

CoSy Compilers

Overview of Construction and Operation

CoSy[®] System Documentation
ACE Associated Compiler Experts by
2003

Version: 2003.7
Date: April 24, 2003
Status: release
Confidentiality: public
Reference: CoSy-8004-construct

CoSy Compilers

by ACE Associated Compiler Experts bv.

© Copyright 1996,1999-2003 by ACE Associated Compiler Experts bv,
Amsterdam, the Netherlands.

© Copyright 1996,1999-2003 by ACE Associated Computer Experts bv,
Amsterdam, the Netherlands.

All rights reserved. No part of this document may be copied, photocopied, reproduced or translated in any way, without prior written consent of ACE Associated Compiler Experts bv.

Every care has been taken in manufacturing the supplied product and its documentation. ACE Associated Compiler Experts bv will neither assume responsibility for any damages caused by the use of its products, nor accept warranty or update claims, unless stated explicitly otherwise in a special agreement.

The information contained in this document is subject to change without notice.

Printed in the Netherlands, May 16, 2003.

Many of the designations used by manufacturers and vendors to distinguish their product are trademarks. ACE Associated Compiler Experts bv has made every attempt to supply trademark information of manufacturers and their products mentioned in this document. ACE Associated Compiler Experts bv also recognizes any trademarks used in this document but not mentioned below.

Trademark notices

CoSy[®] is an international trademark of ACE Associated Computer Experts bv.

SuperTest[™] is an international trademark of ACE Associated Computer Experts bv.

HP-UX[®] is an international trademark of Hewlett-Packard Company.

Microsoft[®], Windows[®], and Windows NT[®] are international trademarks
of Microsoft Corporation.

Solaris[®] is an international trademark of Sun Microsystems, Inc.

Linux[®] is an international trademark of Linus Torvalds.

Intel[®], Pentium[®] and Xeon[®] are international trademarks of Intel Corporation.

UNIX[®] is an international trademark of Unix System Laboratories.

Red Hat[®] is an international trademark of Red Hat, Inc.

PostScript[®], Acrobat Reader[®], Acrobat logo[®] and PDF[®] are
international trademarks of Adobe Systems, Inc.

Netscape[®] and Netscape Navigator[®] are international trademarks
of Netscape Communications Corporation.

Motorola[®] is an international trademark of Motorola, Inc.

PowerPC[®] is an international trademark of International Business Machines Corporation.

SPARC[®] is an international trademark of SPARC International, Inc.

Acknowledgements

This document describes materials some of which were originally developed with CEC funding in the ESPRIT projects COMPARE (5399), PREPARE (6516) and CORE (20575).

Partners in these projects who have contributed to some of the designs, concepts, ideas and implementations documented herein were GMD-FIRST (Germany), GMD Forschungsstelle an der Universität Karlsruhe (Germany), Harlequin Ltd (UK), INRIA (France), IRISA (France), Parsytec GmbH (Germany), PELAB (Sweden), Softlab ab (Sweden), STERIA (France), Stichting Mathematisch Centrum CWI (The Netherlands), Technische Universität München (Germany), TNO-TPD (The Netherlands), Universität des Saarlandes (Germany), University of Amsterdam (The Netherlands), University of Vienna (Austria).

Contents

1	Introduction	5
2	Construction	6
2.1	Environment	6
2.2	Compiler Construction	7
2.3	Engine Construction	10
3	Operation	13
3.1	Environment	13
3.2	Compiler Operation	14
3.3	Supervisor Operation	14
3.4	Engine Support Operation	18
4	Glossary	19
5	Acronyms	24
	Bibliography	26
	Index	27

Chapter 1

Introduction

CoSy[®] is used for constructing compilers. These compilers share various tools and common procedures in the environment in which they are constructed. The compilers themselves share components and interfaces that allow the compiler writer to focus on target code quality instead of on bookkeeping tasks. Furthermore, these compilers are unique for their scalability to host requirements, and the innovative techniques available for their construction[1].

Several details of the construction and operation of CoSy compilers have been described elsewhere. This document attempts to provide an overview of both. It is completed by a glossary, a list of acronyms and an index. Also see the bibliography for references to product documentation of the framework [7], CCMIR [4], the *Engine Writer's Guide* [5], and fSDL [6].

The target-audience of this document are experts with an intimate knowledge of compilers and compiler construction, who need to understand and familiarize themselves with CoSy compilers. To this end, the core tools and procedures receive more attention they might otherwise merit - in a balanced overview the engines and engine generators would arguably have greater weight attached to them.

This document contains three pictures: figure 2.1 showing CoSy compiler construction from its components using the fSDC and EDC tools; figure 3.1 with an example compiler specification in fSDL and EDL; and figure 3.2 showing the operation of the supervisor for the given example.

All items in the glossary appear in the index. Only acronyms appear in the index, not their full text. Bold-face page numbers in the index refer to definitions of the indexed item, either in the course of the text or in the glossary.

The languages and tools described in this document were initially developed in the COMPARE project[2], a collaborative effort of ACE Associated Computer Experts bv, GMD Forschungsstelle an der Universität Karlsruhe, Harlequin Limited, INRIA, STERIA, Stichting Mathematisch Centrum (CWI), and Universität des Saarlandes.

Chapter 2

Construction

This chapter addresses the construction of CoSy compilers. A later chapter will explain their inner workings. The first topic is the construction of CoSy compilers within in a UNIX[®]-like environment, in what we call a “framework”. Subsequently, compiler construction in such an environment is covered step-by-step with an explanation as to their purpose. Engine construction is given particular attention and is dealt with separately since it involves some important programming conventions in the form of interfaces that are a characteristic of CoSy compiler programming.

This chapter introduces the reader to the concepts of constructing a CoSy compiler; it does not attempt to cover all details. See the *Framework Document* [7] for a more thorough description.

2.1 Environment

CoSy compilers are assumed to be constructed on a host running a UNIX-like system, making use of an ISO-C compiler to compile the compiler. ISO-C permeates the whole system: everything is written in ISO-C or translated into ISO-C.

