟨*send*⟩+]

⟨*update*⟩ ::= ⟨*supdate*⟩ | ⟨*iupdate*⟩

⟨*iupdate*⟩ ::= '{' ⟨*indexset*⟩ '}' ⟨*supdate*⟩

⟨*supdate*⟩ ::= ⟨*ulhs*⟩ '**=**' ⟨*urhs*⟩

⟨*ulhs*⟩ ::= ⟨*var*⟩
| ⟨*vardef*⟩

⟨*urhs*⟩ ::= ⟨*var*⟩
| ⟨*new*⟩
| ⟨*applyon*⟩

⟨*send*⟩ ::= ⟨*ssend*⟩ | ⟨*isend*⟩

⟨*isend*⟩ ::= '{' ⟨*indexset*⟩ '}' ⟨*ssend*⟩

⟨*trigger*⟩ ::= ⟨*id*⟩ ['{' ⟨*id*⟩+ '}'] '**(**' ⟨*param*⟩ '**)**'

⟨*new*⟩ ::= '**new**' ⟨*id*⟩ '**.**' ⟨*id*⟩ '**(**' ⟨*var*⟩* '**)**'

⟨*applyon*⟩ ::= '**applyon**' ⟨*id*⟩'**(**' ⟨*var*⟩* '**)**'

⟨*guard*⟩ ::= ⟨*guard*⟩ ['**and**', '**or**'] ⟨*guard*⟩
| '**not**' ⟨*guard*⟩
| ⟨*arithexpr*⟩ ['**<**', '**<=**', '**>**', '**>=**', '**=**'] ⟨*arithexpr*⟩

⟨*arithexpr*⟩ ::= ⟨*arithexpr*⟩ ['**+**', '**-**', '*****', '**/**', '**%**'] ⟨*arithexpr*⟩
| ⟨*var*⟩
| ⟨*int*⟩

⟨*indexset*⟩ ::= ⟨*var*⟩+ '**:**' ⟨*guard*⟩

⟨*var*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩ '**[**' ⟨*arithexpr*⟩+ '**]**'

⟨*type*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩ '**[**' ⟨*range*⟩+ '**]**'

⟨*range*⟩ ::= ⟨*int*⟩
| ⟨*int*⟩ '**..**' ⟨*int*⟩
| ⟨*int*⟩ '**..**' '*****'
| '*****'

A.2 Intermediate Representation

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

⟨*fsm*⟩ := '**fsm**' ⟨*id*⟩ '**.**' ⟨*id*⟩ '**(**' ⟨*var_decl*⟩ '**)**' '**{**' ⟨*labeled_fsmstmt*⟩+ '**}**'

⟨*object*⟩ := '**object**' ⟨*id*⟩ '**{**' ⟨*object_expr*⟩+ '**}**'

⟨*scalar*⟩ := '**scalar**' ⟨*id*⟩ '**{**' ⟨*scalar_expr*⟩+ '**}**'

⟨*object_expr*⟩ := ⟨*association_blk*⟩ | ⟨*interface_blk*⟩

⟨*association_blk*⟩ := '**associations**' '**{**' ⟨*association_decl*⟩+ '**}**'

⟨*interface_blk*⟩ := '**interface**' '**{**' ⟨*interface_expr*⟩+ '**}**'

⟨*interface_expr*⟩ := ⟨*transaction*⟩ | ⟨*op_transaction*⟩ | ⟨*msg_decl*⟩

⟨*transaction*⟩ := ⟨*id*⟩ '**=**' ⟨*msg_decl*⟩ '**->**' ⟨*msg_decl*⟩+
| ⟨*msg_decl*⟩ '**->**' ⟨*msg_decl*⟩+

⟨*op_transaction*⟩ := ⟨*msg_decl*⟩ '**->**' ⟨*msg_decl*⟩+ '**~>**' ⟨*id*⟩

⟨*msg_decl*⟩ := ⟨*id*⟩ '**(**' ⟨*association_decl*⟩* '**)**'

⟨*association_decl*⟩ := ⟨*obj_type*⟩ ⟨*id*⟩+

⟨*obj_type*⟩ := ⟨*simple_type*⟩ | ⟨*array_type*⟩

⟨*simple_type*⟩ := ⟨*id*⟩

⟨*array_type*⟩ := ⟨*simple_type*⟩ '**|**' ⟨*type_mult*⟩+ '**|**'

