Basic Research in Computer Science

BRICS NS-02-8 P. D. Mosses (ed.): AS 2002 Proceedings

Proceedings of the Fourth International Workshop on

Action Semantics
AS 2002

Copenhagen, Denmark, July 21, 2002

Peter D. Mosses
(editor)

BRICS Notes Series **NS-02-8**

ISSN 0909-3206 **December 2002**

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/02/8/`

Foreword

Action Semantics [http://www.brics.dk/Projects/AS/] is a practical framework for formal semantic description of programming languages. Since its appearance ten years ago, Action Semantics has been used to describe major languages such as Pascal, SML, ANDF, and Java, and various tools for processing action semantic descriptions have been developed.

The AS 2002 workshop included contributed talks on:

- recent development and applications of tool support for Action Semantics;
- ways of increasing the modularity of action semantic descriptions and of the definition of Action Notation;
- analysis of information flow and types of actions; and
- test suite generation based on Abstract State Machines.

Thanks to Egon Börger for giving an invited talk on

- *Computation and specification models: A comparative study*.

The workshop concluded with a discussion of plans for future projects involving Action Semantics.

Thanks to the authors for the papers that they have contributed to this Proceedings volume. These papers should not only improve awareness of ongoing projects within the Action Semantics community, but also encourage more widespread interest in the use and development of Action Semantics. (The papers have not been refereed, and copyright remains with the authors, who are encouraged to submit similar papers to workshops and conferences with wider audiences.)

Thanks to Dines Bjørner for the suggestion of holding an Action Semantics workshop as a satellite event of FME, and to the local organizers of FLoC'02 in Copenhagen for logistic support. Finally, thanks to BRICS for financial support in connection with the workshop, and to Uffe Engberg for help with the production of the Proceedings.

Peter D. Mosses
BRICS & Department of Computer Science
University of Aarhus, Denmark

Final Programme

21 July, Morning

08:45-09:00 Session 1: Opening and Introduction

09:00-10:30 Session 2: Tools and Applications

09:00 How ASF+SDF technology can be used to develop an action semantics environment
Mark van den Brand and Jurgen Vinju, CWI Amsterdam, The Netherlands

09:30 The Abaco system: An action tool for programming language designers
Luis Carlos Meneses, Hermano Moura, Wanderley Cansancao, Monique Monteiro, and Pablo Sampaio; Federal University of Pernambuco, Brazil

10:00 Using Abaco to animate a real-time specification language
Adnan Sherif, Ana Cavalcanti, and Hermano Moura; Federal University of Pernambuco, Brazil

10:30-11:00 Refreshments

11:00-12:30 Session 3: Foundations

11:00 An object-oriented view of action semantics descriptions
Cláudio R. V. Carvilhe and Martín A. Musicante, Federal University of Paraná, Brazil

11:30 An extensible definition for action notation
Luis Carlos Meneses and Hermano Moura, Federal University of Pernambuco, Brazil

12:00 A modular SOS for action notation, revisited
Peter D. Mosses, University of Aarhus, Denmark

12:30-14:00 Lunch

21 July, Afternoon

14:00-15:30 Session 4: Related Frameworks

14:00 Computation and specification models: A comparative study (Invited talk)
Egon Börger, University of Pisa, Italy

15:00 Using ASM specification for automatic test suite generation for mpC parallel programming language compiler
Alexey Kalinov, Alexander S. Kossatchev, Mikhail Posypkin, and V. Shishkov; ISP/RAS Moscow, Russia

15:30-16:00 Refreshments

16:00-17:00 Session 5: Foundations

16:00 The analysis of secure information-flow in actions by abstract interpretation
Ki-Hwan Choi, Hanyang University, South Korea; Kyung-Goo Doh, Hanyang University, South Korea; and Seung Cheol Shin, Dongyang University, South Korea

16:30 Type inference for the new action notation
Jørgen Iversen, University of Aarhus, Denmark

17:00-17:30 Session 6: Tools

17:00 AN-2 tools
Tijs van der Storm, University of Amsterdam, The Netherlands

17:30-18:00 Session 7: The Future of AS and Related Frameworks

17:30 Discussion

Author Index

Table of Contents

Tools and Applications

Foundations

Related Frameworks

The ABACO System: an Action Tool for Programming Language Designers

Hermano Moura,
Luis Carlos Menezes,
Monique Monteiro,
Pablo Sampaio,
Wanderley Cansanção

Centre of Informatics,Federal University of Pernambuco
CP 7851, CEP 50732-970, Recife, Brazil.
E-mail: {hermano,lcsm,mlbm,pas,woc}@cin.ufpe.br

Abstract We describe the structure of the ABACO System, an action semantics tool that enables the programming language designer to build and test action semantics descriptions. The system is formed by a set of compiler generation tools like parser generators, specification processors, including sort checkers and interpreters for action semantic descriptions. These tools can be used to process action semantics descriptions for programming languages and to generate interpreters and compilers for them.
ABACO's tools are integrated in a graphical user interface environment, allowing a friendly design of new specifications and their immediate test. One important ABACO's feature is its ability to handle complex descriptions that use new user-defined actions. It allows the building of semantic libraries that can be useful to facilitate the description of complex languages and to improve the reusability of a specification.
We also propose to demonstrate the newest release of the system, which is freely available on the net (http://www.cin.ufpe.br/∼rat), and which is based on a more generic action semantics interpreter that can be used to execute both current versions of action notation and allows the user to describe and to execute action extensions for them.

1 Introduction

Natural language based programming languages specifications frequently suffer from ambiguities and other problems. These problems tend to be solved by the use of mathematical formalisms and software tools which aid in the language specification process and allow the resultant specification to be complete and free of errors.

Secondly, the manual writing of interpreters and compilers tend to be a tedious and error prone activity, and the generated tools are not reusable for other languages. We should have tools capable of automatically generating compilers based on the syntactical and semantic specifications of the source languages.

The main idea in this article is to demonstrate a tool whose aim is to help to solve the cited problems by offering an iterative environment for programming languages specifications based on an already known formalism - Action Semantics. This tool - the ABACO system - is already able to generate interpreters and, in future versions, it will be capable of generating portable code.

First we shall give a general vision of the system, describing its use and functionalities. Secondly we will describe its formal notation, then its architecture, and finally possible improvements and future directions.

2 The ABACO System

The ABACO (Algebraic Based Action COmpiler) system consists of an editing environment for action semantics [Mos92] which helps the programming language designer to build and test action semantics descriptions. It contains tools that help the design of action semantics descriptions, such as parsers generators, descriptions checkers and interpreters.

The ABACO System consists basically of a specification compiler, an action library and a graphical system interface.

The specification compiler is used by the system interface to process abstract syntax and action semantics descriptions. It receives a source specification, written in the algebraic formalism used by ABACO, processes it to verify if there are syntactical errors, and produces interpreters for the specification. The produced interpreter is able to recognize valid terms and evaluate them, reducing them by applying the rewriting equations contained in the source specification.

The action library is formed by the following components:

- the action notation version 1 and version 2 descriptions written in the algebraic formalism used by ABACO. They allow ABACO to understand action semantics descriptions.
- the action interpreter that is responsible to simulate the performance of action written in the first version of the action notation.

The system interface is a friendly graphical interface, which consists of menus, specification editors, action editors and console windows. Each one of these components is used in a phase of the programming language design, (ie.: specification or testing). The menus contain options which allow us to create new projects, add specifications to the currently open project, remove specifications from the currently open project, print specifications, configure and preview printing, access help information (including searching options) and access the particular functionalities of the system. Many of the menu items and menu options are general and self-explainable, others are system specific, such as the ones which will be explained in the next subsections.

2.1 Specific Menu Items

The Run menu item contains options which allow us to compile the current edited specification (ie.: Compile option). When selected, this option creates a

compilation window that shows the status of the specification processing and eventual compilation errors found by the specification compiler. If the compiler does not find any errors, a console window will be opened to allow the user to test the specification. In future versions options for type checking and generation of C++ and Java standalone code may be included.

The available tools in the Tools menu item allow us to interpret and debug actions currently edited in the action editor window. If the option for interpreting is chosen, the results of the execution of the action are displayed in a new window, and are expressed by means of its transients, bindings and storages. The option for debugging the actions performs a similar task, but shows each step of the execution of an action. There are also options to export specifications to HTML and LaTeX formats. In future versions options for customizing this menu item by means of adding and removing tools may also be included, and will allow the user to add and remove system functionalities.

2.2 Editors

The editors present in the ABACO system consist of specification editors, action editors and console windows.

The specification editors enable the user to edit specifications, which should be written in the ABACO algebraic formalism, and may be created by the menu option for creation of new specifications (in the menu File item), or with the project panel, a graphical component responsible for displaying all existing specifications in current project. The specifications are organized in a tree structure, which is stored in project files in XML (*Extensible Markup Language*) format. The system contains at most one current project file, which can be selected by the File menu item.

The action editors enable the typing and testing of small actions. Once the action is typed, the user can interpret or debug it by selecting the corresponding option in the menu bar.

The console window allows the user to test compiled specifications, typing valid terms and evaluating them.

3 ABACO's Formal Notation

To define action semantics descriptions, ABACO uses an algebraic specification formalism based on unified algebras' features. An specification written in this formalism is composed by:

- An introduces sentence that lists the datatypes (sorts) defined by the specification:

  introduces: $sort_1$,$sort_2$,...,$sort_N$
- A needs sentence that lists other specifications used by the current specification.

needs: $spec_1$,$spec_2$,...,$spec_N$

– A list of formula that defines the relationship existent between the defined sorts and the behavior of the defined operators for these sorts. The ABACO's specification formalism contains the following kinds of formula:
  - Operator's functionalities: defines signatures for the specification operators. This sentence has the following syntax:
    $opName :: arg_1 , ... , arg_N \rightarrow result.$
    which declares the operator $opName$, which accepts arguments of the sorts arg_x and returns a new element of the $result$ sort. $opName$ can be formed by a sequence of words and the arguments placement is represented by the symbol _ inside the operator name. The symbols arg_1, arg_2, ..., arg_N are the operands of this operator. The symbol $result$ specifies the return value after application of this operator.
  - Subsort formulae:
    eq $sort_1 <= sort_2$.
    which defines $sort_1$ as a subsort of $sort_2$;
  - Sort equation formulae:
    eq $sort_1 = sort_2$.
    which defines that $sort_1$ and $sort_2$ are the same sort;
  - Sort element formulae:
    eq $el : S$.
    which declares that el is an element (individual sort) of sort S. Equations beginning with eq defines the datatype hierarchy in current specification.
  - Rewrite equation:
    re $term_1 = term_2$.
    which declares that the term1 can be rewritten as the term2. Terms in rewriting equations can be organized with the "|"-notation used by action semantics; Variables names are prefixed by " ". Typed variables are expressed using the following notation "($varName$: $type$)".
  - Pretty printer statements:
    pp $opName = rule$.
    which specifies the way on which ABACO displays terms formed by the operator opName. The rule fields can be a sequence of the following operators:
    * "string" : displays this string;
    * n : new line;
    * % x : displays the x-th operator's argument;
    * (indent "s" rule): displays rule prefixing it with the string "s"
  - syntax declaration: specifies operators functionalities, subsorts and elements using a BNF like notation. Example:

```
syntax
stack = [[ "push" element stack ]] |
        [[ "pop" stack ]] |
        [[ "empty-stack" ]] .
element = [[ integer ]] |
          [[ string ]] .
```

```
end syntax
```

`"push"` and `"pop"` are operators,`"empty-stack"` is a subsort of sort stack,`integer` and `string` are elements.

Below, an example of a stack in the abaco's notation

```
introduces: stack .
needs: integer .
op push _ _ :: integer, stack -> stack .
op pop _ :: stack -> stack .
op top _ :: stack -> integer .
eq empty : stack .
eq stack <= datum .
re pop push ~x ~y = ~y .
re top push ~x ~y = ~x .
```

4 Implementation

In this section we describe the implementation architecture of the Abaco system. The Abaco System is composed of the following modules :

- Algebric Specifications Compiler: Process unified algebras specifications. You can write your own programming language and programs in this language. At this point, it defines:
  - Data structures as lists, sorts and other structures ,allowing build other data from this structures;
  - Programs using this data structures or creating abstract data types from these primitive data;
  - Action Notation, starting from ActionNotation 1;
  - Programming Languages Descriptions, using Action Notation.

  The Algebric Specification Compiler is composed with the sub modules:
  - Parser Generator - Generates a parser for the defined language.
  - Specification Parser - Checks a programming language description and processes it, accepting valid programs and executing it.
  - Plug-ins - Allow native data types definition. They are activated using definition import "className".
  - Execution engine - Applies the operators, using matching patterns and manipulation variables functions.
  - Specification processor - Coordinates all modules above, obeying this sequencing of events :
    * Calls the Specifications parser;
    * Resolves dependencies(imports);
    * Builds a parser and resolves necessary imports;
    * Recognizes rewriting rules and terms;
    * Defines execution engine.

– Actions library: With this module, the Abaco System can process Action Semantics descriptions. It can be described in two parts:
  - Action Semantics description : Uses Action Notation 1. In future releases, Abaco will support Action Semantics descriptions using Action Notation 2.
  - Action Interpreter : Checks actions, returning an outcome of each execution.
– Graphical User Interface: Offers specifications and actions editing facilities and executes the system's modules. You can create projects, adding one or more specifications, validating and executing your specifications.
– XML Module: In the Abaco System, a project is defined with a unique DTD. So, any project built using Abaco is according this DTD. And any project is built XML-like. This module is interconnected with the Algebric Specifications Compiler.

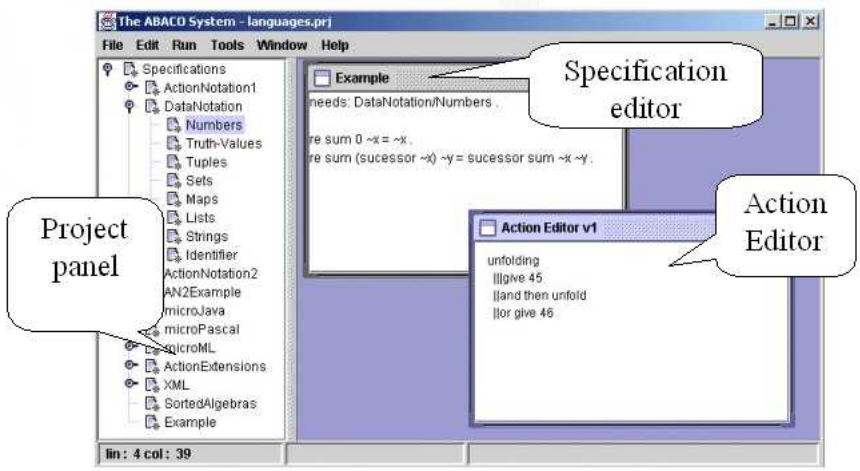

Figure1. Specification and action editors

– Help System: Shows how to use the Abaco System and has a tutorial about Action Notation 1 and Action Notation 2. In Action Notation 1 tutorial, you can execute small examples, viewing the outcome of the action executed. This help system is useful as a reference for novice action semantic users.
– Specifications Exporter: Exports one or more specifications to HTML,LaTeX and ASCII text.
– Action Debugger: Edits one or more actions and calls Action Debugger. You can execute step-by-step, viewing incoming and outcoming information.

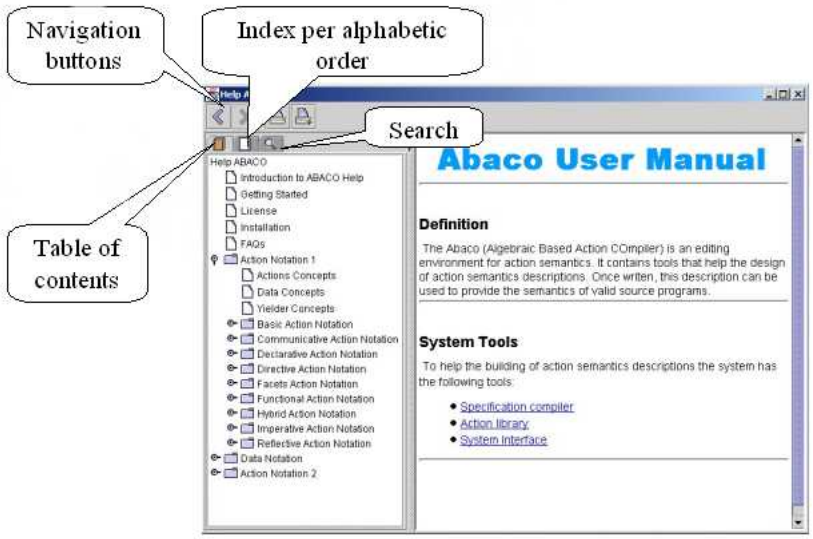

Figure2. Help system

Going to previous page, you can see screenshots of Abaco System. In the first screenshot, you can see specification and action editors. In the second one, you see the help system.

5 Conclusions and future work

Abaco is a tool designed to aid programming languages specifications by means of Action Semantics formalism. The use of this kind of tools, in spite of not being common at present time, is extremely useful to avoid errors in programming languages implementations due to ambiguities commonly found in natural language based descriptions.

A proposal to improve Abaco's performance consists of adapting the specifications execution engine to support code generation, which would improve the speed of specification executions. In this intention, an API (*Application Programming Interface*) named Abaco Generic Object Oriented Language - AGOOL, was implemented to dynamically generate and load bytecodes (instructions interpreted by the Java Virtual Machine - JVM [LY97]) on memory or on disk during the execution of a specification. It should also be possible to adapt this API to generate native code. It would cause a maximum performance, but reduce portability. The user should be able to choose between these two approaches (bytecode or native code generation) in Abaco interface.