In CoSy compiler development, during its construction and subsequent generation, several levels with associated tools and interfaces are distinguished. Each lower level adds a multitude of facilities, sharing and relying on facilities at higher levels. The CoSy system provides all facilities of the lower two levels (*component* and *framework*) and some facilities of the *project* level; the higher two levels are supportive only. Levels from high to low are:

site: A UNIX-like hierarchical directory structure, and various common UNIX tools.

person: A developer’s preferred user interface environment.

project: Tools providing interfaces between different sites: document formatting, encryption, and distribution facilities.

framework: A consistent set of tools and interfaces, supporting a family of compilers and engines with common sources and generator facilities. See below.

component: One component, such as an engine, the main program together with interface specifications of a compiler, a functor library, the CCMIR specification, the framework support tool set itself, an individual generator, etc. For each component there is one single site where it is developed, where its versions and internal consistency are managed.

Everything needed to construct a compiler is stored in standardized locations at the root of the directory structure (called `$COSYROOT`) implementing a framework. This includes sources and binaries of generators, tools, interface header files, common libraries, and the manual pages of these all. Next to these are the directories with compilers, engines and their common sources. The standardized directory structure is complemented by naming conventions so that components can be easily reused and largely automatic compiler generation and consistency management are made possible. By this, a framework defines several strict interfaces between its components both at compiler-generation and at compiler-run-time. The root of this structure also is a common project home directory that allows site or person specific setting of standardized parameters to customize the framework environment to person's and site's specific conventions.

As already mentioned, "framework" is a central notion in CoSy compiler construction; it is used to indicate both the architecture (see above) and an instance of it, also called "tree" (such as in: "build-tree").

2.2 Compiler Construction

A CoSy compiler is constructed in a compiler-private directory with as main components:

micromenu: A specification to construct the compiler. It specifies which components to include and sets of paths to locations where to find these components. Also it enumerates the various options and switches to use for the tools involved in construction. A micromenu has the form of a set of definitions, which is expanded by framework tools to something equivalent to a Makefile by using the micromenu's definitions as parameters. The framework contains customized tools for constructing a compiler, an engine and a library, hiding most construction details from the compiler writer.

main program: Main module of the compiler supporting the user's options to the compiler by partially implementing them and partially converting them to engine specific options (with the help of a compiler-private options file), and then calling the generated supervisor to start the compiler.

Options are passed to arbitrary engines by designating an engine in combination with the option's name and value. The engine designation is done in the form of a dot-separated path down the engine hierarchy with a match pattern at each level mentioning either a local engine name, the class name of a local engine or a wild-card. The engine reached at the end of the path is the designated one. The beginning of the path should match at the top unless it starts with a wild-card.

SDL, EDL and CCL files: The compiler common IR structures are specified in the SDL file; the interaction of individual engines is specified in the EDL file; and the mapping of CoSy facilities like engines on compiler-host facilities like processes is specified in the CCL file. See below.

Compiler construction proceeds as follows:

1. Locating the engine classes from which the compiler is to be constructed; engine classes are found by name in a set of paths to directories.
2. Collecting all SDL files: the private SDL file of each of the engine classes with its private view domains (see below), and all common SDL files such as functor libraries, common IR, etc.. All these SDL files are fed to the fSDC (the flattener) to produce a canonicalized fSDL version of the same, called the flatform fSDL, suitable for subsequent input to other tools.
3. Collecting the EDL/CCL files and feeding them with the flatform SDL file to the EDC. EDC annotates the flatform with one view description per engine enumerating the engine's visible domains with the engine's capabilities on the IR. And it generates engine supervisor code by selectively expanding engine code templates that are written per kind of engine class in ECT. The supervisor code that EDC generates consists of a header file for every engine class with supervisor interface information, and implementation code for every composite engine class.
4. Code generation of the DMCP from the annotated flatform file by the fSDC (the DMCP generator). The DMCP consists of a private header file per engine class and shared implementation code.
5. Compilation of the generated supervisor.
6. For every engine, visiting its directory and recursively performing the engine construction procedure specified in Section 2.3.
7. Compilation of the generated DMCP.
8. Linking the result with libraries such as the ECP (the host-dependent package implementing CoSy facilities such as engines, messages, CDP, CCP, and aggregates), and the CoSy Debugger agents in the form of a library.

In figure 2.1 this is also shown: the solid lines indicate immediate use of information, the dashed lines indicate generated C implementation files, and the dotted lines indicate generated C header files. In the middle the fSDC and EDC generators are shown with the flatform fSDL information flowing between them. Everything arriving at the bottom is compiled and linked together to form the compiler's executable image.

The rôle of fSDL, EDL and CCL in compiler specification deserves some further explanation. Please refer to figure 3.1 on page 16 for an example compiler specification in fSDL and EDL.