⟨*type_mult*⟩ := ⟨*int*⟩
| '*****'
| ⟨*int*⟩ '**..**' ⟨*int*⟩
| ⟨*int*⟩ '**..**' '*****'

⟨*labeled_fsmstmt*⟩ := [⟨*label*⟩] ⟨*fsmstmt*⟩

⟨*fsmstmt*⟩ := ⟨*par_stmt*⟩
| ⟨*forall*⟩
| ⟨*var_decl*⟩
| ⟨*wait*⟩
| ⟨*wait_in*⟩
| ⟨*goto*⟩
| ⟨*done*⟩

⟨*forall*⟩ := '**forall**' '**[**' ⟨*index_set*⟩ '**]**' '**{**' ⟨*labeled_fsmstmt*⟩+ '**}**'

⟨*goto*⟩ := '**goto**' ⟨*id*⟩

⟨*done*⟩ := '**done**' ⟨*id*⟩

⟨*wait*⟩ := '**wait**' ⟨*var_expr*⟩ [⟨*waitfor_expr*⟩] ⟨*when_expr*⟩+

⟨*wait_in*⟩ := '**wait**' ⟨*var_expr*⟩ '**in**' ⟨*region*⟩+ [⟨*waitfor_expr*⟩] ⟨*when_expr*⟩+

⟨*waitfor_expr*⟩ := '**for**' ⟨*recv_expr*⟩+

⟨*region*⟩ := ⟨*sregion*⟩ | ⟨*iregion*⟩

⟨*sregion*⟩ := '**{**' ⟨*fsmstmt*⟩+ '**}**'

⟨*iregion*⟩ := '**[**' ⟨*index_set*⟩ '**]**' ⟨*sregion*⟩

⟨*when_expr*⟩ := '**then**' '**when**' ⟨*when_cond*⟩+ '**if**' ⟨*if_condition*⟩ ⟨*goto*⟩
| '**then**' '**when**' ⟨*when_cond*⟩+ ⟨*goto*⟩
| '**then**' '**if**' ⟨*if_condition*⟩ ⟨*goto*⟩
| '**then**' '**goto**'

⟨*when_cond*⟩ := ⟨*id*⟩

⟨*label*⟩ := ⟨*id*⟩ '**:**'

⟨*creator*⟩ := '**creator**' ⟨*id*⟩ '**.**' ⟨*id*⟩ '**(**' ⟨*param_def*⟩* '**)**' '**{**' ⟨*creator_stmt*⟩+ '**}**'

⟨*creator_stmt*⟩ := ⟨*par_stmt*⟩
| ⟨*forall_creator_stmt*⟩
| ⟨*var_decl*⟩
| ⟨*start_expr*⟩

⟨*forall_creator_stmt*⟩ := '**forall**' '**[**' ⟨*index_set*⟩ '**]**' '**{**' ⟨*creator_stmt*⟩+ '**}**'

⟨*start_expr*⟩ := '**start**' ⟨*id*⟩ '**of**' ⟨*var_expr*⟩

⟨*if_condition*⟩ := ⟨*if_condition*⟩ '**and**' ⟨*if_condition*⟩
| ⟨*if_condition*⟩ '**or**' ⟨*if_condition*⟩
| '**(**' ⟨*if_condition*⟩ '**)**'
| ⟨*cmp_expr*⟩

⟨*index_set*⟩ := ⟨*index_def*⟩+ ['**:**' ⟨*index_set_constraints*⟩]

⟨*index_def*⟩ := ⟨*id*⟩

⟨*index_set_constraints*⟩ := ⟨*index_set_constraints*⟩ '**and**' ⟨*index_set_constraints*⟩
| ⟨*index_set_constraints*⟩ '**or**' ⟨*index_set_constraints*⟩
| '**(**' ⟨*index_set_constraints*⟩ '**)**'
| ⟨*cmp_expr*⟩