Another improvement to be added to future versions of the system is the homogeneity of the syntactical analysis system. At present time, Abaco has two parsers: one automatically generated by a parser generator and other dynami-

cally generated by an internal compilation engine whose function is to recognize the terms of the specification. The first one was important in the beginning of the implementation due to instabilities and inefficiencies in the first versions of the compilation engine. However, the use of a single tool for both parsers will contribute to the homogeneity of the system. This tool is already at implementation stage and will consist both of a lexical analyzer generator and a parser generator. A regular expressions library and a deterministic automata based lexical processor were implemented for the module responsible for the lexical analyzer generation. The module responsible for the parser generation is capable of generating simple LR parsers [ASU85], efficient and sufficiently powerful for Abaco purposes, and supports ambiguous grammars and the definition of precedence and associativity rules. These rules are not supported in Abaco's current version. The support to ambiguities is achieved by generating two optional forms of parsers: backtracking parsers or pseudo-parallel parsers, in the same way it was proposed by ASF-SDF [BH89].

Other future works are the improvement of the interface with the Java language, which would allow new primitive types to be defined in Abaco through Java source code; the support to other types of specifications, such as unified algebraic specifications, operational semantics and others; and the creation of a generic configurable editor, which would use some kind of language to describe syntax (formatting) and semantics (typesetting) of the input text and an interface to a compilers generator to the incremental construction of syntactical trees.

References

ASU85. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wwesley, 1985.

BH89. J. A. Bergstra and Klint P. Heering. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.

LY97. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Publishing Company, Inc., 1997.

Mos92. Peter D. Mosses. *Action Semantics.* Cambridge University Press, 1992.

Using ABACO to Animate a Real-time Specification Language

Adnan Sherif[1] *, Ana Cavalcanti[2], and Hermano Moura[1]

[1] Informatics Center, Federal University of Pernambuco,
P.O. Box 7851, Cidade Universitária,
Recife - PE CEP 50740-540
Brazil
ams,hermano@cin.ufpe.br
[2] Computing Laboratory, University of Kent at Canterbury,
Canterbury, Kent, CT2 7NF
England
A.Cavalcanti@ukc.ac.uk

Abstract. The Algebraic Based Action COmpiler (ABACO) is an action semantics specification tool. It is used to explore and animate language semantics specification written in Action Notation. In this work, we focus on the framework provided by the ABACO tool that allows specification using Java. This framework can be used in the definition of new yielders and actions like a Java class. The yielders and actions can be used to represents the environment the system interacts with.
We illustrate the use of the framework with a case study: We study a real-time specification language similar to Timed CSP. We first create a Java class to represent a threaded timer; this will be the environment in which our semantics will interact with. We then implement, using the ABACO Java framework, a yielder to return the current reading of the clock. We also implement actions to stop the clock. The semantics of our case study language is given with focus on integration with the previously created classes.

1 Introduction

Many languages, after released by the designers and used by programmers, were noted to have problems in their design; an example is Eiffel, an object-oriented programming language that preceded Java. After Eiffel was released and extensively used a bug was found in the language design. To avoid such problems, language developers have relied on formal language specification and tools to support them.

The study of the language semantic is a fundamental part of the language design. Formal techniques are used to give the semantics of the language. Tools

* Special thanks to Luis Carlos Menezes, for his kind help and guidance on the use of the Abaco system.

are used to support these formal techniques allowing developers to study and explore the desired and undesired behavior of the language.

Action semantics was developed by Peter Mosses and David Watt [5]. As denotational semantics, this formalism uses semantic functions to map programs into semantic entities. The action notation is a simple algebric form of expressing the language semantics. ABACO is a tool that is used to animate and study specifications written in Action Notation.

In a previous work [10], we illustrated the definition of an action semantics for a subset of Timed CSP. Because Timed CSP is a real-time specification language, and because ABACO does not support time properties, we developed an extension which made it possible to use ABACO to explore the time properties of the language. In this work, we concentrate on the extension developed of the tool and how it is used.

The rest of this document is organized as follows: The next section gives a brief description of action semantics. The following section describes the extension developed to add time facilities to ABACO. In Section 4, Timed CSPm is explored as a case study with the aim of illustrating the use of the extension; finally Section 5 lists our conclusions.

2 Action Semantics

Action semantics was developed by Peter Mosses and David Watt [5] in an effort to make semantic specifications more intelligible and, therefore, more widely accepted. Action semantics, as denotational semantics, uses semantic functions to map programs into semantic entities. There are three types of semantic entities used in action semantics:

- Actions: dynamic, compositional entities. The performance of an action directly represents information processing behavior and reflects the gradual stepwise nature of computation.
- Data: static entities of a specification, representing pieces of information. Action semantics offers a rich set of data types such as datum, integer, rational, set, list, and others; it also offers constructs to create new *sorts* of data.
- Yielders: an unevaluated item of data, whose value depends on the current information. An yielder may evaluate to the datum currently stored in a particular cell, which changes during the performance of an action. Another use of a yielder is to represent input from the external world; for example, the external system clock.

The notation used to specify the semantic functions and entities is called *Action Notation*. The information processed by an action may be classified according to how far it tends to be propagated:

- Transient: tuples of data, corresponding to intermediate results;
- Binding: tuples which associate tokens to data, corresponding to a symbol table;

– Storage: data stored in storage cells, corresponding to values assigned to variables;
– Permanent: data communicated between distributed actions.

The different kinds of information give rise to so-called facets of actions, focused on the processing of at most one kind of information at a time:

– *Basic facet*: processes independently of information (control flow). Examples of actions that make part of this facet are the terminating actions such as: complete, which indicates a normal termination, fail, which is used to indicate unsuccessful termination, and escape, which is used for exceptional termination. An action that terminates with escape can be treated using an action operator trap. It associates two actions and creates a new action which behaves as the first action, until it completes, terminates normally, or escapes. If the first action escapes, then the second action is executed. Other operators for choice and parallel execution are available.
– *Functional facet*: processes transient information. Consider the following action.

```
|give 5 and give 6
then
|give sum (the given integer # 1 , the given integer # 2)  .
```

This action composes two actions sequentially using the then operator: the transients produced by the first action are input to the second action. The give action returns the data produced by the argument yielder as transient. Therefore give 5 and give 6 produce transient values 5 and 6 respectively. The and is used to create a parallel composition of the two actions using interleaving. The composed action above produces the sum of the transients produced by each action. The yielder the given integer # 1 produces the first transient in the list of input transients.
– *Declarative facet*: processes scope information; for example, bind 10 to "x" is an action that produces a binding associating the value 10 to the token x. The yielder current bindings returns a mapped-set of all the current bindings.
– *Imperative facet*: processes storage information, including actions to reserve and unreserve storage cells, and to change the data stored in the cells. Consider the following example:

```
|allocate a cell
then
|bind the given cell to "x" and store 10 in the given cell .
```

The action allocate reserves a free cell of memory and gives this reserved cell as transient. In our example, after allocating the cell, the token x is bound to the cell, and the value 10 is stored in the cell.
– *Communicative facet*: processes permanent information, including actions to send messages, receive messages in buffers, and offer contracts to agents. Agent is a new concept added in this facet, which represents the action executing environment (machine or software). Communication is carried out by these agents.

A specification in action semantics uses a mixture of actions from all these facets, according to the need of the language designer. The designer can also produce new sets of actions and facets which are a combination of those above.

3 Extending ABACO with Time

The Algebraic Based Action COmpiler (ABACO) tool is used to study and animate specifications written in Action Notation. The use of a tool helps in the detection of problems related to the specification as it analysis various algebraic properties of the language. To make it possible to animate a real-time language, we we needed to add the notion of time to ABACO.

The ABACO tool was developed in Java with the objectives of portability, extensibility and reusability in mind. The tool offers a set of interfaces to add user code to the available API. Based on this fact, and using the interface we built an extension to add time facilities.

The extension consists of the addition of a yielder *current time* and an action *stop timer* to the current library of ABACO. The yielder returns a value representing the time passed since the start of the execution, while the action *stop timer* is used to terminate the timer. The timer itself was implemented as a singleton class and runs as a separate thread permitting the time to pass along with the execution of the actions which have no timing constraints.

3.1 The yielder

The ABACO tool accepts add-ons to be used from within a specification through the use of the `import` reserved word in the specification document.

```
import "TimedCSP.CurrentTime"
```

This directive instructs the tool to load the class `TimedCSP.CurrentTime` at compilation time. The class contains two methods `dependencies` and `register`. The first returns an array of strings indicating which environment objects need to be created before the extension can be executed. The tool uses this method to determine the order in which the extension is to be loaded and used within the tool.

```
public static String[] dependencies() {
    return new String[]
         { "br.ufpe.abaco.ActionInterpreter.Interpreter" };
}
```

In our case, only the `Interpreter` is need, so the compilation framework guarantees that this environment object is created and properly initialized before the actions and yielder of the extension are used.

The `register` method is used to include the evaluators associated to the extension. This is done by first obtaining the current `ActionInterrpreter` and

then inserting an evaluator for each extension. The insertion indicates the term used to identify the component. In the case of our example the value `current time` is used to identify the *current time* yielder and `stop timer` is used to identify the *stop timer* action. The insertion also requires the *Evaluator* object associated to the identifier, which is used to evaluate the term.

```
public static void register(CompiledSpecification espec) {
  Interpreter inte = (Interpreter)espec.getKey
                                        ("ActionInterpreter");

  inte.insertEvaluator("current time", new YielderEvaluator(){
    public Term evaluateYielder(Term yielder,
                                RuntimeAction runtime) {

      int t = TimedCSPTimer.getInstance().getCurrentTime();
      return new NumeralTerm(t);
    }
  }
  );

inte.insertEvaluator("stop timer", new ActionEvaluator() {
  public  RuntimeAction evaluateAction(Term yielder,
                                       RuntimeAction runtime) {
     TimedCSPTimer.stopTimer();
     return runtime;
   }
  }
  );
}
```

There are two main types of evaluators, `YielderEvaluator` which has a single method `evaluateYielder` that is responsible for the evaluation of the yielder and returns the term produced by the yielder `Term`. `ActionEvaluaters` has a similar method `evaluateAction` used to execute the action. The method receives the environment values as an instance of the class `RuntimeAction`. The methods `evaluateAction` also returns the modified environment in an instance of the same type. For further details please refer to the ABACO manual.

In our case the yielder `current time` gets the active instance of the timer and obtains the time counter; it then creates an integer `NumeralTerm` with the given value and terminates by returning this value. The action `stop timer` is passive; it calls the `stopTimer` method to stop the timer and then returns the same income it received from the system. This is to say that the action has no effect on the income and outcome; it deactivates the timer and returns the received parameter to the system.

3.2 The timer

The specification of Timed CSPm was developed based on a continuous model of time, but the ABACO tool in its current state does not support rational numbers. The ideal implementation, for the timer, is to obtain the system clock and convert it to a real value that is used later by the specification.

In the absence of rational numbers, however we decided to use the discrete model of time for the simulation. The semantics of the program is maintained, as all the time restrictions can be considered as integers.

The timer is implemented as a singleton thread, which is activated by the ABACO tool. As all singleton classes, the timer contains a static reference to itself, the constructor is private, and the static method `getInstance()` is used to obtain an instance of the class.

```
public final class TimedCSPTimer extends Thread {

    private int currentTime = 0;
    private static TimedCSPTimer ref;
    private final int DELAY=100;
    private static boolean active = false;

    private TimedCSPTimer() {
        super();
    }
    public static TimedCSPTimer getInstance() {
        if (ref== null){
            ref = new TimedCSPTimer();
            ref.active =true;
            ref.start();
        }
        if (! ref.active) {
            ref.currentTime=0;
            ref.active=true;
        }
        return ref;
    }
```

The timer operates in two states: either it is active and running, or it is idle (stopped). The timer state is indicated by a boolean variable `active`. When `active` is true, the timer is in a running state, otherwise the timer is idle. When the timer is active, it increments an internal variable `currentTime` at fixed intervals given by the variable `DELAY` in milliseconds.

The timer has two public methods: The method `getCurrentTime()` returns the value of the counter `currentTime`. The method `stopTimer()` is used to put the timer in an idle mode.

```
public int getCurrentTime() {
    return currentTime;
}
public static void stopTimer() {
    if(active) {
        ref.active=false;
        ref.interrupt();
    }
}
```

The main functionality of the timer is implemented in the method `run()`. This method is executed as soon as an instance of the class is created. The main body of the method is an infinite loop. The body of the loop starts by sleeping for a period of time determined by the variable `DELAY`, then it increments the counter by one.

```
public void run() {
    try{
        while(true) {
            try{
                while(true){
                    sleep(DELAY);
                    currentTime = currentTime+1;}
            }catch(InterruptedException e){ }
            // Wait for timer to be reactivated.
            while(!active){ sleep(DELAY);}
        }
    }catch(InterruptedException e){return;}
}
```

This operation implements the normal behavior of the timer. The normal behavior of the timer can be interrupted (see the method `stopTimer()`), and in this case the timer waits (idle state) until it is reactivated again by the user.

4 Case Study: Timed CSPm

To show the use of the extension described in the previous section, we study the semantics of a real-time specification language. It is not of our interest to show the complete specification of the language; this was explored in [10]. We study here only the aspects of the language related to time and illustrate how the tool is used.

Timed CSP adds time to CSP [2]. Like CSP, Timed CSP uses processes and events to define the behavior of a program. A process is a set of observations which define a pattern of behavior [7], and an event is an observation mark indicating the occurrence of an associated action or stimuli. The Timed CSP semantic model is influenced by the same properties of the untimed model of CSP [1]:

– *Maximum progress*: A program executes until it terminates or requires some external synchronization.
– *Maximum parallelism*: Each component of a parallel composition has sufficient resources for its execution.
– *Synchronous communication*: Each communication event requires the simultaneous participation of every program involved.
– *Instantaneous events*: Events have zero duration.

For a complete reference to the untimed CSP model see, [8].

In this paper we use a subset of Timed CSP. We singled out some simple operators, with a special interest in the time operators of Timed CSP. The new language has no communication channels, but has events; we omit the parallel composition and external choice, but include the interleave operator. Parallel composition and communication are considered in [3].

4.1 The Time Model

There are several time models for Timed CSP. We base our work on the Timed Trace Model $\mathcal{M}_{\mathcal{TT}}$ as presented by Davis and Schneider [1]. In this model, a process is represented by a set of timed traces. A timed trace is a sequence of timed events.

The original model has a time domain of non-negative real numbers. In our simulation, we use a positive integer time domain .

$$TIME = [0,\infty)$$

The set of all timed events is the cartesian product below, where Σ is the set of all the possible events of the observed process.

$$TE = TIME \times \Sigma$$

A timed trace is a finite sequence of timed events, such that events appear in a chronological order:

$$TT = \{ s \in Seq\ TE \mid \langle\ (t_1,a_1),(t_2,a_2)\rangle \preceq s \Rightarrow t_1 \leq t_2\ \}$$

where $s_1 \preceq s_2$, if and only if, s_1 is a subsequence of s_2.

4.2 Syntax and Informal Description

In this section we give a brief descrition of the syntax and informal semantics of Timed CSPm.

4.2.1 Specification

Syntax

(1) Specification = ⟦ ProcessDeclarations Process ⟧ .
(2) ProcessDeclarations = [[]] |
⟦ ProcessDeclaration ⟧ |
⟦ ProcessDeclarations ⟧ .
(3) ProcessDeclaration = ⟦ Identifier "=" Process ⟧ .

A specification is composed of two main parts: the first part is a possibly-empty sequence of process declarations; the second part is the specification of the main behavior. As an example, consider a one-shot printer described by the Timed CSP process below.

$$accept \rightarrow print \rightarrow SKIP$$

Initially, the printer is enabled to accept a job, after which it behaves as $print \rightarrow SKIP$. This subsequent process will be enabled to perform a *print* operation, and then behave as $SKIP$. This can be described in Timed CSPm as follows.

```
PRINTER0 = accept -> print -> SKIP
PRINTER0
```

The first line declares a process `PRINTER0` defined just as the Timed CSP process above. The second line defines the main process of the specification as `PRINTER0`; this is the execution point of the program.

4.2.2 Process

Syntax

- Process = "SKIP" | (1)
"STOP" | (2)
Identifier | (3)
⟦ Process "[]" Process ⟧ | (4)
⟦ Event "->" Process ⟧ | (5)
⟦ Process " | | |" Process ⟧ | (6)
⟦ Process ";" Process ⟧ | (7)
⟦ "Wait" Expression ⟧ | (8)
⟦ Process "[" Expression ">" Process ⟧ | (9)
⟦ Process "/" Expression "\ " Process ⟧ . (10)

A process can be a simple predefined process such as `SKIP` (1), which simply terminates normally. `STOP` (2) is another predefined process that suspends the execution of a process, but indicates a failure or an abnormal termination, after which the process cannot engage in any operation. This is known as a deadlock state. A process can also behave as a previously declared process, referenced using the process identifier (3).

The choice operator (4) is used to combine two processes to give a new process which behaves as either one or the other. Event prefix (5) was exemplified above: it combines an event, in which the process is prepared to engage, with a process that gives it behavior after the event. The interleaving composition operator (6) permits the execution of two processes in parallel without any synchronization. The sequential composition operator (7) is used to combine the behavior of two processes. The resulting process behaves initially as the first process until it terminates normally, and then it behaves as the second process. If any of the two processes fails, then the sequential composition also fails.

The `Wait t` process (8) is the first of the three operators involving time; it waits for a defined amount of time (defined by the parameter `t`) before terminating normally. The time out operator (9) is used to suspend a process, if it does not start within a predefined amount of time. Finally, the timed interrupt operator (10) is used to suspend the execution of a process after executing for a predefined amount of time.

4.3 Semantic Entities

In this section we introduce the semantic entities used is the specification.

Trace. This entity is defined to represent the timed traces of a process. A trace is the observable result of program animation. A `traceitem` is composed of a token, representing the event that occurred, and a time stamp associated to it, indicating the time in which the event occurred. Special tokens `TAU` and `TIC` are created to indicate an internal choice and a normal termination, respectively.