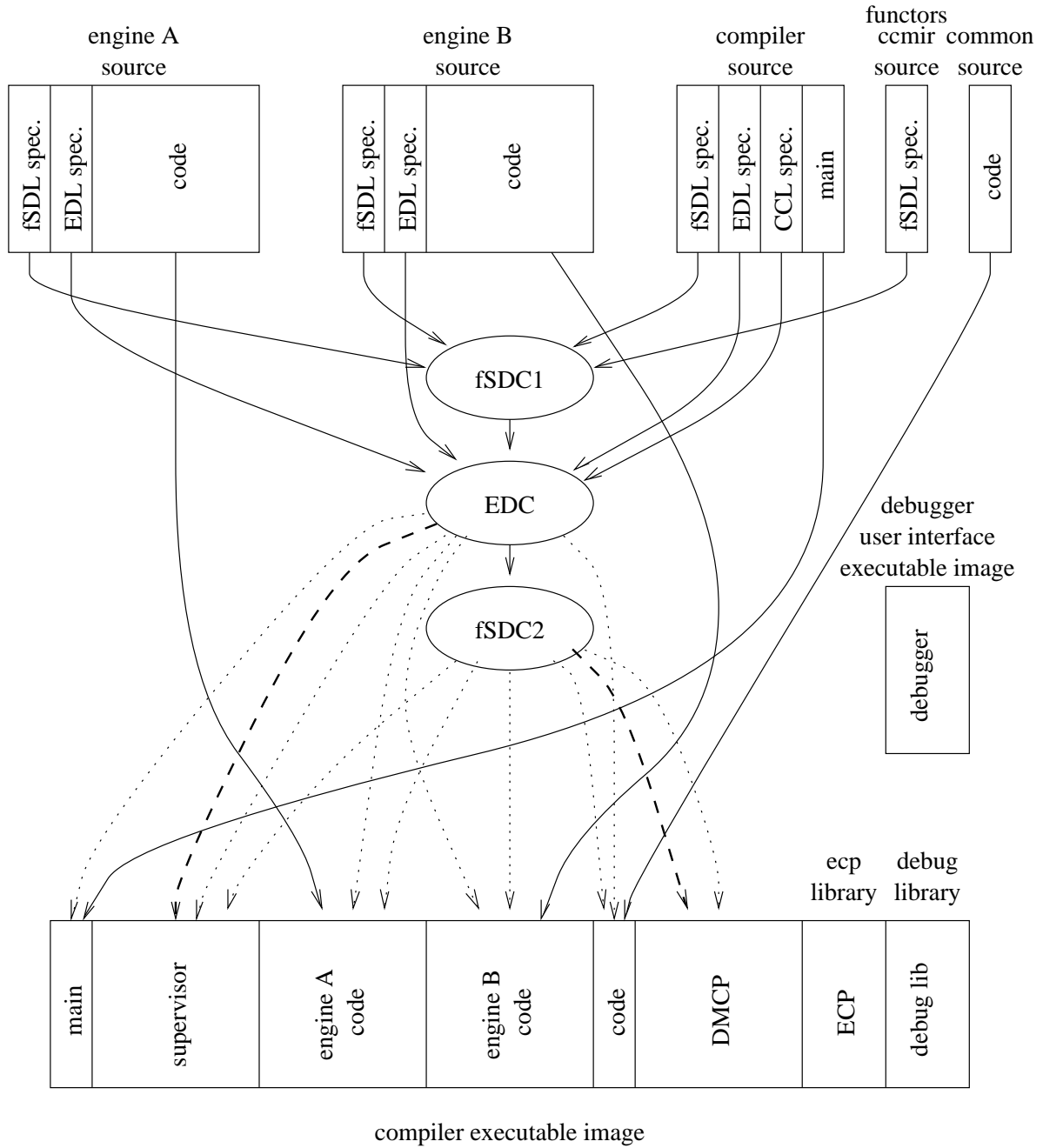


Figure 2.1: CoSy compiler construction

EDL: Central is EDL in which the logical decomposition of the compiler into engines is specified. Engines are typed by an engine class . A composite engine class specifies

component engines. At the leaves of the engine class hierarchy formed by the composite engine classes and the classes of their components are the simple engine classes of which the implementation (not their effect!) is outside the scope of the generated supervisor. The interface to an engine is given in its engine class specification, more specifically, in its side-effect specification, which enumerates its parameters, each typed by a view domain, and with an input/output and aggregate specification. The engine class specification optionally contains a list of properties of the engine class's implementation that determines the interface to the supervisor: engine flow model, engine reentrancy, server capabilities, etc. There can be multiple engines of one engine class, sharing their implementation.

fSDL: With the view domains typing the engine's parameters, fSDL enters the picture. The calculus capabilities of fSDL are used to specify reference level domains that define the data structure and alias constraints of the IR. And furthermore to specify the view domains per engine, based on these reference domains. The latter is done by:

- adding properties denoting access capabilities and the mapping of names to code names, to fields, operators, domains and functor applications; examples of such properties are, of fields: `read`, `add`, and `write`; of operators: `new`, and `destroy`; of domains: `name`, `tmp`, and `cast`; and of functor applications: `read`, `add`, `write`, `new`, `destroy`, `walk`, and `prefix`.
- explicitly specifying edge field types by view domains (instead of reference domains) to compose a structured and engine private view description, that approximates the engine run-time access pattern as closely as possible.

The description goes into so much detail of specifying IR, aliasing restrictions and access capabilities to allow EDC to do an exhaustive engine dependence analysis. This analysis by EDC is directed by the control flow of the compiler specified in EDL and determines whether the specified engine concurrency indeed is safe or needs to be sequentialized in whatever form otherwise. These sequentialization methods include just letting engines be called sequentially by the generated supervisor; fine-grain locking so that engines at run-time control their concurrency; or providing data duplication ("shadowing") to prevent engine interference.

CCL: While the compiler's logical decomposition into engines is specified in EDL, its mapping onto host facilities is controlled in CCL: clustering combines several specific engines into a cluster to decrease process overhead; configuration data specification allows to supply parameters to control the utilization of resources such as the amount of processes by a data parallel engine class. CCL specifications designate specific engines, all engines of an engine class, or simply all engines.

2.3 Engine Construction

An engine is developed in a private directory with a standardized structure. It may be shared between several compilers in the same framework and is compiled separately in the context

of each individual compiler. Two ways of sharing and incorporating an engine are supported:

source: In source form, employed for the majority of the engines.

binary: To avoid licensing problems when distributing engine sources, engines can be partially compiled, distributed and incorporated in a compiler in object form. To this end, the engine must be compiled such that it uses a functional interface to the remainder of the CoSy compiler and in particular to the DMCP. Its distribution contains a source specification of the interfaces that the engine object module assumes. Examples are the C and Fortran front-ends.

The directory implementing an engine component provides:

micromenu: Specifying construction of the engine in various levels of detail and related complexity.

SDL and EDL files: The SDL file describes the engine private domains with its private extensions to the IR and its access capabilities on the total IR. The EDL file describes the engine class's interface.

source code: Constituting the major body of the engine: its core. It can be written in ISO-C or a language providing a compatible interface to ISO-C. CoSy supports an ISO-C dialect to ease writing DMCP accesses: CoSy-C.