⟨*cmp_expr*⟩ := ⟨*arith_expr*⟩ '**=**' ⟨*arith_expr*⟩
| ⟨*arith_expr*⟩ '**<**' ⟨*arith_expr*⟩
| ⟨*arith_expr*⟩ '**<=**' ⟨*arith_expr*⟩
| ⟨*arith_expr*⟩ '**>**' ⟨*arith_expr*⟩
| ⟨*arith_expr*⟩ '**>=**' ⟨*arith_expr*⟩
| ⟨*arith_expr*⟩

⟨*arith_expr*⟩ := ⟨*arith_expr*⟩ '**+**' ⟨*arith_expr*⟩
| ⟨*arith_expr*⟩ '**-**' ⟨*arith_expr*⟩
| ⟨*arith_expr*⟩ '**/**' ⟨*arith_expr*⟩
| ⟨*arith_expr*⟩ '*****' ⟨*arith_expr*⟩
| ⟨*arith_expr*⟩ '**%**' ⟨*arith_expr*⟩
| '**(**' ⟨*arith_expr*⟩ '**)**'
| ⟨*int*⟩
| ⟨*var_expr*⟩

⟨*par_stmt*⟩ := ⟨*par_expr*⟩ '**,**' ⟨*par_expr*⟩ | ⟨*par_expr*⟩

⟨*par_expr*⟩ := ⟨*send_expr*⟩ | ⟨*update_expr*⟩

⟨*update_expr*⟩ := ⟨*update*⟩ | ⟨*forall_update*⟩

⟨*update*⟩ := ⟨*var*⟩ '**=**' ⟨*var_expr*⟩
| ⟨*var*⟩ '**=**' ⟨*create_expr*⟩
| ⟨*var*⟩ '**=**' ⟨*new_expr*⟩
| ⟨*var*⟩ '**=**' ⟨*initializer_expr*⟩
| ⟨*var*⟩ '**=**' ⟨*applyon_expr*⟩

⟨*forall_update*⟩ := '**forall**' '**[**' ⟨*index_set*⟩ '**]**' ⟨*update*⟩

⟨*var*⟩ := ⟨*var_expr*⟩ | ⟨*var_decl*⟩

⟨*applyon_expr*⟩ := '**applyon**' ⟨*var_expr*⟩ ⟨*applyon_type_expr*⟩ ⟨*param*⟩*

⟨*initializer_expr*⟩ := '**{**' ⟨*param*⟩+ '**}**'

⟨*create_expr*⟩ := '**create**' ⟨*id*⟩ '**.**' ⟨*id*⟩ ⟨*param*⟩*

⟨*var_decl*⟩ := ⟨*obj_type*⟩ ⟨*var_expr_leaf*⟩ | ⟨*msg_type*⟩ ⟨*var_expr_leaf*⟩

⟨*var_expr*⟩ := var_expr '**.**' var_expr_leaf | var_expr_leaf

⟨*var_expr_leaf*⟩ := ⟨*id*⟩ | ⟨*id*⟩ '**[**' ⟨*index_var_expr*⟩ '**]**'

⟨*index_var_expr*⟩ := ⟨*arith_expr*⟩+ | ⟨*arith_expr*⟩+ '**:**' ⟨*index_set_constraints*⟩

⟨*new_expr*⟩ := '**new**' ⟨*obj_type*⟩

⟨*send_expr*⟩ := ⟨*send*⟩ | ⟨*forall_send*⟩

⟨*send*⟩ := ⟨*single_send_from_expr*⟩ | ⟨*reply_expr*⟩

⟨*forall_send*⟩ := '**forall**' '**[**' ⟨*index_set*⟩ '**]**' ⟨*send*⟩

⟨*single_send_from_expr*⟩ := '**sendfrom**' ⟨*var_expr*⟩ ⟨*var_expr*⟩ ⟨*msg_type*⟩ ⟨*param*⟩*
| '**sendfrom**' '**[**' ⟨*arith_exprs*⟩ '**]**' ⟨*var_expr*⟩ ⟨*var_expr*⟩ ⟨*msg_type*⟩ ⟨*param*⟩*