A `trace` is stored as a list of `traceitem`, therefore, the type `storable` is of type list. The function `t ( _ , _ )` is the constructor of the trace item. It associates a `toke` to a time stamp. The last line indicates that the function `t` can be used with yielder as parameters and it can be itself used as a yielder.

```
eq token >= string .
eq TIC : token  .
eq TAU : token  .
eq traceItem <= datum .
op t ( _ , _ ) :: token, time -> traceItem .
op t ( _ , _ ) :: yielder, yielder -> yielder .

eq storable = list .
```

Time. The `time` is defined to be of type `number`. The yielder `current time` is defined to be of type `yielder` and an action `stop timer` is also declared. The file containing the Java class described in the previous section is imported into the tool at this point. The Java code should contain the same names as the ones declared in tool.

```
eq time = number .
eq current time : yielder .
eq stop timer : action .
import "TimedCSP.CurrentTime" .
```

The definition of `Bindable` and `Values` can be found in the complete specification [10].

4.4 Semantic Functions

Here, we discuss the main technique used in the specification of the Timed CSPm semantics using actions. Action semantics uses actions to describe the program semantics. In our case, the main action associated to a specification is `run`, which takes a specification written in Timed CSPm and gives the action representing this specification.

```
op run _ :: Specification -> action .

re run ( ~d ~p ) =| elaborate ~d
                   before
                   | execute ~p .
re run ~p = execute ~p .
```

The action `run` depends on the form of the specification. If the specification is a list of process declarations `~d` followed by a process `~p`, the behavior associated to the specification is to `elaborate` the declaration part of the specification and then `execute` the process `~p`. If the specification is simply a process without any declarations, it is associated to an action which simply executes the process `~p`.

```
op elaborate _ :: ProcessDeclarations -> action .

re elaborate ( ~d1 ~d2 ) = elaborate ~d1 before elaborate ~d2 .
re elaborate ( ~i = ~p ) = recursively bind token of ~i to
                                        abstraction of (execute ~p) .
```

The declaration of a process creates an association between the body `~p` of the declaration and its declared identifier `~i`. This association is in the form of a binding from the identifier `~i` to an abstraction of the action associated to the execution of the process `~p`.

The next action to be specified is `execute`, which takes a process and gives an action representing the behavior of the given process. Here we explore only the implementation of the `SKIP`, `STOP` and `Wait` operations. The complete definition of the function and others can be found in [10]

```
op execute _ :: Process -> action .

re execute ( STOP ) = fail .
```

```
re execute ( SKIP ) =
     | init
     before
     | register t (TIC, the time bound to "init") .
```

The termination and failure of a Timed CSPm process is modeled with the help of action semantics basic actions. This can be noticed clearly in the function `execute` for the `Skip` and `Stop` processes. The `Stop` process is interpreted by the `fail` action that simple halts the execution of the program. The `Skip`, on the other hand, terminates normally, and control is passed to the next process or if there are no more processes it terminates. Before the process `Skip` terminates it registers an internal event `TIC` to record the termination of the process and the time in which it terminated.

```
eq init : action .
re init =
    | | give current time
    | then
    | | bind "init" to the given time
    before
    | | check bindings "trace"
    | then
    | | | complete
    | | else
    | | | | | allocate a cell
    | | | | then
    | | | | | | bind "trace" to the given cell
    | | | | | and
    | | | | | | store empty-list in the given cell .
```

The action `init` creates a binding associating the current time to the identifier `init`. It also checks if an identifier `trace` was previously created. If it was not created, then it associates it to the empty list. The identifier `trace` contains the timed trace of the process execution.

```
re execute ( Wait ~e ) =
     | init
     before
     | | | | evaluate ~e
     | | | and
     | | | | give the bindable bound to "init"
     | | then
     | | | bind "FinalTime" to ( sum (the given time # 1,
                                      the given time # 2) )
     | before
     | | unfolding
```

```
| | | give current time
| | then
| | | | give (the given time is greater than
                the time bound to "FinalTime" )
| | | then
| | | | | register t ( TIC , the time bound to "FinalTime" )
| | | | else
| | | | | unfold .
```

The `Wait` process waits for a given amount of time before terminating normally. This is done by evaluating the expression `~e` and then obtaining the `FinalTime`. The process then enters a loop which is executed while the current time is not greater than or equal to the time bound to `FinalTime`.

5 Conclusions

The use of action semantics makes the specification more readable and closer to the programmers way of reasoning. This simplifies the understanding of specifications written in the language, and helps less experienced users.

An important contribution of this work is the use of a tool, ABACO (Algebraic Based Action COmpiler) [4, 6], to aid in the elaboration of the specification, to test the correctness of the specification, and analysis, by simulation, the behavior of programs written in the specified language. ABACO has a well-defined framework and interfaces for the elaboration of extensions to the standard action notation. This facility of ABACO was used to implement the `current time` yielder, which allows animation of specifications and debugging the generated actions.

This work shows the use of the tool in the specification of the semantics of Timed CSPm, a subset of Timed CSP. In this case study, we concentrate on time aspects, leaving aside the communication and synchronization aspects semantically identical to CSP. The specification involves the elaboration of semantic functions, using action semantics to map the different components of the syntax to their corresponding behavior within the system execution. A new sort `Time` was added, and an associated yielder `current time` was defined to represent the system current time. Using these extensions, it was possible to describe the behavior of a Timed CSPm specification using action semantics.

A similar work was presented in [3], where the authors concentrate on describing the semantics of CSPm and, in particular, on the use of the communicative facet of the action notation to describe the communication between processes. The Label Transition System (LTS) is used as an implementation model to guide and define the behavior of the CSPm specification.

Our approach here is slightly different, as we use the action semantics logic and constructs to express the action behavior. This is possible because the communication and synchronization are not taken into consideration. A future work is to combine these two approaches to form a complete specification of the timed and untimed features of the language.

Another approach to the specification of an operational semantics for Timed CSP is proposed in [9]. In that approach, the authors express the semantics of a Timed CSP program in terms of two relations: an evolution relation, which describes when a process becomes another simply by allowing time to pass; and a timed transition relation, which describes when a process may become another by performing an action at a particular time.

References

1. J. Davies and S. Schneider. A brief history of timed csp. *Theoretical Computer Science*, 138(2):243–271, 1995.
2. C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21, 1978.
3. H. Moura L. Freitas, A. Cavalcanti. Animating CSPm using Action Semantics. *Proceedings of IV Workshop of Formal Methods*, 2001.
4. Luis Carlos Menezes, Hermano Moura, Mardoqueu Vieira, Wanderley Cansanção, Francisco Lima, and Leonardo Ribeiro. An action semantics integrated development environment. In Didier Parigot and Mark van den Brand, editors, *First Workshop on Language Descriptions, Tools and Applications*, volume 44. Elsevier Science, April 2001. Electronic Notes in Theoretical Computer Science.
5. Peter D. Mosses. *Action Semantics*, volume 26. Cambridge University Press, New York, NY, 1992.
6. Hermano Moura and Luis Carlos Menezes. The ABACO system - an algebraic based action compiler. In Armando Martín Haeberer, editor, *Seventh International Conference on Algebraic Methodology and Software Technology*, pages 527–529. Springer-Verlag, January 1999. Lecture Notes in Computer Science, volume 1548.
7. G. M. Reed and A. W. Roscoe. A timed model for communication sequential processes. In *Proceedings of ICALP '86*, volume 226. Lecture Notes in Computer Science, 1986.
8. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, 1998.
9. S. A. Schneider. An Operational Semantics for Timed CSP. In *Proceedings Chalmers Workshop on Concurrency, 1991*, pages 428–456. Report PMG-R63, Chalmers University of Technology and University of Göteborg, 1992.
10. A. Sherif, A. L. C. Cavalcanti, and H. Moura. An Action Semantics for Timed CSPm. In *6th Brazilian Symposium on Programming Languages*, pages 100–113, 2002.

AN2 Tools

Tijs van der Storm

University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
tvdstorm@science.uva.nl

Abstract. Action semantics has been designed as a framework for the formal yet readable definition of programming languages. The english-like syntax of Action Notation (AN) has indeed many pragmatic advantages over other formalisms especially when the definition of real-life programming languages is involved. Modularity of action semantics definitions increases scalability and reusability which in the software engineering community have been considered major benefits long since.
However, to put action semantics definitions to real use, one would like to have a way of executing semantic specifications or even compiling actions to some kind of object code. A number of such systems have been developed but most of them are based on AN1 which has been superceded by a new version: AN2.
We present a number of action interpreters and compilers based on the new proposed action notation. Our effort can be seen as a testcase for different approaches towards implementing actions. In addition to the benefits of executing actions, we concentrated on three desirable properties: portability, scalability and deployability. The tools presented all support the complete kernel of the proposed new Action Notation, including the interacting facet (with one exception). We will discuss the overall architecture of the tool suite and some implementation issues. We conclude with preliminary performance results.

Introduction

Action Semantics is a formal, readable and modular formalism for the definition of programming languages. As such it greatly improves the maintainability and reuse of language definitions. However, given such pragmatic advantages over other semantic formalisms, we would like to be able to *use* the languages defined using action semantics. In this paper we explore strategies for executing actions. These strategies comprise: execution by term rewriting, compilation to Java and C, and using an intermediate language. We have focussed on the following issues:

Generality The tools to execute actions should support the full kernel of AN2. This includes the interacting facet as well as reflection.

Self-containment Compiled actions should be a self-contained black box which can be subject to reflection (as provided by AN2).

Deployability It should be an easy job to embed actions, either interpreted or compiled, into existing software environments.

Extendibility Users should be able to extend interpreters or specialize compiled actions to their needs. Furthermore, the specifications of the compilers and interpreters themselves should be maintainable enough to allow (future) extensions or changes.

Portability There should be no restriction as to the platforms that are supported by the interpreters and compilers.

Efficiency Although not a primary goal, the performance of executing actions should be reasonable.

From these desiderata one can deduce that we deem the software engineering aspects of executing actions most important. This is a natural consequence of our view that action semantics is not only a formalism to define the mathematical semantics of programming languages, but also a language to define domain specific languages.

Organization First we give a very short introduction to the new Action Notation and discuss some distinctive features. In Section 2 we describe the term rewriting strategy which is divided in two parts. First we use the ASF+SDF Meta-Environment to derive a term rewriting system from the Modular SOS definition of the kernel of AN2 (AN-K). The second part discusses the pros and cons of reimplementing this term rewriting system by hand in C. In the next section we discuss a way to compile actions to C and Java. Section 4 describes the intermediate language approach. We conclude with assessment of the strategies presented, and a discussion of related work.

Acknowledgements Paul Klint provided useful hints during the preparation of this paper.

1 Action Semantics for Dummies

In this section we give a short introduction to the new Action Notation [10, 13].

The central concept of action semantics is the action. An action is a computational entity that takes tuples of data and gives tuples of data. Actions can be primitive (e.g. computing a data operation, updating a cell) or combined using combinators that capture the various flows of data and control. For example: the **then** combinator is used for functional composition. This means that in action A_1 **then** A_2 the data given by A_1 is passed as input to A_2. Other combinators exist for sequential composition, interleaved composition, non-deterministic choice etc. Actions are able to terminate in three ways: normal, exceptional or failing. Normal and exceptional termination is accompanied by a data value (*given* resp. *raised* data). Combinators such as **exceptionally** and **otherwise** can be used to trap non-normal termination. For example: the action *raise* **exceptionally** *provide* () will terminate normally and give the empty tuple as a result.

Action Semantics is divided over a number of so called *facets* which capture different ways of information processing. For instance, the functional facet consists of all actions having to do with information flow without side effects. Side

```
give current bindings and (provide ("unf", copy and provide 0 then (check _=_
exceptionally fail) then provide 1 otherwise (copy and provide 1 then (check _=_
exceptionally fail) then provide 1) otherwise (copy and provide 2 then give _-_ then
(give the data_ then give provide_ and (give current bindings then give provide_
and (give current bindings and provide "unf" then give bound_ then give the action
[taking () giving bindings]_) then give _hence_) then give _then_ then enact) and
(copy and provide 1 then give _-_ then (give the data_ then give provide_ and (give
current bindings then give provide_ and (give current bindings and provide "unf"
then give bound_ then give the action [taking () giving bindings]_) then give
_hence_) then give _then_ then enact)) then give _+_)) then give binding_) then
give overriding_ hence (copy and provide 0 then (check _=_ exceptionally fail)
then provide 1 otherwise (copy and provide 1 then (check _=_ exceptionally fail)
then provide 1) otherwise (copy and provide 2 then give _-_ then (give the data_
then give provide_ and (give current bindings then give provide_ and (give current
bindings and provide "unf" then give bound_ then give the action [taking () giving
bindings]_) then give _hence_) then give _then_ then enact) and (copy and provide 1
then give _-_ then (give the data_ then give provide_ and (give current bindings then
give provide_ and (give current bindings and provide "unf" then give bound_ then
give the action [taking () giving bindings]_) then give _hence_) then give _then_
then enact)) then give _+_))
```

Fig. 1. Recursive Fibonacci in AN-K

effects are covered in the imperative facet. One of the distinctive differences between the AN1 and AN2 is that for the latter the facet responsible for the flow of bindings is included in the functional facet. That is, bindings have become a subsort of Datum (the sort of individual values) and can be processed as such by actions. There is only one basic combinator that deals with bindings: **hence**. Consider the action A_1 **hence** A_2. If A_1 terminates normally with bindings as a result, these bindings become available in A_2. The primitive action *give current bindings* returns the current set of bindings as data. It is then possible to obtain bound values using specific data operations.

Although facets were present in AN1, AN2 additionally distinguishes two levels of notation: Full AN2 and Kernel AN2 (AN-K). The semantics of Full AN2 is defined in terms of AN-K. As a consequence, AN-K is the only level tool support has to deal with to obtain full generality. Many familiar constructs from AN1 are now defined in terms of AN-K actions using reflection. Reflection in this case amounts to the dynamic *construction* of actions using combinators, as opposed to *inspection* of actions (like reflection in Java). The concept is simple, yet very powerful. The idea is that the sort of actions is subsort of Datum and consequently all action combinators are data operations. Consider for example the following Full AN2 action which computes the Fibonacci number for a given integer:

```
unfolding(
  (copy and provide 0 then (check _=_ exceptionally fail) then provide 1)
  otherwise
  (copy and provide 1 then (check _=_ exceptionally fail) then provide 1)
  otherwise
  ((copy and provide 2 then give _-_ then unfold) and
   (copy and provide 1 then give _-_ then unfold) then give _+_)
)
```

The same action reduced to AN-K is displayed in Figure 1. Note how the behaviour of **unfolding** and *unfold* is mimicked by binding the special token "unf" and employing reflection to inline the unfolded body.

We will use the AN-K action displayed in Figure 1 to assess the runtime performance of the tools presented here as a running example. Furthermore, it gives a good impression of what kind of input we are dealing with here. As we require full support of the kernel of AN2, all tools presented here are able to execute this action in its kernel form without any knowledge of the higher level constructs behind it.

2 Execution by Term Rewriting

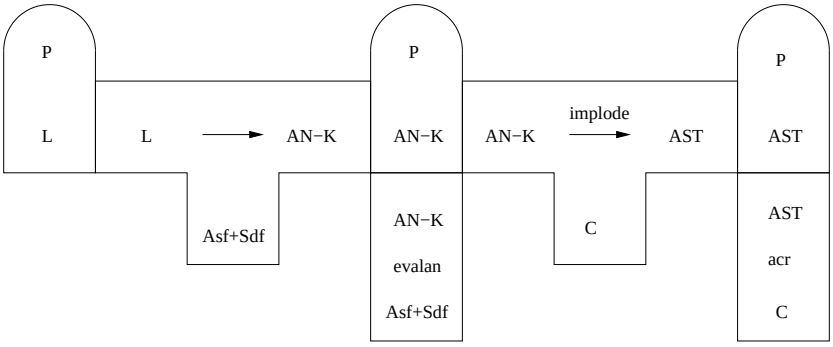

Fig. 2. Interaction of term rewriting tools

In this section we will review two term rewriting approaches to the problem of executing actions. In the first approach we used the ASF+SDF-formalism [9, 3] to specify the semantics for the kernel using conditional equations which are very close to the Modular SOS transitions of AN2 [13]. An interpreter is obtained by viewing these equations as rewrite rules. The second approach is a reimplementation in C of the derived term rewriting system.

In Figure 2 the interactions between the various translators and interpreters are displayed using T-diagrams [6]. The primary input is a program P written in language L. This program is translated to a kernel action by some specification in ASF+SDF. The resulting kernel action can be executed directly by `evalan`, the interpreter implemented using ASF+SDF. Secondly, an implosion step converts an AN-K parse tree to an abstract syntax tree (AST)[1]. The resulting AST can then be executed by the action rewriter `acr` which is implemented in C.

2.1 Kernel AN2 in Asf+Sdf

Since the semantics of Kernel AN2 is defined operationally, ASF+SDF is a useful tool for specifying it. The marriage of SDF to ASF enables one to write condi-

[1] This step is provided for by a tool accompanying ASF+SDF: `implodePT`. Since this tool is provided as is, the implementation language (C) is in fact irrelevant.

tional equations on terms using concrete syntax. Directing these equations gives rise to a (derived) term rewriting system, which can be compiled to an efficient standalone tool. The interpreter `evalan` is such a specification. The specification of `evalan` consists of roughly forty five modules defining both syntax and semantics of data and actions. To actually see how the Modular SOS transition relations are translated to ASF+SDF, let's compare one of the relations defining the **then** combinator in CASL and ASF+SDF. If the left hand operand to **then** has terminated normally, the following transition rule applies:

$$\frac{\alpha' = \alpha[d_1/data] \wedge A_2 \xrightarrow{\alpha'} A_2'}{\mathit{normal}\ d_1\ \mathbf{then}\ A_2 \xrightarrow{\alpha} \mathit{normal}\ d_1\ \mathbf{then}\ A_2'}$$

The transition declares that d_1 is known during the one-step execution of A_2 in the *data* field of label α' . In ASF+SDF this transition is defined in one (conditional) equation:

$$\frac{\begin{array}{r}\downarrow A_2 = \mathit{false},\\ \alpha.\mathit{data} := d_1 = \alpha',\\ [\![A_2, \alpha']\!] = \langle A_2', \alpha''\rangle\end{array}}{[\![\mathit{normal}\ d_1\ \mathbf{then}\ A_2, \alpha]\!] = \langle \mathit{normal}\ d_1\ \mathbf{then}\ A_2', \alpha''\rangle}$$

Since ASF+SDF has no notion of truth *and* we want our specification to be executable, assertions are replaced with equations that faithfully represent the intended semantics of the CASL specification. This involves for example the termination check $\downarrow A$ to ensure that this equation does not fire for *normal* d_1 **then** *normal* d_2. The concept of a Modular SOS label is captured by an environment passed throughout the evaluation of $[\![\ ]\!]$. The equation assigns a new environment updated with d_1 in the *data* field of α which is passed to the one step execution of A_2 resulting in a residual and a (possibly modified) environment. The result of the equation is a tuple of the original action with A_2' substituted for A_2 and the new environment.