There are three interfaces between (user written) engine source code and the remainder of the compiler:

EEtCI: The Engine Envelope to Core Interface (EEtCI) is the interface between the supervisor, represented by a generated engine-dedicated envelope, and the core of an engine with its source code. This function call interface defines the entry functions from the supervisor envelope to the source code and from the latter to the envelope, and corresponds one-to-one with the engine's specification in fSDL and EDL. Depending on the engine class's properties, several variations of the interface are supported. The interface defines, in the simplest case, three entry functions of the core for an engine class named *my*:

```
StateType myCallInit(optionType);
int       myCallWork(StateType, agrType, Parameter1Type, ...);
void      myCallCleanup(StateType);
```

These are called by the envelope on engine creation, each time the engine must do some work, and just before engine destruction, respectively. On creation, the engine interprets its options and creates a private state structure that the envelope will pass to it in all further calls. Aggregates are passed element by element to an engine; the *agrType* indicates whether the head, an element or the tail is passed.

DMCP: The DMCP provides the engine class with specialized IR access functions that adhere to its view specification. They are supported by an implementation shared by all engines.

ECP: Facilities like error reporting and engine state management (engine identity, and global variables) are provided by the ECP.

As a composite engine has EEtCI interfaces above and beneath it, above to its ancestor through an envelope and beneath it to all of its component engine cores through their envelope, it is at the EEtCI that engines are glued together to form a compiler. Note that the engine core of a composite engine has a special name: `stub`.

Engines can be (partially) generated. Examples of generators are: BEG, a generator of back-end engines with code selector, scheduler and register allocator; and: PAG, a generator of analysis engines. See the *Framework Document* [7] for how these should be called. When they need to read or even update the flatform fSDL IR specification, they should be called just before the EDC (item 3 of the compiler construction procedure above).

Engine compilation involves, elaborating on item 6 in the compiler construction procedure above:

- 6a. Compiling its sources in the context of the compiler being constructed.
- 6b. Linking with engine local libraries such as `libc` and the host independent arithmetic packages, to form an engine object module.
- 6c. Perform the first level in the two-level linking procedure to isolate global names inside engines from visibility from other engines unless they are part of the EEtCI, ECP or DMCP interfaces, using the `maskext` facility of the framework.

Chapter 3

Operation

The internal operation of a CoSy compiler, and especially of its supervisor, is explained.

3.1 Environment

CoSy compilers are assumed to operate on a POSIX compliant host. The ECP relies on the OS to implement spinning locks and shared memory.

To enhance portability to other hosts, CoSy host dependency is hidden in the ECP and has led to few architectural constraints in compiler programming, such as to prevent the use of global variables in engines, to allow this. The ECP can map engines in three fundamentally different ways on host processes, while the CDP (Common Data Pool) in each of these models is supported by shared memory facilities:

1. linking the compiler into one image and implementing engines as threads in one process executing this image;
2. linking the compiler into one image but implementing engines as processes (with private data segments) executing this image;
3. or linking engines separately and implementing engines as processes executing these individual images (with private instruction segments).

Most platforms handle the second one efficiently and is actually implemented. Implementing the other ones requires changes to the ECP, the ECT templates and the framework's two-level compiler linking method.

Within each of these models, several engines can be combined to be executed by one process (or thread), directed by a cluster specification in CCL. This decreases the amount of processes needed by a CoSy compiler. Furthermore, the ECP can be tuned at compiler generation time, or at compiler execution time, or can even adapt itself dynamically at execution time to the availability and efficiency of certain resources.

3.2 Compiler Operation

There are several layers in a CoSy compiler:

user interface: The compiler's main program handles command-line options (see also Section 2.2), initializes the ECP and the interface to the CoSy Debugger, and starts the supervisor by creating its top-level engines and giving them the top of the IR to start working in.

supervisor: The supervisor manages the user-written engines: creates and destroys them, decomposes and distributes work over them, controls their communication, schedules them such that they don't interfere, and implements aggregate handling when they cannot. See below.

engine code: Engine code behavior is out of the scope of the supervisor, although it should adhere to its private side-effect specification in fSDL and EDL. Engine code may be generated. It implements a particular algorithm on a selected and well-described part of the IR, and only communicates with other engines through this IR. It is called through entry functions by the supervisor as part of the EEtCI, and should be reentrant. On creation of an engine, it interprets its private options and initializes its private state; after that, it can be repeatedly called to do some work, until it is finally called to destroy its private state.

IR: The IR implementation is hidden behind the DMCP interface providing engine-customized access methods: to access fields in nodes, to create, copy or destroy nodes of an operator type, to test the relation between domains and operators, etc., all on nodes allocated in the Common Data Pool. These functions may use spinning-locks to synchronize accesses; or map to an alternative copy of a field to implement shadowing in speculative interaction directed by the engine's current shadowing context.

host interface: The ECP handles the interface to host facilities. It maintains a free memory administration for CDP and CCP, and allocates more memory in huge bunches from the system, when required. When an engine is created in a new cluster, a new process is forked off, and when engines are being destroyed, it implements waiting on the death of their processes. And it supports message queues, options passing, etc.

3.3 Supervisor Operation

The operation of the supervisor is further explained. It is structured as a hierarchy of engines which interact according to a small fixed set of predefined interaction schemes that are specified in the form of EDL engine classes. Each single engine parameter can be simple or aggregated. As aggregate, it is a collection of region handles, all of the same region type, which can be operated on individually.

Figure 3.1 presents an example specification of a compiler in which its fSDL and EDL descriptions have been combined. It describes a compiler with a front-end producing an aggregate of

procedures, a data-parallel optimizer operating concurrently on each procedure, and a code generator and a speculative scheduler, each operating on one procedure at a time. Note that corresponding actual and formal engine parameter types may be different domains: the formal one specifies the view domain for the engine.