⟨*reply_expr*⟩ := '**reply**' ⟨*var_expr*⟩ ⟨*var_expr*⟩ ⟨*msg_type*⟩ ⟨*param*⟩*

⟨*recv_expr*⟩ := ⟨*single_recv_expr*⟩ | ⟨*recv_from_expr*⟩

⟨*recv_from_expr*⟩ := '**(**' ⟨*id*⟩ '**,**' ⟨*var_decl*⟩ '**)**' '**=**' '**recv**' ⟨*msg_type*⟩
| '**(**' ⟨*id*⟩ '**,**' ⟨*var_decl*⟩ '**)**' '**=**' '**recv**' '**[**' ⟨*index_set*⟩ '**]**' ⟨*msg_type*⟩

⟨*msg_type*⟩ := ⟨*simple_msg_type*⟩
| ⟨*qualified_msg_type*⟩
| ⟨*array_msg_type*⟩

⟨*simple_msg_type*⟩ := ⟨*id*⟩ '**<**' ⟨*obj_type*⟩* '**>**'

⟨*qualified_msg_type*⟩ := ⟨*id*⟩ '**'**' ⟨*id*⟩ '**<**' ⟨*obj_type*⟩* '**>**'

⟨*applyon_type_expr*⟩ := ⟨*id*⟩ '**'**' ⟨*id*⟩

⟨*param*⟩ := ⟨*int*⟩ | ⟨*float*⟩ | ⟨*var_expr*⟩

Bibliography

[1] Dirk Seifert. An Executable Formal Semantics for a UML State Machine Kernel Considering Complex Structured Data. Rapport de recherche, INRIA, 2008. URL http://hal.inria.fr/inria-00274391.

[2] Basilio Fraguela, Jia Guo, Ganesh Bikshandi, Maria Garzaran, Gheorghe Almasi, Jose Moreira, and David Padua. The Hierarchically Tiled Arrays Programming Approach. 2004. URL http://www.des.fi.udc.es/~basilio/papers/lcr04.pdf.

[3] 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. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188543. URL http://doi.acm.org/10.1145/1188455.1188543.

[4] Frank Mueller and Yongpeng Zhang. Hidp: A Hierarchical Data Parallel Language. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4673-5524-7. doi: 10.1109/CGO.2013.6494994. URL http://dx.doi.org/10.1109/CGO.2013.6494994.

[5] Pierre Boulet. Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Technical report, INRIA, 2007. URL http://hal.inria.fr/docs/00/12/92/23/PDF/RR-6113v2.pdf.

[6] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. SCCharts: Sequentially Constructive Statecharts for Safety-critical Applications: HW/SW-synthesis for a Conservative Extension of Synchronous Statecharts. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 372–383, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594310. URL http://doi.acm.org/10.1145/2594291.2594310.

[7] T. Schattkowsky and W. Muller. Transformation of UML State Machines for Direct Execution. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 117–124, Sept 2005. doi: 10.1109/VLHCC.2005.64.

[8] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. A Model-Driven Design Framework for Massively Parallel Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 10(4):39:1–39:36, November 2011. ISSN 1539-9087. doi: 10.1145/2043662.2043663.

[9] Manuel M. T. Chakravarty, Roman Lechtchinsky, and Simon Peyton Jones. Data Parallel Haskell: a Status Report. *DAMP*, 2007. URL `http://www.cse.unsw.edu.au/~chak/papers/data-parallel-haskell.pdf`.

[10] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73, May 2010. ISSN 0740-7475. doi: 10.1109/MCSE.2010.69. URL `http://dx.doi.org/10.1109/MCSE.2010.69`.

[11] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0131387685, 9780131387683. URL `http://dl.acm.org/citation.cfm?id=1891996`.

[12] A. Sangiovanni-Vincentelli. Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design. *Proceedings of the IEEE*, 95(3):467–506, 2007. doi: 10.1109/JPROC.2006.890107. URL `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4167779`.

[13] Thomas A. Henzinger and Joseph Sifakis. The Discipline of Embedded Systems Design. *Computer*, 40(10):32–40, October 2007. ISSN 0018-9162. doi: 10.1109/MC.2007.364. URL `http://dx.doi.org/10.1109/MC.2007.364`.

[14] Thomas A. Henzinger and Joseph Sifakis. The Embedded Systems Design Challenge. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-37215-8. doi: 10.1007/11813040_1.