To enforce the composition constraints defined on labels, we explicitly ensure that no local updates to labels are passed through multiple steps in the definition of the transitive closure of $[\![\ ]\!]$. Thus, the equations for $[\![\ ]\!]^+$ compute a valid environment based on the incoming environment and the environment resulting from the computed step using \$ (which is similar to \$ defined in [13]).

$$\frac{\begin{array}{c}\downarrow A = \mathit{false}\\ [\![A, \alpha]\!] = \langle A', \alpha'\rangle\end{array}}{[\![A, \alpha]\!]^+ = [\![A', \alpha\$\alpha']\!]^+} \qquad \frac{[\![A, \alpha]\!] = \langle T, \alpha'\rangle}{[\![A, \alpha]\!]^+ = \langle T, \alpha\$\alpha'\rangle}$$

The interpretation function to perform an action A with input d is defined as follows:

$$\mathrm{perform}(A, d) = [\![\mathit{normal}\ d\ \mathbf{then}\ (\mathit{normal}\ \mathit{no\ bindings}\ \mathbf{hence}\ A), \varepsilon]\!]^+$$

In this equation ε denotes the empty environment.

2.2 Action Rewriting in C

Although ASF+SDF provides a useful framework for specifying the semantics of Kernel AN there are some drawbacks. Since term rewriting is the only computational device ASF+SDF (currently) supports, even the most primitive operations are to be specified using equations. For instance addition of two integers is computed in a term rewriting fashion. Of course we would like to use native support to compute integer operations to speed up execution. The same holds for updating of stores, rotating the schedule and so on. Furthermore, ASF+SDF's capabilities seemed far too general for our purposes: since signatures are described by production rules in SDF we had to devise *concrete* syntax for all auxiliary structures such as stores, bindings, finite maps etc. Disambiguation of all these sorts—using syntactic sugar—hindered a perspicuous implementation even more. Finally, we thought we could improve upon the performance of `evalan`. These considerations taken together, led to the decision to reimplement the term rewriting system by hand in C. Having a valuable prototype in ASF+SDF and the ATerm library [16] to implement a data notation, this was not a hard job.

`Acr` takes an Abstract Syntax Tree as input. The algorithm traverses the tree in a Modular SOS fashion: the traversal is directed by combinators, until a subtree can be collapsed. The tree thus decreases in depth in a bottom up fashion, while traversing it top down for each step. The main difference between `acr` and `evalan` is that `acr` employs term rewriting only to reduce combined actions. All primitive actions (including data operations and predicates) are hardcoded in C. Especially for interacting actions the performance gain is expected to be more than marginal since no matching is involved in rotating the schedule (and this is likely to occur very often).

2.3 Comparing `evalan` and `acr`

Since both `acr` and `evalan` traverse the action tree for each computational step, the complexity of the reduction algorithm should be roughly the same as for `evalan`. However, primitive actions such as updating a cell have complexity $O(1)$ in `acr`. The complexity of primitive actions in `evalan` depends on the data structures involved. For example, updating a cell in `evalan` will cost $O(n)$ in the worst case, where n is the number of allocated cells. This is due to the fact that finite maps in ASF+SDF are essentially lists of tuples. To assess the influence of these primitive actions more concretely, we compared the performance the of the Fibonacci action presented earlier and an iterative version. The results show that `acr` is on average two times as fast as `evalan` for the recursive algorithm. Since no imperative actions are used and bindings are implemented using bounded balanced trees [1] both in `acr` and `evalan` this can only be accounted for by the arithmetic operations and the more complex matching of terms in `evalan`. For the iterative Fibonacci algorithm we see that for larger values of n (≈ 100) `evalan` takes considerably more time than `acr` and this difference is growing fast. Again term rewriting of arithmetic is probably the cause of this.

Remark The fact that `acr` traverses the tree and constructs new ones on the way up for each step, could have been avoided by using a different tree representation. (Abstract) Syntax trees resulting from ASF+SDF related components are always represented by ATerms which allow no destructive updates (copy-on-write semantics). Using a tree representation that would allow destructive updates on subterms, an executing agent could have been represented by a *cursor* walking over the tree, collapsing and creating trees only locally. However, there is one 'small' problem to this approach: syntax trees would need an enormous amount of memory compared to the corresponding ATerm, since ATerms are maximally shared. Precisely this problem of explosion in size has been one of the reasons for making ATerms maximally shared [16].

Assessment While experiments show that `acr` is generally faster than `evalan`, there are a number of drawbacks to the handcoded approach. First, `acr` uses a fixed length representation for integers. Thus, integer values are restricted to the size native to the machine `acr` is running on. A related problem is that the used data notation of `acr` is hard to extend by the user. For `evalan` one has the full algebraic power of ASF+SDF available to extend the interpreter with arbitrary data types. This is achieved using an interface mechanism. If a user specifies his own data constructors and data operations one can extend the interpreter by complying to this interface and adding his equations to the equations of the `evalan` specification. Using the interface the interpreter can "know" about foreign data constructs. The meaning of the data operations is specified using the generic `result` function which is used by `evalan` to execute **give** *o* actions. Depending on the importance of full generality and extendibility it might not be such a good idea at all to reimplement the term rewriting system by hand. Moreover, `acr` has been designed to primarily optimize non-functional aspects such as scheduling and store updates, so for purely functional actions the performance penalty induced by `evalan` is expected to be relatively small and constant (as is corroborated by the comparison of recursive Fibonacci).

3 Compilation to Java and C

In this section we describe ways to compile actions to Java and C. In Figure 3 the compilers `acc` and `ajc` are depicted using T-diagrams. Both compilers are implemented using ASF+SDF. As before, we only consider kernel notation, with the exception of unfoldings which are detected by `ajc`. For the compilation to C we restrict ourselves to single threaded actions. We require our compiled actions to be self-contained and compositional. Self-containment allows for easy deployment of actions, while compositionality ensures the possibility of (off line) reflection. To achieve these requirements, we introduce Action Functors in C and Enactables in Java. Both are interfaces (signatures) to which compiled actions should comply. Since we require that reflection is supported, this opens the way to separate compilation of actions (even from different source languages) and then combining them into one. Furthermore, in Java, an action implementing the

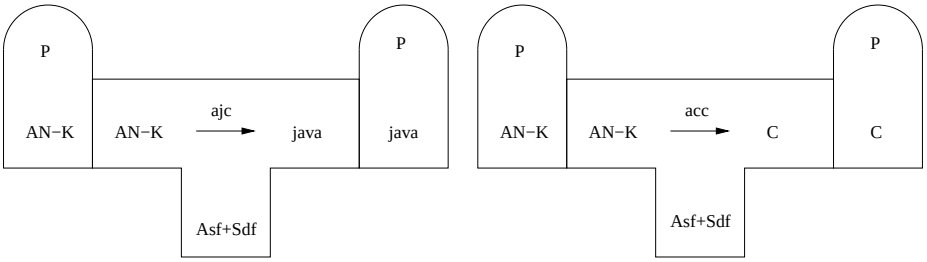

Fig. 3. Compilation of AN-K to Java resp. C

Enactable interface can also implement the standard Serializable interface, which makes it possible to store actions on file or send it over a network connection.

First we describe `acc` the action to C compiler. We then compare this compiler to `ajc`, the java compiler.

3.1 Action Functors

An Action Functor in C is a type definition describing a function that represents an action, that is, a function that transforms data and bindings into data, while perhaps referencing cells. Its definition in C is as follows:

typedef AN_Data (∗ACCFunctor)(AN_Data,AN_Data);

The `acc` runtime library defines all Kernel AN2 primitive actions in such a way that they obey this function type. The compiler translates an action tree by mapping each subtree to an Action Functor. So for example A_{i0} then A_{i1} at position i in the tree is compiled to:

AN_Data action$_i$(AN_Data data, AN_Data bindings) {
  register AN_Data temp = action$_{i0}$(data, bindings);
  return action$_{i1}$(temp, bindings);
}

One would expect that this way of compiling actions to C would result in inefficient code, since the number of function calls is equal to the number of nodes in the action tree. However, we have experimented with different compilation schemes (such as using intermediate variables in one big function, or exploiting the runtime stack) but they, suprisingly, all turned out to be slower than the compilation scheme presented here. This is probably due to sophisticated optimizations of GCC.

To cope with exception handling and (non deterministic) choice we use a choice point library which allows non-local jumps at a very high level [11]. To illustrate this, the code for A_{i0} exceptionally A_{i1} looks like this:

```
AN_Data action_i(AN_Data data, AN_Data bindings) {
  if (!ACC_try())
    return action_i0(data, bindings);
  else {
    register AN_Data temp = ACC_catch_exception();
    return action_i1(temp, bindings);
  }
}
```

Here ACC_try returns 0 when setting the choice point and returns 1 when $action_{i0}$ raised an exception or failure. In the **else** branch ACC_catch_exception returns the raised data in case of an exception and rethrows a failure otherwise. When an exception or failure occurs all registers and local variables are restored.

Now, the hard part is, of course, reflection. To accomplish *real* reflection in a portable way, we employed Paolo Bonzini's GNU Lightning. Lightning has been designed to implement fast just-in-time compilers, and has been used as such in GNU Smalltalk. It provides a set of macros that define a generic assembly language. These macros allow a programmer to build ordinary C functions at runtime for a number of platforms (i386, Sparc and PowerPC). Data operations defined on action operands are implemented by this runtime assembler, specialized for Action Functors. Since any action may be operand to, e.g. _**then**_ we have to ensure that all actions present in the runtime have the function type defined earlier. Currently, `acc` does not yet detect unfoldings to prevent reflection at runtime, but a memotable prohibits the construction of an action for each pass through a loop.

3.2 Enactables

Enactables are the Java equivalent of Action Functors. Enactables are classes that implement the Enactable interface. This interface declares one method `enact`, accepting Data and Bindings and returning Data. Enactables can be embedded in Action classes which are subject to reflection. The compilation to Java proceeds much in the same way as for `acc`, except that `ajc` generates a class implementing the Enactable interface for the top action, and private methods for each subaction. Provided actions are compiled to an inner class implementing the Enactable interface. The action data operations receive Action instances that are constructed from Enactables. This way it is easy to combine compiled actions with hand written java classes. Another difference is the implementation of the data notation. Since the subclass concept resembles the subsort relation in AN2, implementation of the basic data types was straightforward. Runtime type checks are performed using standard casting operators of Java. The Data Notation is implemented using the Factory design pattern [8] to allow future changes[2] to the representation of values.

[2] For example, an implementation based on ATerms would allow communication between actions compiled to Java and actions compiled to C.

3.3 Comparing `acc` and `ajc`

Since `acc` and `ajc` compile actions almost in the same way, the comparison in performance between C-compiled actions and Java-compiled actions does not say us much about the compiler, but more about the difference between C and Java as a target. Generally speaking, the recursive fibonacci action compiled to C executes approximately 2.5 times faster than the same action compiled to Java. For the iterative version this factor is close to 5.

Assessment It is obvious that actions compiled to C are more efficient than actions compiled to Java. However, the actions in Java have a number of important pragmatic advantages. It is our view that these advantages outweigh the performance penalty by far. This is motivated by the following observations:

- First of all the slogan "compile once, run anywhere" applies here.
- The use of the Enactable interface allows the combination of compiled actions with *arbitrary* Java code. This is also possible for actions compiled to C but the process is more tricky and less type safe.
- For extension or specialization of compiled actions one has the complete Java runtime library at one's disposal.
- Java classes are mobile. E.g. sending compiled actions over a network connection using serialization should pose no problem.
- Object Orientation alleviates the burden of changing and/or maintaining the `ajc` runtime library. Since the java runtime library contains a host of standard data structures the supported data notation can be extended uniformly.

4 Action Intermediate Language

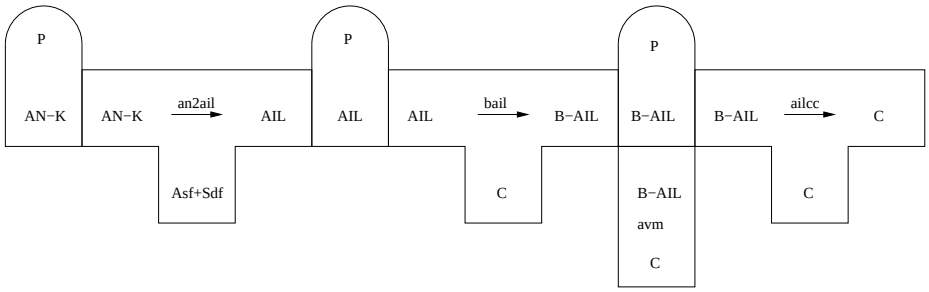

Fig. 4. Compilation of Actions to AIL

It is evident that the way action trees are reduced in the term rewriting paradigm is a source of inefficiency. The compilation to C remedies most of this, but at

the cost of full generality: multi threading is not supported. Another drawback is that compiled actions are not mobile in a dynamic way. Combination of two compiled actions always involves writing a glue function which uses the reflection operations of the `acc` runtime and (re)compiling the result. In this section we discuss the intermediate language approach to compilation of actions. Using Action Intermediate Language (AIL) we obtain full generality and performance comparable to that of actions compiled to C.

The interaction of the tools involved in the compilation of actions to AIL is depicted in Figure 4. We assume that P is available as a Kernel AN2 action. This action is first translated to an AIL parse tree (`an2ail`) which is then converted to a binary representation by the independent tool `bail`. The resulting object can be executed by the Action Virtual Machine (`avm`) or compiled to C.

Speed is achieved by translating the action tree to a sequence of instructions. The advantages are obvious. For example, in case of an exception, we can now jump to the appropriate handler instead of stepwise proliferating the exception up te action tree. One execution step now corresponds to incrementing an instruction pointer, instead of traversing a very large tree. Action Intermediate Language (AIL) has primarily been designed to remedy the efficiency problems of the term rewriting approach. Of course, since a number of SOS steps are collapsed into one jump, the behaviour of multi threaded actions can significantly differ. We expect, however, that it is possible to achieve correct operational behaviour modulo exceptional/alternative flow for multi threaded actions by inserting appropriate yield points in AIL byte code.

Action Intermediate Language can be seen as a typed assembly language of the stack based kind. AIL instructions can accept one (optional) parameter. If a parameter is present it must be an integer (indicating a non-datum constant value), a label, an ATerm (representing a datum or a data sort) or AIL code itself. The last argument type allows for reflection. Labels are translated to offsets by the bytecode compiler to ensure compositionality for AIL bytecode. Again, we use ATerms for the representation of data. One of the reasons to use ATerms for AIL bytecode lies in the efficient IO capabilities of the ATerm library. ATerms can be written to file *while retaining maximal sharing*. So we get serialization of all data (including actions provided as data) for free. The standalone tool `bail` takes an AIL parsetree as input and then builds an array of bytes, mapping instructions to opcodes and serializing data to binary ATerm format within the stream. The resulting byte stream is converted to a BLOB (Binary Large OBject) ATerm and written on file.

A part of the instruction set of AIL is displayed in Table 1. Data operations, predicates and primitive instructions more or less correspond directly to Kernel AN2 primitive actions and are not listed in the table. A difference is that typed primitive instructions do not check their arguments for type correctness. This is dealt with by the special instruction `cast`. The data flow instructions all operate on a number of stacks the function of which is explained in the next section. Control flow instructions are able to change the instruction pointer. The instruction `goto` l just does this. If a frame needs to be allocated, the

Aspect	Instructions
Given Data Flow	`prov, push, drop, copy, merge`
Raised Data Flow	`eprov, epush, edrop, ecopy, emerge`
Argument Data Flow	`publish, unpublish, epublish`
Scope Data Flow	`enter, leave, scope`
Normal Control Flow	`frame, goto, return, enact`
Escaping Control Flow	`trye, tryf, raise, throw, fail, catch`

Table 1. Subset of AIL Instructions

instruction `frame` can be used. The label argument to this instruction denotes a return address. The instructions `trye` and `tryf` install a handler at the program point specified by the argument label.

4.1 AIL Runtime Environment

The runtime environment for AIL programs consists of a store, a schedule and a code region. When a program is run, an AIL Control Block (ACB) is created for agent "main". ACBs contain all necessary data structures that are local to a thread (or agent). At runtime, new ACBs may be added to the schedule by the `activate` instruction. ACBs consist of two registers and a number of stacks. The first register is used for normal data flow (nreg) and the second for exceptional data flow (ereg). Each register has an associated stack (nresult resp. eresult) that is used for saving computed values. All primitive instructions *not* merely concerned with flow of data, take arguments from the registers and return values in the registers. If a sequence of instructions needs the same argument, the register is first saved on the argument stack and copied back into the register when needed. The third kind of stack is used for binding flow: the scope stack. Bindings can be transferred to and from the normal flow component. Finally, ACBs contain a context stack and a frame stack. The context stack is used for exception and failure handling. Contexts contain snapshots (shallow copies) of the result stacks, argument stack and scope stack, used for restoring the environment after an exception has occurred. Furthermore, a context contains a reference to the frame in which it was saved and a continuation address. The frame stack is used to save the return address when AIL code is enacted. To elucidate the way data may flow at runtime, the basic data flow instructions are depicted in Figure 5.