The subsequent picture, figure 3.2 shows the operation of the corresponding supervisor. Engines are indicated by solid boxes and engine classes by dotted boxes. The components in the dotted boxes together form the supervisor. Each engine is split in an upper half, its envelope in its parent's class, and a lower half, its engine core or stub, implementing its own engine class. Message flow between stub and envelopes in a class and function calling over the EEtCI inside an engine is indicated by arrows. Arrows are labelled to indicate what kind of information is flowing through. At the top is the main program interfacing through an anonymous engine class with the single engine that has a `top` property.

The remainder of this section will give details of the operation of engine classes and aggregates.

Predefined kinds of engine classes are:

pipeline: Several engines of arbitrary classes pass on their result to their successor to work on. Aggregates allow them to work concurrently on subsequent aggregate elements.

data-parallel: Every aggregate element creates an independent engine of the same class to work on it.

fork: Several engines of arbitrary classes concurrently and independently operate on one work item; the final result is composed from their individual results.

loop: As in a pipeline, engines pass on their result to a successor and the engines can operate concurrently on subsequent aggregate elements. Some special status engines (producing a condition) are used to decide whether an intermediate result is sufficiently refined to be delivered as result outside the loop; otherwise the last engine passes its result back to the first engine.

speculative: Several engines of arbitrary classes concurrently operate on one work item but only one of these results is finally chosen by a selector engine; effects of the other engines are invisible. This works by shadowing parts of the IR and restoring only the effects of the selected result.

optimistic: One engine quickly produces several potential solutions for a certain problem until a second (status) engine decides that enough of them have been generated; a subsequent third one (also a status engine) filters out the most unrealistic solutions; the remaining solutions are turned into shadows for several instances of the fourth engine class to chew on concurrently; finally a fifth (status) engine selects one of the remaining results as the best one and destroys the effects in the IR of the other ones.

Each engine class is implemented independently from every other engine class in the compiler by one engine stub for the class itself and an engine envelope for each of its component engines. In such an engine class, the related engines communicate by exchanging messages

```

// reference domains from common SDL files
domain IR: ...;
domain mirProcGlobal: ...;

// compiler description from compiler's EDL file
// starting with top engine of compiler
engine class compiler(in IR) [top]
{
    region p<>: mirProcGlobal;    // local region
pipeline
    frontend(IR, p<>);
    optimizers(p<>);
    codegen(p<>);                // repeated call for each p
    specsched(p<>);             // repeated call for each p
};

engine class optimizers(in p{: mirProcGlobal)
{
dataparallel
    optimizer(p);                // one new engine for each p
};

engine class specsched(in p: mirProcGlobal)
{
speculative
    sched1(p);                   // side-effect can be undone
    sched2(p);                   // side-effect can be undone
select schedselect(p);          // decides which branch wins
};

// view domains collected from engines' SDL file
domain frontendIR: IR + ...;
domain frontendProcGlobal: mirProcGlobal + ...;
domain optimizerProcGlobal: mirProcGlobal + ...;
domain codegenProcGlobal: mirProcGlobal + ...;
domain sched1ProcGlobal: mirProcGlobal + ...;
domain sched2ProcGlobal: mirProcGlobal + ...;
domain schedselectProcGlobal: mirProcGlobal + ...;

// engine class declarations from engines' EDL file
engine class frontend(in frontendIR; out frontendProcGlobal<>);
engine class optimizer(in optimizerProcGlobal);
engine class codegen(in codegenProcGlobal);
engine class sched1(in sched1ProcGlobal);
engine class sched2(in sched2ProcGlobal);
engine class schedselect(in schedselectProcGlobal) [status];