[15] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A Model-Driven Design Environment for Embedded Systems. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 915–918, 2006. doi: 10.1109/DAC.2006.229412. URL `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1688928`.

[16] Chihhsiong Shih, Chien-Ting Wu, Cheng-Yao Lin, Pao-Ann Hsiung, Nien-Lin Hsueh, Chih-Hung Chang, Chorng-Shiuh Koong, and W.C. Chu. A Model-Driven Multicore Software Development Environment for Embedded System. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 2, pages 261–268, 2009. doi: 10.1109/COMPSAC.2009.148. URL `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5254115`.

[17] Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'H, and Jean-Luc Dekeyser. 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, March 2011. URL `https://hal.inria.fr/inria-00578646`.

[18] Nicolas Hili, Christian Fabre, Sophie Dupuy-Chessa, Dominique Rieu, and Ivan Llopard. Model-Based Platform Composition for Embedded System Design. In *2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs*, Aizu-Wakamatsu, Japan, September 2014. University of Aizu. URL `https://hal.inria.fr/hal-01071208`.

[19] Nicolas Hili. *Une méthode pour le développement collaboratif de systèmes embarqués*. PhD thesis, Ecole doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique , Grenoble Universités, Université de Grenoble, 2014. URL `http://www.hili.fr/thesis/output.pdf`.

[20] David Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/0167-6423(87)90035-9.

[21] OMG. Unified Modeling Language (UML) V2.4.1, 2011. URL `http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF`.

[22] Shuang Liu, Yang Liu, Étienne André, Christine Choppy, Jun Sun, Bimlesh Wadhwa, and Jin Song Dong. A Formal Semantics for Complete UML State Machines with Communications. In *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, pages 331–346, 2013. doi: 10.1007/978-3-642-38613-8_23. URL `http://dx.doi.org/10.1007/978-3-642-38613-8_23`.

[23] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. On formalizing UML state machines using ASMs. *Information and Software Technology*, 46(5):287 – 292, 2004. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2003.09.009. URL `http://www.sciencedirect.com/science/article/pii/S0950584903002027`. Special Issue on Software Engineering, Applications, Practices and Tools from the ACM Symposium on Applied Computing 2003.

[24] Michael von der Beeck. A Structured Operational Semantics for UML-Statecharts. *Software and Systems Modeling*, 1(2):130–141, 2002. ISSN 1619-1366. doi: 10.1007/s10270-002-0012-8. URL `http://dx.doi.org/10.1007/s10270-002-0012-8`.

[25] Ivan P Paltor. The Semantics of UML State Machines. Technical report, 1999.

[26] Michael von der Beeck. A Comparison of Statecharts Variants. In Hans Langmaack, Willem-Paul de Roever, and Jan Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer Berlin Heidelberg, 1994. ISBN 978-3-540-58468-1. doi: 10.1007/3-540-58468-4_163. URL `http://dx.doi.org/10.1007/3-540-58468-4_163`.

[27] A. Charfi, C. Mraidha, and P. Boulet. An Optimized Compilation of UML State Machines. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on*, pages 172–179, April 2012. doi: 10.1109/ISORC.2012.30.

[28] M. Alras, P. Caspi, A. Girault, and P. Raymond. Model-Based Design of Embedded Control Systems by Means of a Synchronous Intermediate Model. In *Embedded Software and Systems, 2009. ICESS '09. International Conference on*, pages 3–10, May 2009. doi: 10.1109/ICESS.2009.36.

[29] James C. Brodman, Basilio B. Fraguela, María J. Garzarán, and David Padua. New Abstractions for Data Parallel Programming. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 16–16, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855591.1855607.

[30] S. F. Hummel, S. Talla, and J. Brennan. The Refinement of High-level Parallel Algorithm Specifications. In *Proceedings of the Conference on Programming Models for Massively Parallel Computers*, PMMP '95, pages 106–, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7177-7. URL http://dl.acm.org/citation.cfm?id=525697.826735.

[31] Mike Delves. HPF. *Linux J.*, 1998(45es), January 1998. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=327171.327175.

[32] Guy E. Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). Technical report, Pittsburgh, PA, USA, 1993.