4.2 Mapping AN-K to AIL

Let's look at how the basic action combinators are translated to AIL. The normal data flow and binding flow translations are rather obvious. The following four sequences represent the **then**, **and then**, and **hence** combinators (`[[A]]` denotes the translation of a sub action of a combinator).

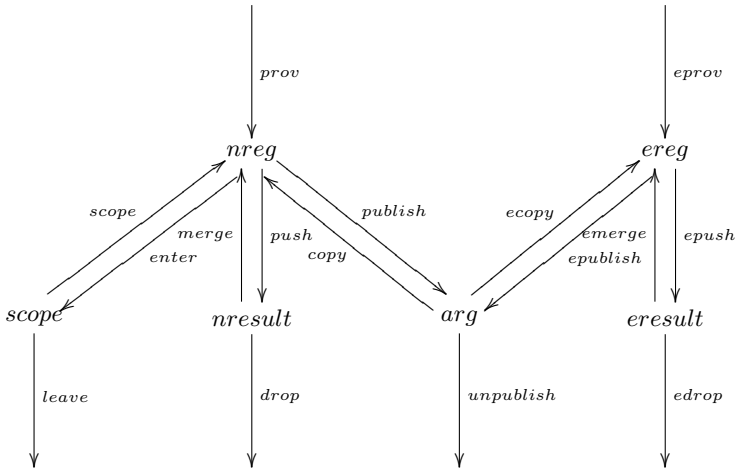

Fig. 5. Transfer graph relating stacks and registers

```
[[A1]]
publish;
[[A2]]
unpublish;
```

```
[[A1]]
push;
copy;
[[A2]]
merge;
```

```
[[A1]]
cast bindings(<term>);
enter;
copy;
[[A2]]
leave;
```

In the first sequence A1 executes and leaves its result in nreg. This result is published onto the argument stack. Then A2 executes, probably using the published value. Finally the published data is withdrawn. In the second sequence, after A1 has finished the value in nreg is pushed onto the result stack. Then the published value (incoming data) is copied from the argument stack back into nreg. So, when A2 executes it receives the same input as A1. Finally the `merge` instruction prepends the top of the result stack to the data in nreg. The third sequence involves scoping. Since hence is the only typed combinator we have to ensure that the data given by A1 is of the proper type. The `cast` instruction leaves nreg as it is when the data is of the argument type, i.e. bindings, and throws an exception otherwise. If A1 returned bindings they are pushed onto the scope stack, making them the current set of bindings. Then the published data is copied and after A2 has finished, the current scope is left.

To see how exceptional control flow is mimicked in AIL, the translation of the **and exceptionally** combinator is an interesting case. An action A_1 **and exceptionally** A_2 terminates exceptionally if both subactions terminate exceptionally. This exception is accompanied by the concatenation of the raised data values of the subactions. In AIL this is achieved by guarding the righthand ac-

tion with a `trye` block and throwing an exception with concatenated data in the handling part. Thus, A_1 **and exceptionally** A_2 is mapped to:

```
      trye l1;      // install handler at l1
        [[A1]]
      catch l2;     // branch to l2
l1:   epush;        // push raised data onto eresult
      trye l3;      // install another handler
        [[A2]]
      catch l4;
l3:     emerge;     // merge raised data
        throw;      // throw exception
l4:   edrop;        // pop datum of first exception
l2:   ...
```

Recall that `emerge` prepends the top of eresult to the data in ereg (the data raised by A_2). Note also that the `catch` instruction just pops the context stack and branches to the argument label.

The output of `an2ail` for the fibonacci action presented earlier is displayed in Figure 6. Note that no reflection is present since **unfolding** and *unfold* are detected by the compiler.

4.3 Simple Optimization of AIL

The reason to separate the type checking from the actual computation of data operations is that if the type is statically known, the `cast` instruction can be left out. The use of registers in combination with stacks allows a number of simple optimizations of AIL code that reduce the number of stack operations considerably.

$$\begin{aligned}
\texttt{publish; } i\texttt{; unpublish;} &= i\texttt{;}\\
\texttt{update; push; copy; } i\texttt{; merge;} &= \texttt{update; copy; } i\texttt{;}\\
\texttt{copy; copy;} &= \texttt{copy;}\\
\texttt{copy; prov } term\texttt{;} &= \texttt{prov } term\texttt{;}
\end{aligned}$$

The first rule states that if only instruction i uses the published data (which still resides in nreg after `publish`), the data need not be pushed onto the argument stack at all. In the second equation, the merging of data can be left out, since `update` (if terminating normally) returns the empty tuple. The third equation is obvious. Finally, for an instruction that never uses the data given to it, the published data does not have to be copied into nreg.

4.4 Assessing `avm`

Experiments with the recursive fibonacci action show that interpreting bytecode is slightly slower than the action compiled to C but this difference is growing for

```
{
            frame @l2;
      @l1:
            tryf @l11;
            tryf @l101;
            copy;
            push;
            prov int(0);
            merge;
            publish;
            trye @l100002;
            cast [⟨appl(⟨term⟩)⟩,⟨appl(⟨term⟩)⟩];
            eq;
            catch @l100003;
@l100002:
            fail;
@l100003:
            unpublish;
            prov int(1);
            catch @l102;
    @l101:
            copy;
            push;
            prov int(1);
            merge;
            publish;
            trye @l100102;
            cast [⟨appl(⟨term⟩)⟩,⟨appl(⟨term⟩)⟩];
            eq;
            catch @l100103;
@l100102:
            fail;
@l100103:
            unpublish;
            prov int(1);
    @l102:
            catch @l12;
     @l11:
            copy;
            push;
            prov int(2);
            merge;
            publish;
            cast [int(⟨term⟩),int(⟨term⟩)];
            sub;
            unpublish;
            publish;
            frame @l101001;
            goto @l1;
@l101001:
            unpublish;
            push;
            copy;
            push;
            prov int(1);
            merge;
            publish;
            cast [int(⟨term⟩),int(⟨term⟩)];
            sub;
            unpublish;
            publish;
            frame @l101011;
            goto @l1;
@l101011:
            unpublish;
            merge;
            publish;
            cast [int(⟨term⟩),int(⟨term⟩)];
            add;
            unpublish;
     @l12:
            return;
      @l2:
            return;
}
```

Fig. 6. Recursive Fibonacci in AIL

large values of n. We have not yet tested the additional compilation to C, since this is not fully operational yet.

The pragmatic advantages of the intermediate language approach are obvious. Actions are compiled to a single object which is mobile, compositional and self contained. Offline combination of different actions is straightforward using the reflection primitives of AIL;—the result is just another binary object file. This is an improvement with respect to actions compiled to C where one has to recompile the separate compiled actions if one wants offline combination. Another pragmatic advantage is the extendibility of `avm`. Although this cannot be done dynamicly or on request, `avm` is designed in such a way that extension consists only of adding instructions to the instruction set and defining their semantics in C. No changes to `bail` or `ailcc` have to be made.

5 Conclusions and Discussion

In this final section we compare our work to previously conducted research in this area and present some conclusions.

5.1 Discussion

The generation of interpreters and compilers from action semantic descriptions has a long history (cf. [4, 17, 15]). For the new Action Notation however, this field of research has only just begun. In this paper we have presented some strategies for interpreting and compiling actions in ways that have not been explored before. Our approach differs in a number of aspects from the previously conducted research in this field.

The first and foremost difference is the central role of the algebraic specification formalism ASF+SDF. The Action Semantic Description (ASD) tools [17] used ASF+SDF to generate term rewriting systems from an ASD, a formalism on its on. That is, the ASD tools operated by first parsing an ASD and then *generating* an number ASF+SDF modules which could be used to check and execute the language defined. In our approach, however, ASF+SDF *itself* is used as action semantic description formalism. Syntax is defined in SDF, which is then rewritten to action terms by the ASF component. This approach has a number of important advantages. First, since SDF is a declarative formalism that allows the full class of context-free grammars, the syntax of a language is easily specified and need not be molded into the class of LR or LALR grammars. This has the additional advantage that grammars can be designed in a modular way, since only full context-free grammars are closed under union. Furthermore, the compositionality of action notation and the algebraic defined data notation make ASF a perfect formalism for mapping syntax to semantics. This can be done using concrete syntax which makes action semantic descriptions all the more readable. Finally, a specification can easily be used outside the Meta-Environment by reusing the standalone components of ASF+SDF. In this paper we have described an interpreter of actions which is independent of the language defined: the interpreter is only defined for Kernel AN2. The composition of a language specification and the interpreter yields an interpreter for the language defined.

Previous efforts to execute actions were primarily focussed on performance issues relative to hand written compilers and interpreters. The execution speed of, e.g., OASIS [15], depends largely on thorough analysis of actions, involving for example binding time analysis and on restricting the set of action combinators and primitives that are supported. While not disregarding the issue of performance, we take the opposite route, by stating that support for the full kernel of AN2 is the primary goal. This includes the interacting facet and reflection. Of course, we probably have to pay for this in terms of execution speed, but since bindings have become data and loops are defined using reflection in Kernel AN2, this may turn out to be unavoidable without the analysis of Full AN2 constructs. In the framework we have presented compilation and/or interpretation can always be preceded by numerous analysis and optimization phases if this

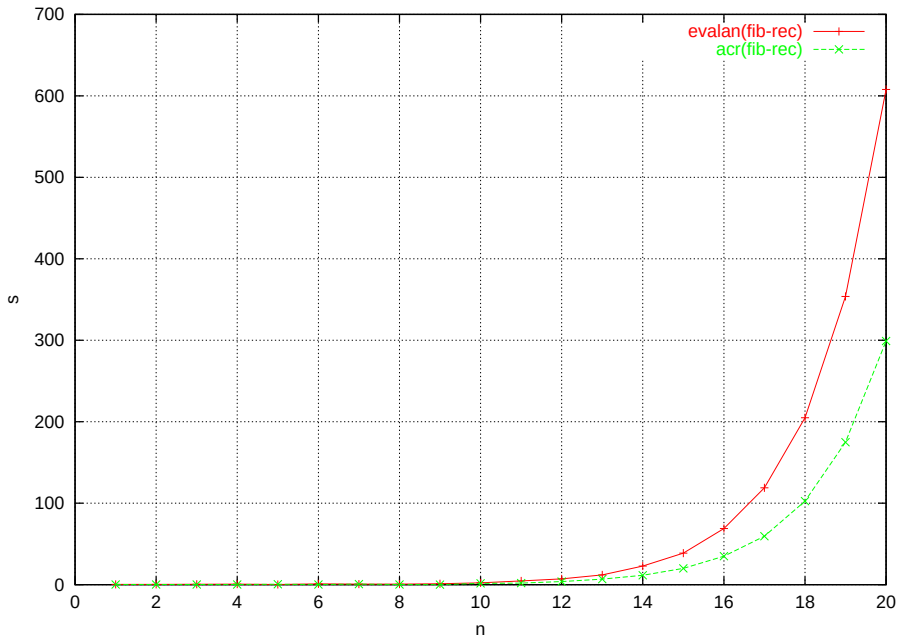

Fig. 7. Execution times in s for Recursive Fibonacci executed using `evalan` and `acr`.

turns out to be desirable. A related design decision is to strive for portability. Therefore the option of compiling to native code has never even been considered.

A third difference to existing approaches is our focus on using actions in the real world. Put differently, we see action semantics not only as a formalism to mathematically define an programming language's semantics, but possibly also as a way to define domain specific languages [5]. This poses the question on how to connect generated interpreters or compiled actions to existing software environments. One solution to this has been provided by the ASF+SDF Meta-Environment itself: compiled specifications can be connected to the Toolbus coordination architecture [2]. A second solution, addressed in this paper, is compilation of actions to a Java class definition. By letting this class implement the Enactable interface, a compiled action can be combined with handcoded "enactables" using reflection as provided by AN2.

5.2 Conclusions

We have presented a number of interpreters and compilers to put action semantic descriptions to use. From the experiments we have performed some conclusions can be drawn. First of all, if performance is important, the term rewriting approaches will not do. The difference in execution time between the term rewriting

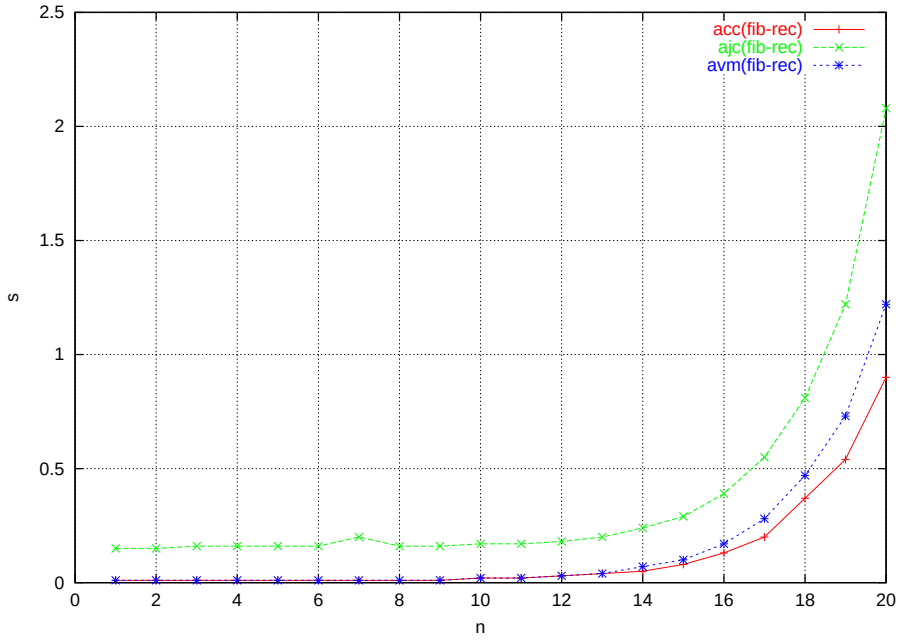

Fig. 8. Execution times in s for Recursive Fibonacci executed using `acc`, `ajc` and `avm`

approach and the compiled actions (C, Java and AIL) is indeed dramatical. This conclusion can be drawn immediately from the plots displayed in Figure 7 and Figure 8[3]. The reason why one would still want to use `evalan` lies in the fact that it can be augmented with arbitrary algebraic data types. Additionally, the tight integration with the ASF+SDF Meta-Environment allows for the interactive specification as well as testing of programming languages. The interpreter `evalan` might best be used during the process of designing a language.

Compiling actions has numerous advantages over interpretation. The resulting objects are faster and easier to embed in existing software environments. The compilers can be instructed to map certain (primitive) actions to user provided native code. This way it is possible to let actions really connect to the real world. We conclude that, if performance is not the main objective, the compilation to Java has the best credits for defining domain specific languages, since it combines Object Orientation with deployability and mobility.

5.3 Future Work

First of all we will have to test the tools presented here with interacting actions as input. Since we have only assessed the performance of a very small action in

[3] All tests have been executed on an AMD Athlon XP1800+ running Linux.

this paper, the question whether the interpreters and compilers will scale up to larger actions is an urgent one. We are currently working on the migration of the JOOS action semantics [19] to ASF+SDF.

Although it is explicitly not our intention to compete with commercial compilers, more benchmarking should enable us to assess the performance penalties of the various strategies more accurately. There is reason to assume that there is room for improvement, especially for the compilers. Restricting the compositionality requirement to apply only to the top action is an option we will have to explore. But this is probably hardly possible without a thorough analysis of actions. These analyses should take both Full AN2 constructs and Kernel AN2 constructs into account. The problem is, that, whereas it is easier to analyse Full AN2 constructs only, a user is still permitted to use Kernel AN2 in his language definitions. As a consequence any agressive optimization that does not take both levels into account may lead to loss of information. For example, algebraic simplification of Kernel AN2 actions might destroy the link between some kernel subactions and their Full AN2 origins (e.g. **unfolding**). An important step to more efficient compilation (especially to C and Java) would be the availability of a staticness condition. It would then be possible to eliminate all bindings and possibly many type checks.

Apart from the issue of performance, we plan to center our future work around increased usability and deployability of actions. This work includes making the interpreters as well as compilers extendible, connecting all tools to the Toolbus coordination architecture, and targeting .NET intermediate language.

References

1. Stephen Adams. Implementing sets efficiently in a functional language. Technical report, University of Southampton Department of Electronics, 1992.
2. J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.
3. M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.
4. Deryck F. Brown, Hermano Moura, and David A. Watt. Actress: an action semantics directed compiler generator. In *Proc. 4th Intl. Conf. on Compiler Construction (CC '92)*, volume 641 of *Lecture Notes in Compute Science*, pages 95–109. Springer-Verlag, 1992.
5. A. van Deursen and P. Klint. Little languages: Little maintenance? In S. Kamin, editor, *First ACM-SIGPLAN Workshop on Domain-Specific Languages; DSL'97*, pages 109–127, January 1997. Technical report, University of Illinois, Department of Computer Science.
6. Jay Earley and Howard Sturgis. A formalism for translator interactions. *Communications of the ACM*, 13(10):607–617, 1970.

7. J. Field, J. Heering, and T. B. Dinesh. Equations as a uniform framework for partial evaluation and abstract interpretation. *ACM Computing Surveys (CSUR)*, 30(3es):2, 1998.
8. E. Gamma, Helm R., R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Massachusetts, USA, 1994.
9. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.
10. Søren B. Lassen, Peter D. Mosses, and David A. Watt. An introduction to AN-2, the proposed new version of Action Notation. In Mosses and de Moura [14], pages 19–36.
11. Pierre-Etienne Moreau. A choice-point library for backtrack programming. In *Implementation Technology for Programming Languages based on Logic*, pages 16–31, 1998.
12. Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.
13. Peter D. Mosses. AN-2: Revised action notation–syntax and semantics, 2000.
14. Peter D. Mosses and Hermano Perrelli de Moura, editors. *AS2000*, Recife, Brazil, 2000. BRICS, Dept. of Computer Science, Univ. of Aarhus.
15. Peter Ørbæk. Oasis: an optimizing action-based compiler generator. In *Proc. 5th Intl. Conf. on Compiler Construction (CC '94)*, volume 786 of *Lecture Notes in Compute Science*, pages 1–15. Springer-Verlag, 1994.
16. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.
17. Arie van Deursen and Peter D. Mosses. ASD: The action semantic description tools. In *Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 579–582. Springer-Verlag, 1996.
18. Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Amsterdam, 1997.
19. David A. Watt. JOOS action semantics. draft version 2.0, 2000.