```

Figure 3.1: Example compiler specification

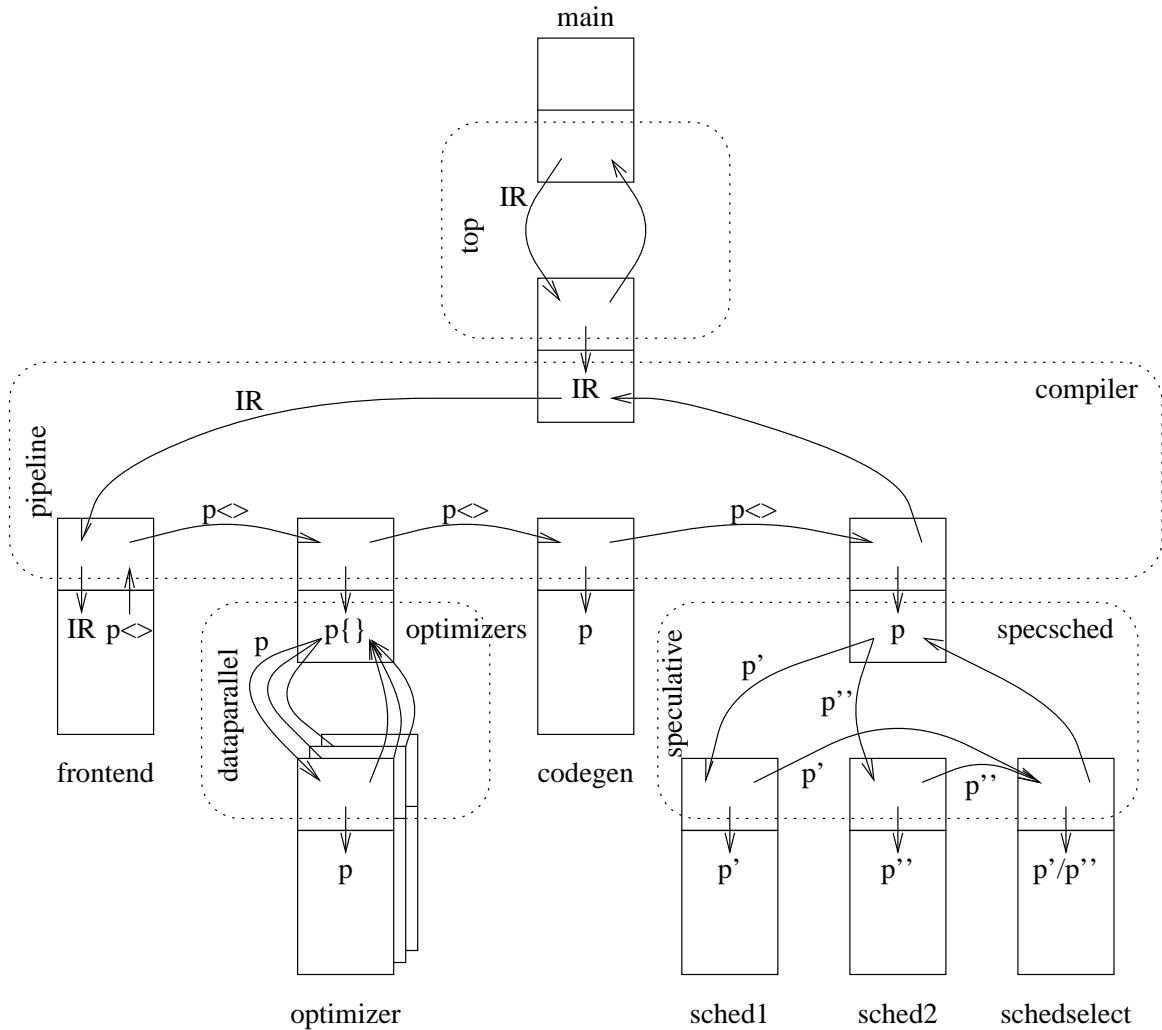


Figure 3.2: Example supervisor operation

containing region handles and meta-information according to a protocol determined by the kind of engine class, closely following the flow of regions through such an engine class . Each envelope represents its associated engine in the engine class and converts the messages received in the class to subroutine calls of the EEtCI inside the engine, and vice-versa. Below the EEtCI can be the stub of another engine class on a lower level or the engine code of a user engine.

User engines can also be present in the middle of the hierarchy but these are only allowed to synchronously call engines marked with the server property; these get a special envelope so that they are able to handle calls for work from several engines one by one.

Aggregate implementation is everywhere in the supervisor. Every single actual (or formal)

engine parameter can be annotated with one of three kinds of aggregate: set, series or block. Apart from indicating that the parameter *is* an aggregate, it conveys various degrees of dependence that should be enforced internally between the aggregate elements. For an actual parameter, this dependence is with respect to the operation of the engine call in which it appears; for a formal parameter, the dependence is with respect to the operation of the whole engine class. In order of increasing severity, the kinds of aggregates are:

set: “{}”, no mutual dependence between the aggregate’s regions; the elements can be arbitrarily reordered during the operation

series: “<>”, dependence on preceding regions of the same aggregate; the order between the elements should be preserved during the operation

block: “[]”, dependence on all other regions of the same aggregate.

The actual concurrency of the engine’s operations on the individual handles depends on this kind and is taken into account by the supervisor. It determines scheduling of the engine, distributing aggregate elements over several independent engine calls of the same class or over several consecutive calls of the same engine, and it determines where aggregate are composed by collecting results from engine calls and where they are created or just pass by.

3.4 Engine Support Operation

The engine can call its envelope through the EEtCI, operate on the IR through the DMCP, and use the ECP for some standard facilities.

The EEtCI also contains up-calls for the engine to call the supervisor on its own behalf when its needs more work or produces a result out-of-line with the supervisor’s calls for work to the engine. These up-calls are supported by the envelope and their presence and form depends on the engine class’ flow model property.

The accesses by an engine to the IR are supported by the DMCP. Its functions hide the physical structure of the IR nodes and their connections. An access function generally accepts a handle on a node and is able to get or set the value of a field. In this it takes into account whether the field is shadowed and then uses the engine’s current shadow context number to help selecting the appropriate copy. Accesses can be trapped by the CoSy Debugger, when indicated, so that they can be traced per engine.

The supervisor as well as the user written engines call upon the ECP for various primitive operations. The ECP allows the CoSy Debugger to trace engine and message related operations when indicated. The debugger interfaces to DMCP and ECP to retrieve the compiler’s status.

Chapter 4

Glossary

This is a definition of terms used in CoSy. The meaning given here may differ from the one in your dictionary, but is the *intended* meaning used in the documents in the project. Acronyms and their meaning are summarized in the next chapter. This glossary only uses acronyms as entries when in the course of time their meaning has surpassed the text they stand for.

Aggregate: Collection of region handles, all of the same region type, that can be specified as single engine parameter, and then indicates that engine work may be performed concurrently on (or producing) each of the individual handles. The actual concurrency depends on the kind of aggregate: set, series or block.

Back-end Generator: BEG, the generator of back-end engines based on an initial development at GMD, the University of Karlsruhe; productized and significantly extended by ACE. See also the *BEG-CoSy Manual* [3].

CADESE: The version control system used by CoSy. Being essential to the operation of the CoSy framework, it is packaged with the CoSy tool set. See also the *CADESE Document* [8].

Code Selector: The transformer from MIR to LIR is called a code selector. The current code selectors are generated by BEG. See also the *BEG-CoSy Manual* [3].

Common CoSy Medium-level Intermediate Representation: The CCMIR is a particular instance of a MIR. It is common in the sense that it is able to support representations for programs in all CoSy source languages and targeted for all CoSy target architectures. See also the *CCMIR Definition Document* [4].

Common Control Pool: The CCP is the shared memory facility that supports the storage of supervisor data structures that are independent of the program being compiled.

Common Data Pool: The CDP is the shared memory facility that supports the storage of a particular instantiation of the intermediate representation of a program, whereas the

IR is the specification of its data structures. Often the two terms are used as synonyms.

COMPARE: Project consortium that developed a prototype of the CoSy compilation system. Partners were: ACE Associated Computer Experts bv, GMD Forschungsstelle an der Universität Karlsruhe, Harlequin Limited, INRIA, STERIA, Stichting Mathematisch Centrum (CWI), and Universität des Saarlandes. The project ended in the spring of 1995.

Compiler Configuration Language: CCL, formalism in which to specify parameters of the mapping of CoSy entities like engines and messages on resources provided by the compiler host, in order to tune the algorithms of the generated supervisor to what can be efficiently supported by this host.