[33] Guy E. Blelloch. NESL: A Nested Data-Parallel Language. Technical report, Carnegie Mellon University, 1995. URL http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-95-170.html.

[34] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. ISBN 0262533022, 9780262533027.

[35] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, May 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.14.

[36] Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, 2009.

[37] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and a. Shtul-Trauring. Statemate: A Working Environment for the Development of Complex Reactive Systems. In *Proceedings of the 10th International Conference on Software Engineering*, ICSE '88, pages 396–406, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press. ISBN 0-89791-258-6. URL http://dl.acm.org/citation.cfm?id=55823.55861.

[38] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996. ISSN 1049-331X. doi: 10.1145/235321.235322. URL http://doi.acm.org/10.1145/235321.235322.

[39] David Harel and Hillel Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). In Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23135-6. doi: 10.1007/978-3-540-27863-4_19. URL http://dx.doi.org/10.1007/978-3-540-27863-4_19.

[40] IBM. Rational Rhapsody Family.

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

[42] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-983768-X.

[43] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-59414-6.

[44] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (FUML), 2013.

[45] Charles André. Computing SyncCharts Reactions. *Electron. Notes Theor. Comput. Sci.*, 88:3–19, October 2004. ISSN 1571-0661. doi: 10.1016/j.entcs.2003.05.007. URL `http://dx.doi.org/10.1016/j.entcs.2003.05.007`.

[46] Gérard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.*, 19(2):87–152, November 1992. ISSN 0167-6423. doi: 10.1016/0167-6423(92)90005-V. URL `http://dx.doi.org/10.1016/0167-6423(92)90005-V`.

[47] Reinhard Von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'brien, and Partha Roop. Sequentially Constructive Concurrency: A Conservative Extension of the Synchronous Model of Computation. *ACM Trans. Embed. Comput. Syst.*, 13(4s):144:1–144:26, July 2014. ISSN 1539-9087. doi: 10.1145/2627350. URL `http://doi.acm.org/10.1145/2627350`.

[48] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages 12 Years Later. *Proceedings of the IEEE*, 91(1):64–83, Jan 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2002.805826.

[49] Franck Chauvel and Jean-Marc Jézéquel. Code Generation from UML Models with Semantic Variation Points. In Lionel Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 54–68. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-29010-0. doi: 10.1007/11557432_5. URL `http://dx.doi.org/10.1007/11557432_5`.

[50] Eran Gery, David Harel, and Eldad Palachi. Rhapsody: A Complete Life-Cycle Model-Based Development System. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43703-1. doi: 10.1007/3-540-47884-1_1.

[51] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[52] Robert C Martin. UML Tutorial: Finite State Machines.

[53] P. Metz, J. O'Brien, and W. Weber. Code Generation Concepts for Statecharts Diagram of UML v1.1. Technical report, Object Technology Group, 1999.

[54] Eladio Dominguez, Beatriz Pérez, Angel L. Rubio, and Maria A. Zapata. A Systematic Review of Code Generation Proposals from State Machine Specifications. *Inf. Softw. Technol.*, 54(10): 1045–1066, October 2012. ISSN 0950-5849. doi: 10.1016/j.infsof.2012.04.008. URL `http://dx.doi.org/10.1016/j.infsof.2012.04.008`.

[55] Andreas Huber Dönni. The Boost Statechart Library. Technical report, Boost, April 2007. URL `http://www.boost.org/doc/libs/1_59_0/libs/statechart/doc/reference.pdf`.

[56] A. Charfi, C. Mraidha, S. Gerard, F. Terrier, and P. Boulet. Toward Optimized Code Generation Through Model-Based Optimization. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1313–1316, March 2010. doi: 10.1109/DATE.2010.5457010.

[57] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Sci. Comput. Program.*, 72(1-2):31–39, June 2008. ISSN 0167-6423. doi: 10.1016/j.scico.2007.08.002. URL `http://dx.doi.org/10.1016/j.scico.2007.08.002`.

[58] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The Epsilon Transformation Language. In *Theory and practice of model transformations*, pages 46–60. Springer Berlin Heidelberg, 2008.

[59] Radomil Dvorak. Model Transformation with Operational QVT, 2008.

[60] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885.