How ASF+SDF technology can be used to develop an Action Semantics Environment

M.G.J. van den Brand[1,2] and J.J. Vinju[1]

[1] CWI, Amsterdam, The Netherlands
[2] LORIA-INRIA, Nancy, France

Abstract

ASF+SDF [2] and action semantics [7] have had a close relation over the years. This relation is two-fold: both formalisms can be used to describe (programming) language [5] and ASF+SDF technology has been used to develop tools for action semantics. The first version of the ASF+SDFMeta-Environment [6] was used to develop a prototype of an action semantics environment [3]. The approach taken there was quite complicated because of the use of mixfix syntax used in action semantics on one hand and the lack of re-usability of components, such as the parser, in the Meta-Environment on the other hand. The problem was solved by generating new modules and incorporating these modules to parse the action semantics expressions.

The next generation of the ASF+SDFMeta-Environment [1] is again used to develop an environment for action semantics [8]. The approach is quite different this time. The modularity of ASF+SDF has been fully exploited, each language construct is formulated in a separate module [4]. Furthermore, the syntax and semantics of the action semantics constructs is directly specified in ASF+SDF, which makes the generation phase used in the old action semantics system obsolete.

We will present the technical background of the new ASF+SDFMeta-Environment from the perspective of developing the action semantics environment. The new ASF+SDFMeta-Environment is based on a software coordination architecture and this should offer the possibility to connect tools like a type checker, an interpreter, and a compiler for action semantics in a straightforward manner. We will discuss the various approaches to do this. The aspect of genericity of the Meta-Environment will be discussed and what has to be done when the action semantics modules become less ASF+SDF like. The current prototype if the action semantics environment based on ASF+SDF technology uses directly SDF to describe the syntax rules. This means that quite some SDF specific keywords are used, given the fact that only one syntax rule is specified per module, half of such a module consists of SDF keywords. A logical step would be to develop an action semantics specific SDF variant. This step has quite some consequences on the architectural level of the ASF+SDFMeta-Environment.

References

1. M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.
2. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.
3. A. van Deursen and P.D. Mosses. Executing Action Semantics descriptions using ASF+SDF. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology (AMAST'93)*, Workshops in Computing, pages 415–416. Springer-Verlag, 1993. System demonstration.
4. K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.
5. J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notice*, 35(3):39–48, 2000.
6. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.
7. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
8. P.D. Mosses. Action Semantics and ASF+SDF. In M.G.J. van den Brand and R. Lämmel, editors, *Proceedings of the 2st International Workshop on Language Descriptions, Tools and Applications*, volume 65-3. Electronic Notes in Theoretical Computer Science, 2002. System Demonstration.

An Object-Oriented View of Action Semantics

Claudio Carvilhe[1,2] and Martin A. Musicante[2]

[1] Computer Science Department, Catholic University of Paraná Curitiba-PR-Brazil
[2] Computer Science Department, Federal University of Paraná Curitiba-PR-Brazil

Abstract. Action Semantics is a well known formalism for specifying programming languages. Recently methods were proposed to provide modularity, allowing for the specification reusability and extension in Action Semantics. Modules can also be seen as objects from object-oriented paradigm, increasing the specification organization, once concepts as instantiation and inheritance can be employed. A Pascal like imperative Language is specified using Action Semantics objects, as well as a language extension, obtained by extending the object´s hierarchy representing the original language.

1 Introduction

Action Semantics represents a spreading framework for specifying programming languages. According to [MM1] Action Semantics descriptions has proven to be adequate for reusing and extension. However, the standard Action Semantics, as proposed by [Mo1] lacks of syntactic support for the definition of libraries, whose components would be reused in new descriptions.

Two solutions for this problem have been proposed by different authors. In [DM1], the standard Action Notation is extended to define modules, while the definition of components is described in [MM2].

In this work, we propose the use of Object-Oriented concepts for the definition of yet another extension of Action Notation. Our proposal is motivated by the study of the two previous ones, where some problematic aspects were detected.

A main advantage of our proposal is the use of standard Action Notation for the specifications "in small", combined with class constructors, to provide an object-oriented way of composing specifications.

This work is organized as follows: The next two sub-sections present Action Semantics, as well as the introduction of modules and components in the formalism.

Section 2 presents Object-Oriented Action Semantics by means of an example. The operational semantics of our notation is also given in this section.

Section 3 is a case study: a simple imperative language is specified, and then this description is extended, in order to demonstrate the capabilities of our proposal.

Section 4 is devoted to the conclusions of this work.

1.1 Action Semantics

Action Semantics [Mo1] is a formal framework for describing languages semantics. Specifications in Action Semantics framework, employ the same structure as those in Denotational Semantics, using *semantic functions* and *semantic equations*. The meaning of each phrase is specified using *actions*. An Action, is an entity which receives and propagates data and can be executed. An action, when performed, may produce some outcomes:

- complete: indicating normal termination;
- escape: representing abnormal termination;
- fail: representing that the execution failed;
- diverge: indicating no termination - the action will be executed forever;

Action Semantics is a simpler way for defining formal specifications. Its english language-based writing style constitutes an important factor on such simplicity. However, as reported by [La1], traditional action semantics does not allow the definition of reusable building blocks. Such a requirement, constitutes an important feature on the context of creating new languages specifications based on existing languages structures.

In order to improve modularity, some methods were proposed. Such methods are sumarized in the following section.

1.2 Structuring Action Semantics Descriptions

In [DM1], Doh and Mosses proposed the use of Action Semantics *modules*. Each part of a programming language specification is defined in isolation, allowing for separated view of specification elements; module composition; specification reusability. In [MM2], Menezes and Moura proposed the *Component Based Action Semantics*, allowing the creation of generic components and component libraries.

In both approaches, specification parts can be reused in new projects. A module or component can be written in an abstract way and extended to meet actual needs. The readability is increased since small specification parts compose a large project. In what follow, we summarize some positive and negative points of both approaches:

Modules

(+) The notation for defining modules is simple, improving readability of the resulting specification;
(+) Reusability is enhanced. Specification parts can be used in new specifications;
(+) Modules can be combined, including existing modules in the actual project;
(–) Modules combination is problematic, and can cause inconsistencies (as detailed in [DM1]).

Components

(+) Component notation and Programming Language Notation provide an advanced way to create components and to define language specifications;
(+) Components are generic and can be used in new projects avoiding repetition;
(+) A component library can be defined;
(−) Functions and operations provided by the framework are cumbersome. Specifications can be obscure.

Focused in the specification reusability and extension context, we introduce another approach for the organization of specifications in Action Semantics. Our proposal, consists in the use of Object-Oriented concepts for structuring Action Semantics specifications. The details of our approach are given in the following section.

2 Object-Oriented Action Semantics

An Object-Oriented Action Semantics specification consists on the definition of a *hierarchy of classes*, in which the objects are defined by means of a standard Action Semantics. The class hierarchy for a given language is derived from its syntactic structure. Each language phrase (declarations, commands, expressions, etc) is represented by one or more *objects*. Let us exemplify these concepts using a toy command language:

Example 1 (Commands Abstract Syntax).

- Command = ⟦"skip"⟧ |
  ⟦Identifier ":=" Expression⟧ |
  ⟦"do" Command "while" Expression⟧ |
  ⟦Command ";" Command⟧ .

The non-terminal symbol Command defines the commands present in the language. They are respectively: the null command, assignments, iterations and sequences. There are some features that are common to all commands. However, each one of them presents a particular structure and meaning. Such context can be represented using a *class hierarchy*, defined in figure 1.

Command represents common command features. Specific features can be expressed by Command sub-classes which are: Null, Assignment, DoWhile and Sequencing.

In Object-Oriented Action Semantics classes are divided in two basic parts: syntax and semantics, using the same idea from [DM1] and [MM2].

Example 2 (Command Semantics).

```
Class Command
        syntax:
            Com
        semantics:
            execute _ : Com -> Action
End Class
```

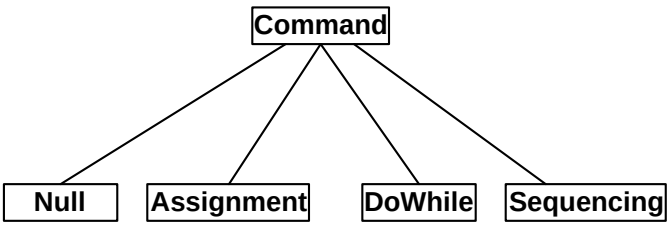

Fig. 1. Example 1 language´s class hierarchy

The class Command, detailed in example 2, is defined to be the command base-class. In the syntax section (`syntax`), the syntactic *sort* `Com` is introduced. This sort will contain all the command syntactic trees.

A semantic function '`execute _`' is defined in the semantic section (`semantics`) indicating a mapping from syntactic sort `Com` to an `Action`. In Object-Oriented Action Semantics, semantic functions and semantic equations has the same role as *methods* in Object-Oriented framework [Ru1,Me1]. The semantic function '`execute _`' can be seen as a method of the class Command, and can be *overloaded*[3] by its sub-classes.

Example 3 (Do-While command semantics).

```
Class DoWhileCommand
        extending Command
        using E:Expression, C:Command
        syntax:
            Com ::= "do" C "while" E
        semantics:
            execute[[ "do" C "while" E ]]=
               unfolding
                        execute C
                    and then
                        evaluate E
                            then unfold else complete
End Class
```

Let us see the example 3. The `extending` directive indicates that `Do-While-Command` class is an extension from `Command` class. Two objects (`E:Expression` e `C:Command`) are instantiated on the `using` directive.

In the syntax section we add a new command structure, redefining the syntax tree (`Com`). The '`execute _`' method is overloaded and the do-while command action semantics is given.

[3] This concept is called polymorphism by the Object-Oriented community.

The classes `Command` and `Expression` must provide the methods '`execute _`' and '`evaluate _`'. In the definition of the method '`execute _`' for `Do-While-Command` class, the functions (methods) '`execute _`' and '`evaluate _`' of objects `C` e `E`, respectively, are called.

The next sections present the syntax and semantics of Object-Oriented Action Notation. The syntax extends standard Action Notation to express object-oriented concepts. The big-step operational semantics of the notation is given in section 2.2.

2.1 Syntax

Specifications in Object-Oriented Action Semantics are structured by a finite class set. The relationship between classes is specified using directives, indicating: which objects will be instantiated by the actual class as well as its position in the class hierarchy. The syntactic structure for classes is defined as follows, using BNF. This syntax definition simply extends the standard Action Notation to include the class apparatus.

(1) Class-Module =
⟦ "Class" Class-Name Class-Body "End-Class" ⟧.
(2) Class-Body =
⟦ ⟨ "extending" Base-Class-Name ⟩? ⟨ "using" Objects-Declaration ⟩?
⟨ Class-Definition ⟩? ⟧.
(3) Base-Class-Name =
⟦ Class-Name | Class-Name "::" Base-Class-Name ⟧.
(4) Objects-Declaration =
⟦ Object-Declaration ⟧ | ⟦ Object-Declaration "," Objects-Declaration ⟧
(5) Object-Declaration =
⟦ Identifier ":" Class-Name ⟧
(6) Class-Definition =
⟦ "syntax" ":" Syntactic-Part "semantics" ":" Semantic-Part ⟧
(7) Syntactic-Part =
⟦ TokenName | TokenName "::=" syntax-tree ⟧
(8) Semantic-Part =
⟦ ⟨ Semantic-Functions ⟩? ⟨ Semantic-Equations ⟩? ⟧.
(9) Semantic-Functions =
⟦ Semantic-Function ⟧ | ⟦ Semantic-Function Semantic-Functions ⟧.
(10) Semantic-Function =
⟦ Function-Name "_"* Tokens "-¿" "Action" | data ⟧.
(11) Tokens =
⟦ TokenName ⟧ | ⟦ TokenName "," Tokens ⟧.
(12) Semantic-Equations =
⟦ Semantic-Equation ⟧ | ⟦ Semantic-Equation Semantic-Equations ⟧.
(13) Semantic-Equation =
⟦ Function-Name "[[" syntax-tree "]]" "=" Action ⟧.

(14) Action =
⟦ "complete" ⟧ | ⟦ "fail" ⟧ | ⟦ "unfold" ⟧ |
⟦ Action "and" Action ⟧ | ⟦ Action "and then" Action ⟧ |
⟦ "unfolding" Action ⟧ | ⟦ "give" Yielder ⟧ |
⟦ "check" Yielder ⟧ | ⟦ Action "then" Action ⟧ |
⟦ "bind" Yielder "to" Yielder ⟧ |
⟦ "furthermore" Action ⟧ | ⟦ Action "hence" Action ⟧ |
⟦ Action "moreover" Action ⟧ | ⟦ Action "before" Action ⟧ |
⟦ "store" Yielder "in" Yielder ⟧ | ⟦ "deallocate" Yielder ⟧ |
⟦ "enact" Yielder ⟧ | ⟦ Action "else" Action ⟧ |
⟦ "recursively" "bind" Yielder "to" Yielder ⟧ |
⟦ "allocate a" Sort ⟧ .

(15) Yielder =
⟦ "the" Sort "#" n ⟧ |
⟦ "the" Sort "bound" "to" k ⟧ |
⟦ "the" Sort "stored" "in" Yielder ⟧ |
⟦ Yielder "with" Yielder ⟧ |
⟦ "closure" Yielder ⟧ |
⟦ "abstraction of" Action ⟧ .

(16) Sort =
⟦ "bindable" ⟧ | ⟦ "cell" ⟧ | ⟦ "cell" "[" Sort "]" ⟧ |
⟦ "storable" ⟧ | ⟦ "abstraction" ⟧ | ⟦ "datum" ⟧ |
⟦ "integer" ⟧ | ⟦ "value" ⟧ | ⟦ "truth-value" ⟧ |
⟦ Sort "—" Sort ⟧ .

The following section defines the operational semantics of this notation.

2.2 Semantics

Base-classes defined using our notation are *implicitly* part of a hierarchy. Object-Oriented Action Notation defines a class to which all classes belong. This *super-class* is called *State*. All classes defined in Object Oriented Action Semantics are sub-classes of *State*.

The class *State* has generic attributes, corresponding to transient information, bindings and storage; and provide *operations*, allowing us to handle such attributes. Both attributes and operations are visible to all specific classes. We define the *State* methods behavior by means of a big-step operational semantics [Wi1], characterized by the following relation:

$$S = B, t, b, s \vdash o \rhd o', t', b', s'$$

The relation schema above identifies B as the *methods environment* (user-defined methods), t as the transient information, b as bindings and s as the current store. We say that the operation (action) o, when performed produces the outcome o', together with transient information t', bindings b' and storage s'. The definition of such a relation is adapted from [Mo2].

2.2.1 Operations on *State* Operations are functions on *State* attributes. Operations maps the triple (t, b, s) to (t', b', s') and are classified according to the action facets [Mo1], being divided as follows:

- *basic*: when applied to flow control;
- *functional*: when applied to *transients*;
- *imperative*: when applied to current *storage*;
- *declarative*: when applied to the *bindings*;
- *reflective*: defining abstractions;
- *hybrid*: applied to more than one attribute.

The next rules define some basic operations.

$$B, t, b, s \vdash \mathsf{complete} \rhd \mathsf{completed}, \{\}, \{\}, s \tag{1}$$

$$B, t, b, s \vdash \mathsf{fail} \rhd \mathsf{failed}, \{\}, \{\}, s \tag{2}$$

$$\frac{B, t, b, s \vdash a_1 \rhd \mathsf{completed}, t_1, b_1, s_1 \quad \mathsf{mergeable}\ t_1 t_2 \qquad B, t, b, s_1 \vdash a_2 \rhd \mathsf{completed}, t_2, b_2, s_2 \quad \mathsf{mergeable}\ b_1 b_2}{B, t, b, s \vdash a_1\ \mathsf{and\ then}\ a_2 \rhd \mathsf{completed}, t_1 \oplus t_2, b_1 \oplus b_2, s_2} \tag{3}$$

$$\frac{B, t, b, s \vdash a_1 \rhd o_1, t_1, b_1, s_1 \qquad o_1 \neq \mathsf{completed}}{B, t, b, s \vdash a_1\ \mathsf{and\ then}\ a_2 \rhd o_1, t_1, b_1, s_1} \tag{4}$$

$$\frac{B, t, b, s \vdash a_1 \rhd \mathsf{completed}, t_1, b_1, s_1 \qquad B, t, b, s_1 \vdash a_2 \rhd o_2, t_2, b_2, s_2 \quad o_2 \neq \mathsf{completed}}{B, t, b, s \vdash a_1\ \mathsf{and\ then}\ a_2 \rhd o_2, t_2, b_2, s_2} \tag{5}$$

$$\frac{B, t, b, s \vdash a_1 \rhd o_1, t_1, b_1, s_1 \qquad o_1 \neq \mathsf{failed}}{B, t, b, s \vdash a_1\ \mathsf{or}\ a_2 \rhd o_1, t_1, b_1, s_1} \tag{6}$$

$$\frac{B, t, b, s \vdash a_1 \rhd \mathsf{failed}, \{\}, \{\}, s_1 \qquad B, t, b, s_1 \vdash a_2 \rhd o_2, t_2, b_2, s_2}{B, t, b, s \vdash a_1\ \mathsf{or}\ a_2 \rhd o_2, t_2, b_2, s_2} \tag{7}$$

Rules (1) and (2) indicate the semantics of the operations complete and fail, respectively. Rules (3), (4) and (5) defines the behavior of the and then operation. Rule (3), establishes the behavior of the and then operation when both sub-operations (sub-actions) a_1 and a_2 completes. Rule (4) defines the behavior of the and then operation when the first sub-operation a_1 does not complete, and rule (5) establishes the behavior when sub-operation a_2 does not complete.