CoSy: Model of cooperation of components of a compiler that is characterized by interaction that does not follow the sequential flow of control of usual compilers.

Also used to indicate the more special case of constructing a compiler from engines that communicate through shared memory and are supervised by a generated distributed supervisor.

But also any compiler constructed according to such a model, or the toolbox with which such a compiler can be built.

CoSy-C: ISO-C dialect that allows to write DMCP accesses in a concise way. See also the *Engine Writer's Guide* [5].

CoSy Debugger: The interactive facility to inspect and debug a CoSy compiler with special features to intercept, control and display supervisor and IR access events.

Data Manipulation and Control Package: Set of C types, functions and macros that give engines access to the IR. The DMCP is generated by the fSDC and isolates the engines from a particular implementation of the IR. It comprises the implementations of the engines' views. See also the *fSDL Definition Document* [6].

Domain: A type in fSDL: a set of operators, fields and properties. Operators again have private fields; domains, operators and fields have properties describing access capabilities, aliasing conditions, etc. Each field is typed by a domain.

A reference domain doesn't specify access capabilities and is shared by the view domains importing it. View domains additionally specify access capabilities and mapping of the domain name to a code name, and are used to type engine class parameters. See also the *fSDL Definition Document* [6].

Engine: Component of the compiler that can only communicate with other parts of the compiler through the IR. An engine may contain other engines (called a *Composite Engine*). An engine is instantiated from an engine class in the EDL specification. Several engines can be clustered into one process in the CCL specification during configuration time.

Engine class: An engine class describes the input/output behavior and the algorithm of an engine. It is specified in EDL. A simple engine class is a leaf class in the engine hierarchy formed by the engine interaction specifications in EDL. With a simple engine class an algorithm is associated (e.g. written in C and stored in some source files) and a side-effect specification written in fSDL.

Engine Code Template: ECT, language (or specification in this language) that supports writing (usually supervisor) engine code based on an actual IR description in fSDL and a compiler description in EDL. EDC contains an expander for ECT specifications and uses ECT templates to generate supervisor stubs and envelopes. ECT allows decoupling of analysis in EDC from the coding of the specific supervisor algorithms.

Engine Control Package: The explicitly host-dependent runtime package that provides primitives for engine control, synchronization, shared memory management, option handling, aggregated and simple message passing and some other common facilities. It is independent of any actual compiler specification and configuration.

Engine core: The source code module of an engine class, implementing its algorithm.

Engine Description Compiler: Tool that compiles a compiler description consisting of several engine descriptions in EDL and the collection of all engine domains in flatform fSDL, to the code of a distributed supervisor and to an update of that flatform fSDL that makes fSDC generate a DMCP with the proper views.

Engine Description Language: Formalism in which to specify the compiler's internal interaction between its engines and the side-effect that execution of these engines will have on the IR.

Engine Envelope to Core Interface: EEtCI, the function call interface between the supervisor, represented by an engine-dedicated envelope, and the engine core of the engine with its source code. See also the *Engine Writer's Guide* [5].

Envelope: A generated engine-dedicated supervisor component that interfaces the engine core through the EEtCI to the supervisor. See also the *Engine Writer's Guide* [5].

Platform: Canonicalized fSDL description produced by the first, flattening, phase of the fSDC. See also the *fSDL Definition Document* [6].

Framework: The architecture (or an instance of it) of a standardized directory structure naming various framework components, that defines strict interfaces between these components both at compiler-generation and at compiler-run-time. See also the *Framework Document* [7].

Front-End: A compiler component whose main task is to transform source code from its lexical representations to some semantic equivalent expressed in an IR. See also the *CCMIR Definition Document* [4].

full-Structure Definition Compiler: Tool that canonicalizes an fSDL description of an IR to its equivalent flatform fSDL description, and that translates such a flatform fSDL description into a corresponding DMCP. See also the *fSDL Definition Document* [6].

full-Structure Definition Language: Formalism in which to specify an IR, and access capabilities of engines to such an IR. The prefix “full” is an anachronism. See also the *fSDL Definition Document* [6].

Functor: A parameterized data type constructor that can be specified in fSDL to support IR data structure definition. Examples are the LIST functor for non-embedded lists of arbitrary elements, and the BITUNI functor for embedded bipartite unidirectional graphs between two kinds of arbitrary elements. A functor application takes functor-properties and domains as parameters and produces one primary result domain. See also the *fSDL Definition Document* [6] and the *Engine Writer’s Guide* [5].

High-level Intermediate Representation: HIR, a front-end’s internal representation of a source program, directly corresponding to the source. See also the *CCMIR Definition Document* [4].

Intermediate Representation: The central data repository through which engines communicate with each other. An IR is specified in fSDL. It has one root node, of domain type IR. See also the *CCMIR Definition Document* [4].

Low-level Intermediate Representation: LIR, Linearized assembly-like code, generated by a code selector from the MIR code. See also the *CCMIR Definition Document* [4] and the *BEG-CoSy Manual* [3].

Medium-level Intermediate Representation: MIR, common representation of program code; to a large extent independent of source language and target processor. Generated by a front-end and consumed by a code generator. In a CoSy compiler, most of the optimization is done at this level. See also the *CCMIR Definition Document* [4].

Micromenu: Specification how to generate a component, such as a CoSy compiler or engine. See also the *Framework Document* [7].

PREPARE: Companion ESPRIT project consortium of COMPARE, oriented towards the support of HPF, making extensive use of COMPARE facilities.

Program Analyzer Generator: Analyzer generator with a general data flow equation solver.

Region: The collection of nodes that can be reached by a certain engine, remaining in its view, by starting at the handle on a node that was passed as a parameter to it and following only edges that correspond to readable edge fields in its view. It is an instance of a region type.

Region type: The directed graph of domains with one root domain, which usually is the type of a parameter of an engine class. For each readable edge field in a domain, it has an edge to the domain that is the type of the edge field.