[61] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. 2013. ISBN 9781782160304. URL `http://www.packtpub.com/implementing-domain-specific-languages-with-xtext-and-xtend/book`.

[62] Jason Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proc. GCC Developers Summit, 2003*, pages 171–180, 2003.

[63] Philippe Dumont. *Spécification multidimensionelle pour le traitement du signal systématique*. PhD thesis, Université de Lille, 2005. URL `http://www.lifl.fr/west/publi/Dumo05phd.pdf`.

[64] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991. ISSN 0018-9219. doi: 10.1109/5.97300.

[65] Stephen A. Edwards. Tutorial: Compiling Concurrent Languages for Sequential Processors. *ACM Trans. Des. Autom. Electron. Syst.*, 8(2):141–187, April 2003. ISSN 1084-4309. doi: 10.1145/762488.762489. URL `http://doi.acm.org/10.1145/762488.762489`.

[66] Object Management Group. Concrete Syntax for a UML Action Language: Action Language for Foundational UML (ALF), 2013.

[67] Mojżesz Presburger and Dale Jabcquette. On the Completeness of a Certain System of Arithmetic of Whole Numbers in Which Addition Occurs as the Only Operation. *History and Philosophy of Logic*, 12(2):225–233, 1991. doi: 10.1080/01445340910883718 7. URL http://dx.doi.org/10.1080/01445340910883718 7.

[68] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.

[69] Dana Scott. Toward a Mathematical Semantics for Computer Languages. Technical Report PRG06, OUCL, August 1971.

[70] Amy Brown and Greg Wilson. *The Architecture Of Open Source Applications*, volume II. lulu.com, May 2012. ISBN 1257638017.

[71] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A Declarative Language for Real-time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM. ISBN 0-89791-215-2.

[72] Abdoulaye Gamatié. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 1441909400, 9781441909404.

[73] Gilles Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974.

[74] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. doi: 10.1109/IEEESTD.2008.4610935.

[75] Gordon D. Plotkin. A Structural Approach to Operational Semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[76] P. Dechering, L. Breebaart, F. Kuijlman, K. van Reeuwijk, and H. Sips. Semantics and Implementation of a Generalized Forall Statement for Parallel Languages. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 542–548, Apr 1997. doi: 10.1109/IPPS.1997.580953.

[77] Wuxu Peng and S. Puroshothaman. Data Flow Analysis of Communicating Finite State Machines. *ACM Trans. Program. Lang. Syst.*, 13(3):399–442, July 1991. ISSN 0164-0925. doi: 10.1145/117009.117015.

[78] Bruno Blanchet. Escape Analysis for Java™: Theory and Practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, November 2003. ISSN 0164-0925. doi: 10.1145/945885.945886. URL http://doi.acm.org/10.1145/945885.945886.

[79] Thomas Kotzmann and Hanspeter Mössenböck. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 111–120, New York, NY, USA, 2005. ACM. ISBN 1-59593-047-7. doi: 10.1145/1064979.1064996. URL http://doi.acm.org/10.1145/1064979.1064996.

[80] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[81] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400713.

[82] Sven Verdoolaege. ISCC Tutorial, September 2010. URL http://ocs.math.kobe-u.ac.jp/pukiwiki-1.4.7_notb_utf8/index.php?plugin=attach&refer=SoftwareTutorial&openfile=verdoolaege-iscc.pdf.

[83] K.B. Wheeler, R.C. Murphy, and D. Thain. Qthreads: An API for Programming with Millions of Lightweight Threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008. doi: 10.1109/IPDPS.2008.4536359.

[84] Cedric Bastoul. 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, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2229-7. doi: 10.1109/PACT.2004.11. URL http://dx.doi.org/10.1109/PACT.2004.11.

[85] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.*, 37(4):12:1–12:50, July 2015. ISSN 0164-0925. doi: 10.1145/2743016. URL http://doi.acm.org/10.1145/2743016.

[86] Christoph Loeffler, Adriaan Ligtenberg, and George S. Moschytz. Practical Fast 1-D DCT Algorithms with 11 Multiplications. In *IEEE*, 1989. URL http://www3.matapp.unimib.it/corsi-2007-2008/matematica/istituzioni-di-analisi-numerica/jpeg/papers/11-multiplications.pdf.