Rules (6) and (7) defines the behavior of the operation or. Rule (6) defines the behavior of the or operation when sub-operation a_1 does not fail, and rule (7) indicates the results when a_1 fail.

Notice that all kinds of operations can be obtained using operational rules. A full description can be seen in *www.ppgia.pucpr.br/~carvilhe/full*.

2.2.2 Methods environment All user-defined methods are stored in the methods environment. Two operations are needed to handle methods: insertion and fetch. These operations are defined in this section.

Methods are structured as $f[\![H]\!] = o$, being H a *syntax-tree* schema[4] [Mo1] and o an operation (action). The behavior of the operations is defined by the following rules:

$$\frac{}{B \ \vdash \ f[\![H]\!] = O \ \rhd \ B[f[\![H]\!] \rightarrow O]} \tag{8}$$

$$\frac{(f[\![H]\!]) \in dom(B), \qquad B, t, b, s \ \vdash \ O[H/h] \ \rhd \ o, t', b', s'}{B, t, b, s \ \vdash \ f[\![h]\!] \ \rhd \ o, t', b', s'} \tag{9}$$

Rule (8) indicates that methods defined by means of the syntax '$f[\![H]\!] = O$' must be inserted in the methods environment. The notation $B[f[\![H]\!] \rightarrow O]$ indicates that a new method is inserted in the environment. Rule (9) defines that methods defined as $f[\![h]\!]$ produces de result o, t', b', s', provided that the method f exists in the methods environment B, and that the method can be executed by substituting the *syntax-tree* schema H by the actual syntax-tree h.

3 A case study - The μ-Pascal Language

In this section we present the Object-Oriented Action Semantics of a toy language similar to Pascal. μ-Pascal is a simple imperative programming language which contains simple commands and expressions. The language possesses a block-structure.

The Object-Oriented Action Semantics of μ-Pascal is defined in this work to exemplify the extensibility properties of our framework. In section 3.3, the μ-Pascal Object-Oriented Action Semantics will be extended to a language with procedures and functions.

μ-Pascal syntax is defined as follows.

3.1 μ-Pascal Syntax

(1) Program =
    ⟦ "declare" Declaration "used-in" Command ⟧.

[4] We call a syntax tree schema as a syntax tree with (free) variables standing for sub-trees.

(2) Declaration =
⟦ "var" Identifier ":" Type ⟨ "=" Expression ⟩? ⟧ |
⟦ "const" Identifier ":" Type "=" Expression ⟧ .
(3) Type =
"boolean" | "integer"

A Program is formed by a declaration followed by commands. The tokens declare and used-in delimitates these structures. Constants and Variables can be declared indicating their type.

(1) Command =
⟦ "declare" Declaration "used-in" Command ⟧ |
⟦ Identifier ":=" Expression ⟧ |
⟦ "if" Expression "then" Command ⟨ "else" Command ⟩? "end-if" ⟧ |
⟦ "repeat" Command "until" Expression ⟧ |
⟦ Command ";" Command ⟧ .
(2) Expression =
⟦ "true" ⟧ | ⟦ "false" ⟧ | ⟦ Numeral ⟧ | ⟦ Identifier ⟧ |
⟦ Expression "+" Expression ⟧ | ⟦ Expression "-" Expression ⟧ |
⟦ Expression "*" Expression ⟧ | ⟦ Expression "¡" Expression ⟧ |
⟦ Expression "=" Expression ⟧.
(3) Identifier =
⟦ Letter ⟨ Letter — Digit ⟩* ⟧.
(4) Numeral =
⟦ Digit ⟨ Digit ⟩* ⟧.

Commands in μ-Pascal contains assignment, selection, iteration and sequences. Expressions can be arithmetical or logical.

3.2 μ-Pascal Semantics

Based on the μ-Pascal abstract syntax a class hierarchy can be defined. The syntactic rules of the language definition can be mapped into objects which incorporate syntax and semantics. Some diagrams will be constructed using a tree notation to express the hierarchy.

Let us begin by constructing classes to represent the language identifiers, numerals and types. After this phase, we define the declarations, commands and expressions hierarchy to express the meaning of the language.

3.2.1 Identifiers, Numerals and Types

```
Class Identifier
        syntax:
            Id ::= letter [ letter | digit ]*
End Class
```

```
Class Numeral
        syntax:
            N ::= digit+
        semantics:
            valuation _ : N -> integer
End Class

Class Type
        syntax:
            T ::= "boolean" |   "integer""
        semantics:
            allocate-for-type _ : T -> Action
            allocate-for-type[[ "boolean" ]] =
                allocate a cell [ truth-value ]
            allocate-for-type[[ "integer" ]] =
                allocate a cell [ integer ]
End Class
```

The class `Identifier` represents names (variables and constants). This class has only syntactic definition.

The class `Numeral` represents integer numbers. In the `semantics` section a '`valuation _`' method is defined, mapping numerals to integers. The definition of the '`valuation _`' function is straightforward.

The class `Type` represents the language data types. The method '`allocate-for-type`' is defined, establishing truth-value or integer data types allocation.

3.2.2 Declarations Constants and variables can be declared in μ-Pascal. The declarations hierarchy can be constructed based on figure 2 as follows:

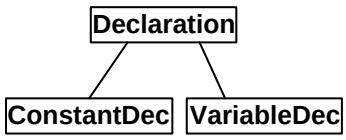

Fig. 2. μ-Pascal´s declarations hierarchy

```
Class Declaration
        using I:Identifier, E:Expression
        syntax:
            Dec
        semantics:
            elaborate _ : Dec -> Action
End Class
```

The class `Declaration` instantiates two objects: `I` and `E`, respectively of `Identifier` and `Expression`. Both objects are not used by the class itself. They are defined at this point to be available to the sub-classes of `Declaration`. A non-terminal symbol `Dec` is introduced at the syntax section to be redefined by the sub-classes. In the semantics section an '`elaborate _`' method is declared, to be overloaded by the sub-classes of `Declaration`.

Let us define the first sub-class of `Declaration`:

```
Class ConstantDec
        extending Declaration
        syntax:
            Dec ::= "const" I ":" T "=" E
        semantics:
            elaborate[["const" I ":" T "=" E]]=
                evaluate E then
                bind I to the value
End Class
```

The `Dec` token is redefined in the `ConstantDec` class (above), establishing the structure for constant declarations. In the semantics part, the '`elaborate _`' method is redefined. The objects `I:Identifier` and `E:Expression` are used at this point, and the semantics of a constant declaration is defined using standard Action Notation.

```
Class VariableDec
        extending Declaration
        using T:Type
        syntax:
            Dec ::= "var" I ":" T ["=" E]
        semantics:
            elaborate[["var" I ":" T]]=
                    allocate-for-type T
            then
                    bind token I to the cell #0
            elaborate[["var" I ":" T "=" E]]=
                        evaluate E
                and
                        allocate-for-type T
            then
                    bind I to the cell #1
                and
                    store the value #0 in the cell #1
End Class
```

The `VariableDec` class instantiates a `T:Type` object. The variable declaration syntax is redefined at the syntax section. The '`elaborate _`' method is redefined twice, considering the absence (or not) of an expression `E`.

3.2.3 Commands The commands class hierarchy is defined based on the figure 3. The main class of this hierarchy is `Command`, as follows.

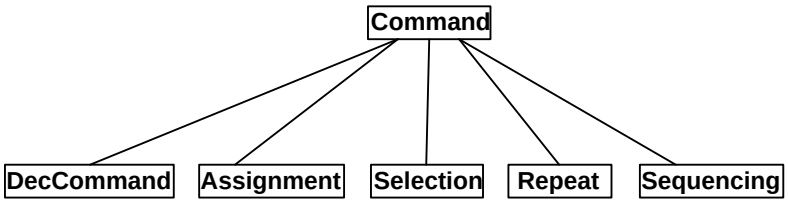

Fig. 3. μ-Pascal´s commands hierarchy

```
Class Command
        syntax:
            Com
        semantics:
            execute _ : Com -> Action
End Class
```

In the syntax section, the token `Com` is introduced to represent the syntactic sub-trees of commands. The method `'execute _'` is declared in the semantics section to be redefined by the sub-classes of `Command`.

```
Class DecCommand
        extending Command
        using D:Declaration, C:Command
        syntax:
            Com ::= "declare" D "used-in" C
        semantics:
            execute[["declare" D "used-in" C]]=
                    furthermore elaborate D
                hence
                    execute C
End Class
```

The class `DecCommand` is introduced in the commands hierarchy on the `'extending'` directive. Two objects are instantiated by `DecCommand` class: `D:Declaration` and `C:Command`. In the syntax section the token `Com` is redefined to express declarations followed by commands. In the semantics section the standard action semantics description is defined.

```
Class Assignment
        extending Command
        using I:Identifier, E:Expression
        syntax:
            Com ::= I ":=" E
        semantics:
            execute[[I ":=" E]]=
                    evaluate E
                then
                    store the value in the cell bound to I
End Class
```

The class `Assignment` instantiates two objects `I:Identifier` and `E:Expression`. In the syntax section the assignments structure is represented. In the semantics section, assignment´s semantics is defined.

Notice that the remaining classes `Selection`, `Repeat` and `Sequencing` are structured in the same way as `DecCommand` and `Assignment`. Thus, their definitions are omitted in this paper.

3.2.4 Expressions The expressions class hierarchy is defined based on figure 4.

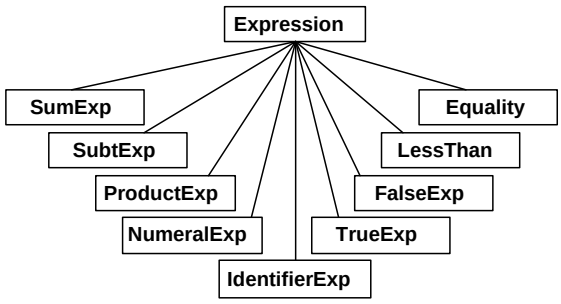

Fig. 4. μ-Pascal´s expressions hierarchy

```
Class Expression
        syntax:
            Exp
        semantics:
            evaluate _ : Exp -> Action
End Class
```

The super-class `Expression` presents a similar structure from `Command`. In the syntax section a token `Exp` is introduced to represent the syntactic sub-trees of expression. In the semantics section the method `'evaluate _'` is introduced.

```
Class SumExp
        extending Expression
        using E1:Expression, E2:Expression
        syntax:
            Exp ::= E1 "+" E2
        semantics:
            evaluate[[ E1 "+" E2 ]] =
                    evaluate E1
                and
                    evaluate E2
            then
                give sum(the integer#0, the integer#1)
End Class
```

The class `SumExp`, instantiates two objects from class `Expression` (`E1` and `E2`). The token `Exp` is redefined in the syntax section, and the semantics of a sum of expressions is specified in the semantics section.

```
Class IdentifierExp
        extending Expression
        using I:Identifier
        syntax:
            Exp :: = I
        semantics:
            evaluate [[ I ]] =
                    give the value bound to I
                or
                    give the value stored in
                    the cell bound to I
End Class
```

The class `IdentifierExp` instantiates an object `I:Identifier`. The syntax of expressions is extended to represent identifiers. In the semantics section, the action semantics of a identifier is defined.

Notice that the classes `SubtExp` and `ProductExp` can be obtained in the same way as `SumExp`. Both `SubtExp` and `ProductExp` classes, and also `NumeralExp`, `TrueExp`, `FalseExp`, `LessThan` and `Equality` are omitted in this paper.

3.2.5 The Language Once the declarations, commands and expressions hierarchies are defined, it is possibly now to specify the class representing the whole μ-Pascal language.

```
Class muPascalLanguage
        using D:Declaration, C:Command
        syntax:
            Prog ::= "declare" D "used-in" C
        semantics:
            run _ : Prog -> Action
            run[[ "declare" D "used-in" C ]]
                elaborate D
            hence
                execute C
End Class
```

Two objects are instantiated in the `muPascalLanguage` class: `D:Declaration` and `C:Command`. The `Prog` token defines the μ-Pascal program general structure. In the semantics section, a `'run _'` method is defined mapping a program to an action.

3.3 Extending the Specification - The m-Pascal Language

Let us now use the class hierarchy defined in the previous sections, to create a new language, which we will call m-Pascal. The language incorporates two imperative languages fundamental structures to μ-Pascal: procedures and functions. First of all, the language syntax must be extended to contain the new structures.

3.3.1 m-Pascal syntax

(1) Declaration =
... |
⟦ "proc" Identifier "(" Formal-Par ")" "=" Command ⟧ |
⟦ "func" Identifier "(" Formal-Par ")" "=" Expression ⟧ .

(2) Formal-Par =
⟦ "var" Identifier ":" Type ⟧.

(3) Command =
... | ⟦ "call" Identifier "(" ⟨ Actual-Par ⟩ ")" ⟧ .

(4) Expression =
... | ⟦ Identifier "(" ⟨ Actual-Par ⟩ ")" ⟧.

(5) Actual-Par =
⟦ Expression ⟧.

The abstract syntax of declarations is extended by rule (1) to consider procedures and functions. Commands and expressions are also changed in (3) e (4) respectively, to include procedure calls and function applications.

3.3.2 Procedures and Functions The objects, containing the definitions of procedures and functions are expressed by extending the `Declaration` class

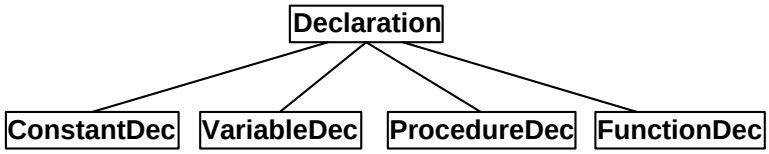

Fig. 5. m-Pascal´s declarations hierarchy

defined in the m-Pascal specification (section 3.2.2). The declarations hierarchy will be changed to include two new sub-classes: `ProcedureDec` and `FunctionDec`, according to figure 5:

```
Class ProcedureDec
    extending Declaration
    using C:Command, FP:FormalPar
    syntax:
        Dec ::= "proc" I "(" FP ")" ":" C
    semantics:
    elaborate[["proc" I "(" FP ")" ":" C]]=
        recursively bind I
            to closure abstraction of
                    furthermore elaborateFP FP
                hence
                    execute C
End Class

Class FunctionDec
    extending Declaration
    using T:Type, FP:FormalPar
    syntax:
        Dec ::= "func" I "(" FP ")" ":" E
    semantics:
    elaborate[["func" I "(" FP ")" ":" E]]=
        recursively bind I
            to closure abstraction of
                    furthermore elaborateFP FP
                hence
                    evaluate E then give the value
End Class
```

Notice that the actions describing the semantics of the new constructors are written in standard action notation, being similar to those in [Mo1].

Two auxiliary classes are now defined to represent formal and actual parameters. The classes `FormalPar` and `ActualPar` are defined as follows:

```
Class FormalPar
        using I:Identifier, T:Type
        syntax:
            FP ::= "var" I ":" T
        semantics:
            elaborateFP _ : FP -> Action
            elaborateFP [[ "var" I ":" T ]] =
                allocate-for-type T
            then
                bind I to the cell #0
End Class

Class ActualPar
        using E:Expression
        syntax:
            AP ::= E
        semantics:
            evaluateAP _ : AP -> value
            evaluateAP [[ E ]] =
                evaluate E
End Class
```

3.3.3 Procedure Call and Function Application Procedure calls and function applications require additions to `Command` and `Expression` hierarchy. These changes are represented in figure 6 and figure 7, respectively:

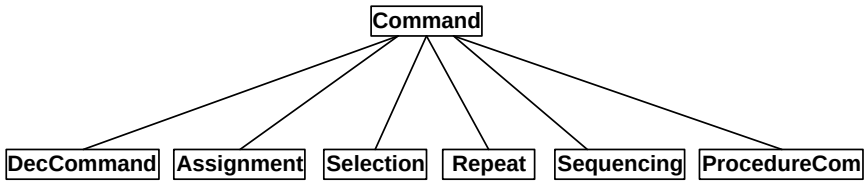

Fig. 6. m-Pascal´s commands hierarchy

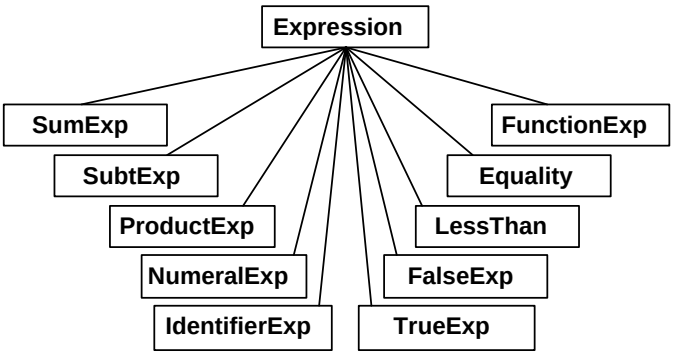

Fig. 7. m-Pascal´s expressions hierarchy

```
Class ProcedureCommand
    extending Command
    using I:Identifier, AP:ActualPars
    syntax:
        Com ::= "call" I "(" AP ")"
    semantics:
    execute[["call" I "(" AP ")"]]=
            evaluateAP AP
        then
            enact ( the procedure bound to I
                with the value )
End Class

Class FunctionExpression
        extending Expression
        using I:Identifier, AP:ActualPars
        syntax:
            Com ::= I "(" AP ")"
        semantics:
            execute[["call" I "(" AP ")"]]=
                    evaluateAP AP
                then
                    enact(the function bound to I
                        with the value)
End Class
```

3.3.4 The Extended Language Object-Oriented features where applied to extend the μ-Pascal language. `ProcedureDec` and `FunctionDec` objects were created extending `Declaration`, to represent procedures and functions. Procedure calls were incorporated to `Command`, based on the creation of the `ProcedureCom` object and function applications were incorporated to expressions, once the `FunctionExp` class was defined in the `Expression` hierarchy.