Shadowing: The method employed by CoSy compilers to solve engine interference in the IR by duplication of certain parts of the IR.

Side-effect specification: The side-effect specification of an engine (class) is its list of parameter declarations, each with an input/output and an aggregate specification, and each typed by a domain.

Stub: The generated engine core of a composite engine class.

Two-level Linking: The method to link object modules (in a CoSy compiler) such that one engine can be composed from several object modules and libraries with global procedure names that are made static (private to the object module) before linking with other engines to form the compiler module. See also the *Framework Document* [7].

View: A view of an IR specification is a subset of its nodes, edges and fields, extended with properties that specifies the allowed operations on those nodes, edges and fields. The DMCP implementation provides support for an engine to access the IR in the Common Data Pool only for the nodes, edges and fields, and only for those operations that are in the view. The view of an engine of a particular engine class is computed from the side-effect specification in its engine class specification. See also the *Engine Writer's Guide* [5].

Chapter 5

Acronyms

This is a summary of acronyms used in CoSy.

See the glossary (Chapter 4) for more informative texts. Note that only the acronyms appear as keys in this document's index, indexing the uses of the acronyms and their full texts.

BEG: Back-end Generator.

CADESE: Computer Aided Distribution Environment for Software Engineering.

CCL: Compiler Configuration Language.

CCMIR: Common CoSy Medium-level Intermediate Representation.

CCP: Common Control Pool.

CDP: Common Data Pool.

COMPARE: COMpilers for Parallel ARchitecturEs.

CoSy: COmpilation SYstem.

DMCP: Data Manipulation and Control Package.

ECP: Engine Control Package.

ECT: Engine Code Template.

EDC: Engine Description Compiler.

EDL: Engine Description Language.

EEtCI: Engine Envelope to Core Interface.

fSDC: full-Structure Definition Compiler.

fSDL: full-Structure Definition Language.

HIR: High-level Intermediate Representation.

IR: Intermediate Representation.

LIR: Low-level Intermediate Representation.

MIR: Medium-level Intermediate Representation.

PAG: Program Analyzer Generator.

PREPARE: PRogramming Environment for Parallel ARchitEctures.

Bibliography

- [1] M. Alt, U. Assmann, and H. van Someren. CoSy Compiler Phase Embedding with the CoSy Compiler Model. In P. A. Fritzson, editor, *Compiler Construction, Lecture Notes in Computer Science 786*, volume 786 of *Lect. Notes in Comp. Sci.*, pages 278–293. Springer, April 1994.
- [2] U. Assmann, H. van Someren, and Alt M. Compilers for Parallel architectures – The COMPARE project. In Sips H.J., editor, *Fourth International Workshop on Compilers for Parallel Computers*, volume 786, pages 451–454. Delft University of Technology, Faculty of Applied Science, Advanced School for Computing and Imaging, December 1993.
- [3] ACE Associated Compiler Experts bv. BEG-CoSy Manual. Ref. CoSy-8005-beg, 2003.
- [4] ACE Associated Compiler Experts bv. CCMIR Definition, Specification in SDL, Description and Rationale. Ref. CoSy-8002-ccmir, 2003.
- [5] ACE Associated Compiler Experts bv. Engine Writer’s Guide. Ref. CoSy-8001-ewg, 2003.
- [6] ACE Associated Compiler Experts bv. fSDL, Definition and Generated Interfaces. Ref. CoSy-8003-fsdl, 2003.
- [7] ACE Associated Compiler Experts bv. The CoSy Framework, A compiler construction system. Ref. CoSy-8006-fw, 2003.
- [8] ACE Associated Computer Experts bv. Cadese Change Control. Ref. CoSy-8077-cadesecc, 2003.

Index

- aggregate, 8, 10, 11, **14**, 15, 17–19, **19**, 21, 23
- BEG, 12, **19**, 24
- CADESE, **19**, 24
- CCL, 8, **10**, 13, **20**, 24
- CCMIR, 7, **19**, 24
- CCP, 8, 14, **19**, 24
- CDP, 8, **13**, 14, **19**, 23, 24
- code selector, 12, **19**, 22
- COMPARE, **5**, **20**, 22, 24
- CoSy, **5**, 5–8, 11, 13, 14, 19, **20**, 22–24
- CoSy Debugger, 8, 14, 18, **20**
- CoSy-C, 11, **20**
- DMCP, 8, 11, **12**, 14, 18, **20**, 20–24
- domain, 8, **10**, 11, 14, 15, **20**, 20–23
- ECP, 8, **12**, 12–14, 18, **21**, 24
- ECT, **8**, 13, **21**, 24
- EDC, 5, **8**, 10, 12, **21**, 24
- EDL, 5, 8–11, **9**, 14, 20, **21**, 24
- EEtCI, **11**, 12, 14, 15, 17, 18, **21**, 24
- engine, 5, 7–15, **9**, 17–23, **20**
 - class, 7–12, **9**, 14, 15, 17, 18, 20–23, **21**
 - core, 11, 12, 15, **21**, 23
- envelope, 11, 12, 15, 17, 18, **21**
- flatform, **8**, 12, **21**, 22
- framework, **7**, 10, 13, **21**
- front-end, 11, 14, **21**, 22
- fSDC, 5, **8**, 20, 21, **22**, 24
- fSDL, 5, 8, **10**, 10–12, 14, 20–22, **22**, 24
- functor, 7, 8, 10, **22**
- HIR, **22**, 25
- IR, 8, 10–12, 14, 15, 18, 20–23, **22**, 25
- LIR, 19, **22**, 25
- micromenu, **7**, 11, **22**
- MIR, 19, **22**, 25
- PAG, 12, **22**, 25
- PREPARE, **22**, 25
- region, 14, 17–19, **22**
- region type, 14, 19, **22**
- SDL, 8, 11
- shadowing, 10, 14, 15, **23**
- side-effect specification, **10**, 14, 21, **23**
- stub, 12, 15, 17, 21, **23**
- two-level linking, 12, **23**
- view, **8**, 10, 12, 15, 20, 22, **23**