Since specialization were applied to define the new language features, the main class can be defined as follows:

```
Class mPascalLanguage
        using D:Declaration, C:Command
        syntax:
            Prog ::= "declare" D "used-in" C
        semantics:
            run _ : Prog -> Action
            run[[ "declare" D "used-in" C ]]
                elaborate D
            hence
                execute C
End Class
```

Notice that since the language was redefined at the declarations, commands and expressions levels, the body of the most general class for the language remains the same.

4 Conclusions And Future Work

The application of object-oriented concepts represents an interesting approach to Action Semantics descriptions. We have shown, based in a simple case study, a way to construct objects that can be instantiated by others and also specialized, building specifications based on objects hierarchy.

Object-Oriented Action Semantics represents a new way to organize Action Semantics descriptions. In the case study we built a class hierarchy representing the μ-Pascal language semantics and using object-oriented features we provided specifications reusability and extension, defining a new language named m-Pascal.

We summarize the contributions of our approach, as follows:

- Object-Oriented Action Notation is simple. The class structure is based on *modules* notation introduced in [DM1], and was inspired by the class syntax for object-oriented programming languages;
- Reusability is reached using instantiation and extension reached defining a class hierarchy. Specification parts reuse and extension can be obtained avoiding repetition.

Our future research topics are summarized as follows:

- Formal approach were only defined to operations and user-defined methods. Formal definition of classes, hierarchy, extension and instantiation are still under development;
- Action Notation defined in this work is based on [Mo2]. One future research is the extension of our approach considering the standard Action Notation, including interleaving and concurrency, as in [Mo1].

References

[DM1] Doh, K., Mosses, P.: Composing programming languages by combining action semantics modules. In *Elsevier Science Publishers*, 2002.

[La1] Labra, J.: Reusable Semantic Specifications of Programming Languages, Department of Computer Science - Univerity of Oviedo, Spain. In *SBLP 2002 - VI Brazilian Symposium on Programming Languages*, 2002.

[MM1] Musicante, M. A., Mosses, P.: An Action Semantics for ML Concurrency Primitives. In *Formal Methods Europe*, 1994.

[MM2] Menezes, L. C., Moura, H. P.: Component-based action semantics: A new approach for programming language specifications. In *SBLP 2001 - V Brazilian Symposium on Programming Languages*, pages 152–163, Curitiba-Brazil, 2001. Universidade Federal do Paraná.

[Me1] Meyer, B.: Object-Oriented Software Construction 2nd ed., Prentice-Hall, 1997.

[Mo1] Mosses, P.: Action Semantics, Cambridge University Press, 1992.

[Mo2] Moura, H.: Action Notation Transformations - Ph.D. Thesis, University of Glasgow, 1993.

[Ru1] Rumbaugh, J.: Object-Oriented Modeling and design, Campus, 1994.

[Wi1] Winskel, G.: The Formal Semantics of Programming Languages: An Introduction, Foundations of Computing Series, MIT Press, 1993.

An Extensible Notation for Action Semantics

Luis Menezes, Hermano Moura, and Geber Ramalho

Center of Informatics
Federal University of Pernambuco
Caixa Postal 7851
CEP 50732-970
Recife, PE - Brazil
E-mail: {lcsm,hermano,glr}@cin.ufpe.br
Phone: +55 81 32718430 r4314

Abstract. Action notation allows the elegant formal definition of programming languages. However, this notation does not contain any well-defined process to define new action operators. This lack of extensibility makes difficult define new operators to express semantic features witch were not supported by original action notation.
To bypass this problem, the action notation formal semantics has been recently redesigned using more extensible approaches (Modular SOS). This redesign simplified the process of defining new actions.
However, this change demands that compiler generator tools have to deal with this action semantics definition for processing action semantics descriptions using new actions. This feature makes these systems complex and hard to propose generic optimization techniques.
This paper proposes a new approach to improve the action notation extensibility using a modular meta-notation, formed by a set of generic operators that can be used to define the action notation facet's behaviors. By defining the action notation semantics using the proposed meta-notation, the programming language designer can use the same operators that described the action notation to create action notation extensions, which describe programming languages' new features, or customize the action notation to better fitting in the described language properties.

1 Introduction

Action semantics [3] is a useful formalism designed to describe programming languages using a friendly meta-notation (action notation). Besides their friendly notation and support of programming language's concepts found in conventional programming languages, it is difficult using the action notation to describe unusual programming languages because action notation is not designed to allow the inclusion of the features in the model.

Another problem found in action semantics is the complexity of its notation that difficulties the study of the programming languages properties and define more efficient analysis algorithms.

To resolve these problems, the action notation has been recently redesigned and new proposed action notation were defined to resolve these problems. The

proposed notation is based on a reduced kernel [5] with a small number of primitive operators and their formal definition is based on a modular extension of structural operational semantics (modular SOS) [4] that would allow the insertion of new features, preserving the existing operators' properties.

However, the revised action notation is still based on a fixed set of primitive facets. This feature difficulties the definition of new kinds of data and control flow because these definitions demand the change of the action notation formal definition written in modular SOS.

The building of action semantics processors for defined programming languages that use new action operators written in modular SOS is a complex task because it forces the analysis system to deal with the two different used formalisms (unified algebras and modular SOS).

This paper presents a new action notation kernel that can be used to define actions of the original notation and the new proposal one. Initially, we describe the most important primitive actions existing in our proposal; the following section shows how the proposal notation can be used to describe one of the action notation facets; the following section compares the primitive operators existing in the proposal action notation version 2 and our proposal; finally the last section shows some of the proposal notation advantages to build action semantics compiler generators systems and concludes this paper with future research directions.

2 An Extensible Definition

This paper proposes to define the action notation using a simple meta-notation formed by operators that can be used to define the action notation facet's behaviors.

Our proposed notation improves the reusability and simplicity in action semantics because it does not fix the existing set of facets. The standard action notation defines a fix set of facets and operators that implicitly manipulate them (for example the yielders the given integer and the integer bound to "a"). In the proposed notation, the designer has to specify the desirable facet set (transients, bindings, etc) and the primitive operators are parameterized with the facet that it will access (for example: current transients, current bindings).

This property makes the primitive operators become more general and able to express similar concepts in different facets (retrieve the current facet value, for example) and the operators can be used in the definition of new facets, avoiding the meta-notation redefinition.

The proposed meta-notation operators are classified according to the kind of concept they manipulate. The next sections will describe each kind of concept and the primitive operators that manipulate them.

2.1 Basic Actions

complete : action

_ and then _ :: action, action → action.
_ ; _ :: action, action → action.
$(x\ ;\ y) = x$ and then y.
_ choice _ :: action, action → action.
_ □ _ :: action, action → action.
$x \square y = x$ choice y.

The basic actions are used to model simple kinds of control flows used by actions, the following basic operators are defined: The action complete represents an action that, when is executed, generates no execution error neither produces any kinds of side effects; the sequential combinator a and then b (or the simplified form $a\ ;\ b$) executes the action a and b sequentially; and the action choice combinator a choice b (or the simplified form $a \square b$) chooses one of its sub-actions to execute.

The following action illustrates the basic actions' behavior:

|complete ; complete
□
|complete

This action chooses between execute the complete action once or twice.

2.2 Stack of Sorts

push _ :: datum → action.
apply _ :: datumOp → action.
dup : action.

In the proposed notation, the operations are executed by manipulating a stack of terms that holds temporarily calculated values. To manipulate this stack the following primitive actions were proposed: The action push c pushes a constant datum into the stack; the action apply o pops terms from the stack (according to the arity of o), applies the operators with these terms and put the result in the stack; finally, the action dup duplicates the term stored in the stack top.

To exemplify the stack manipulation actions, the following action:

push 2;
dup;
push 3;
apply sum (_, _);
apply product (_,_)

initially pushes the constant 2, duplicates the stack top term (2) and pushes the constant 3. After this initialization, the following actions sums apply sum (_,_) and multiplies apply product (_,_) two values of the stack top and pushes the resulted value.

2.3 Producers

produce _ :: producer → action.
frame _ _ :: producer, action → action.

Actions can produce useful information to be used later by the program. In order to model this feature, the action processor maintains a list of produced values. Each one of these produced values are labeled with a producer value to express the information type.

To model the produced information handling, the following actions are defined: the action produce p gets one term from the stack and stores it in the list of produced terms with the label p; and the action frame p a executes the action a and, after the execution, a tuple formed by the values with label p produced by a is pushed into the stack.

The following action illustrates the use of produced values in actions:

```
frame P1                          (1)
|frame P2                         (2)
||push 2 ; produce P1 ;           (3)
||push 3 ; produce P2 ;           (4)
||push 4 ; produce P2             (5)
|and then                         (6)
||apply sum _ ; produce P1        (7)
and then                          (8)
|apply product _                  (9)
```

This action initially produces the values 2,3,4 respectively labeled with the producers P1,P2,P2 (lines 3-5). After this execution the internal frame action (line 2) takes the values produced with label P2 (3,4)and pushes them into the stack. The next action (line 7) gets these values and applies the sum _ operator to them, resulting a value (7) that will be produced with label P1. After the execution of this action, the external frame action (line 1) will take all produced values with label P1 and push them into the stack. Finally, the last performed action (line 9) will take the tuple of values from the stack and calculate their product.

2.4 Consumers

select _ _ :: consumer, action → action.
current _ :: consumer → action.

Actions can also receive information necessary to calculate the execution result. To model this behavior, the action processor maintains a list of values given to this action by external entities. Like the producer list, each value given by the action is labeled by a consumer information, indicating its kind. Differently of the producers, that support several values labeled with the same values, the consumer list can have at most one value labeled with the same value.

To handle this kind of information the following action were defined: The action current c that pushes into the stack the information labeled with c, stored in the list of current values; and the action select c a that removes one element from the stack and executes the action a. During the execution of a, the term removed from the stack is stored in the list of current values with label c. If there is other value labeled with the same consumer, this value is temporally overridden during the action of a.

The following action exemplifies the use of consumers actions.

```
push 10 ; select C1    (1)
|push 20 ; select C2   (2)
||current C1;          (3)
||current C2;          (4)
||apply sum (_, _)     (5)
```

The first performed actions (lines 1,2) calculate 2 values (10, 20) and put them into the list of given values with labels C1 and C2, respectively. The following action retrieves these values (lines 3,4) and calculate the sum of both values (line 5).

2.5 Storage

```
allocate : action.
store : action.
stored : action.
deallocate : action.
```

The action allocate reserves an unused memory location and pushes this position into the stack. The action store removes a pair of terms (representing the memory position and the updated value) from the stack and stores the value in the memory position. The action stored removes a memory position from the stack and pushes the value stored in this memory position. Finally, the action deallocate pops a memory position from the stack and makes it available to be used in later memory allocations. The following action exemplifies the proposed storage actions.

```
allocate ; dup ;
push 2 ; store ;
stored ;
```

This action, when it is performed, allocates a free memory cell, and duplicates the value stored in the stack top (the allocated memory cell). The following action puts the value 2 into the stack and execute the action store. This action takes the recently allocated memory cell and the value 2 from the stack and stores this value into the memory position. The following action takes a memory position from the stack and returns the value stored in it.

2.6 Jumps

jump-id = □ .
jump : action.
_ trap _ :: action, handler^{+} → action.
_ : _ ; :: jump-id, action → handler.
fail-action : action.
fail-action = □ .

Jumps are concerned to model breaks in the normal sequencing execution flow defined by the basic actions. To handle jumps the following action were defined: The action jump gets a tuple from the stack and makes the jump, the first value of the retrieved tuple must be of the sort jump-id that identifies the jump type; the action combinatory a trap h executes the action a and captures the jumps produced during this action performance. The combinator trap contains a sequence of jump handlers (h) that specifies how the combinator will act when a jump is performed by the action.

Each jump handler is formed by a pair of a jump identification jump-id subsort and an action. It means that when jump identified by a subsort of the specified identification happens, the associated action should be performed. If no handler is able to capture the performed jump, it will be propagated.

The notation of jumps also defines a constant action fail-jump is executed when an action tries to push the soft nothing into the stack and is used to model the action processor behavior when a evaluation error occurs.

The following action exemplifies the jump actions execution:

```
|push (j2,10); jump;                (1)
|push 3 ; produce RESULT;           (2)
trap                                (3)
|j1: complete;                      (4)
|j2: push 2; apply element of _;    (5)
|    produce RESULT;                (6)
```

When this action is performed, the first action (line 1) executes a jump identified with the pair (j2,10). The following action (line 2) will not be executed and the control will be transferred to the handlers of the current performed trap combinator (line 3). This combinator will select the appropriated handler to be executed (according to the jump kind) and execute the associated action (lines 5,6).

2.7 Reflexive Actions

enact : action

The only reflexive action defined by this notation is the action enact, that gets an action from the stack and executes it. The following action exemplifies this action:

```
push (push 5; sum);
select Proc
|push 2; retrieve Proc; enact;
|retrieve proc; enact
```

The first action pushes an action that sums 5 to a value from the stack. This action is labeled with Proc and passed as given value to the rest of action, which twice retrieves the labeled action and executes the specified procedure.

2.8 Processes

```
process-id = □ .
create a process _ :: action → action.
sleep : action.
wakeup : action.
destroy : action.
wait : action.
```

The process notation deals with process that parallely performs actions. In the proposal notation environment, several processes can be executing in parallel. Each process has a private process identification number, stack and current/produced values lists. All processes share the same memory.

The action create a process a defines a new process that will execute the action a. The created process will import the current consumer values and their stack are initialized with two process identifications that specify the recently created process and the process that performed the action create a process. After its performance, this action put in the stack the created process identification.

The action sleep suspends the execution of the current process until another process executes the action wakeup. This action gets a suspended process identification from the stack and reactivates it. The action destroy gets a process identification from the stack and removes it from the list of active processes.

The last defined action: wait, gets a tuple of process identifications from the stack and waits until their normal performance or one of them terminates abnormally with a jump. If all processes terminate normally, the values produced by these actions are retrieved by the wait action and propagated to the current action processor. If one of these processes terminates abnormally, the other processes are killed and the wait action terminates abnormally with the same information.

The following action specifies how the concurrent process action can be used to specify an interleaving combinator for the action notation.

```
x and x =
    create a process x ;
    create a process y ;
    apply (_ , _);
    wait
```

To execute two actions in parallel, we created two process to execute each sub-action and we wait until both process finish to allow the program processing continue.

3 Example of Definition

To exemplify the use of the proposed notation in the definition of complex behaviors, we use it to specify the semantics of the action notation functional facet. The functional facet is used to store temporally calculated values and specifies a transient information, which is received and given by actions. The functional facet defines the following actions to handle with transients: the action give e evaluates the expression e and gives the resulted value as transient information; The action combinator x then y executes the sub-action sequentially and all transients given by the first action execution are passed as input for the second action execution; the current transient given by an action can be retrieved using the expression the given x that returns the current transient value if it is of type x.

The semantics of the functional operators can be given in terms of our proposed notation as shown by the following specification:

```
transients : (producer & consumer).
give x =
    |evaluate x
    and then
    |produce transients.
x then y =
    |frame transients x
    and then
    |select transients y.
evaluate the given x =
    |current transients and then apply x
    and then
    |apply _ & _
```

The transient information is defined to behave like both producers and consumers (functional facet produces values to the external environment and receive values from it).

The give action just evaluates the argument and the resulted value is propagated as a transient value. The functional combinator then is modeled like an action that executes the first sub-action and the produced transients are compacted into a tuple that will be passed as transient to the execution of the second action. Finally, the the given x is modeled like an action that retrieves the current transient and checks if it conforms to the expected type.

	Proposed Notation	AN2	MSOS
Calculations	stacks	transient values	OS rules
Kinds of Information	stacks, consumers producers, storage	transients, bindings, storage, messages	computation results, environments, storage
Extensibility	Yes	No	Yes
Number of Operators	20	26	No applicable

Fig. 1. Comparison with Proposed Action Notation and Modular SOS

4 Comparison

The proposed meta-notation can be compared with two recently proposed formalisms: the proposed new version of action notation and modular structural operational semantics. The summary of their comparisons is shown by the Figure 1.

The proposed new version of action notation is an action notation redesign whose main aim is to reduce the notation complexity. It is based on small kernel of primitive operations (about 26 actions) that can be composed to describe the full notation. Comparing it with our proposal, the most notorious difference is that the proposed new version of action notation is based on a fixed set of facets (transients, bindings, storage, messages, etc) and our proposed notation is based on a fixed set of kinds of facets (consumers, producers, jumps). It allows the designer to define new facets and change the current actions behavior.

The modular structured operational semantics (MSOS) extends operational semantics with labeled transitions that are useful to abstract the data flow existing in the specified language. Like our notation, MSOS is based on kinds of facets (environments, storage, etc.) and allows the designer to specify new facets. The difference between both approaches is that MSOS uses operational semantics rules to express the program semantics and our approach combines actions to express it. We think that the last approach is more suitable to build an action semantics based environment because it does not demands the tool support one more formalism (operational semantics rules) to become able to recognize extended notations, simplifying the building of analysis tools.

5 Implementing

At the present moment we are design an implementation for an action semantics compiler generator based on the proposed meta-notation. This compiler generator is planned to be integrated in release 3.0 of the Abaco system [1] to be released.

Besides allow the designer defines your own action extensions set, We think that another big advantage of these models is that the same compilation generator engine will be able to handle action written in the current version of the action notation and the proposal new version. It will avoid we implement specific engines for each version.

6 Conclusions

Using the meta-notation proposed in this paper, we will be able to build more efficient action semantics environments that can be used to test complex kinds of languages. In particular, we used in [2] the ideas of this papers to describe new action combinators to model operations found in logic programming languages and agent programming languages and use them to simplify the descriptions of these languages.

We think that another big advantage of our meta-notation is their simplicity, because each element handles with single and isolated concepts. We think that this feature could be useful to simplify the building of analysis algorithms for action notation and we will try to proposed algorithms for this notation in future research papers.

References

1. Luis Menezes Hermano Moura. The abaco system: An algebraic based action compiler. In *Proceedings of the Second International Workshop on Action Semantics*, number NS-99-3 in BRICS Notes Series, pages 143–154, 1999.
2. Luis Menezes. *Uma Descrição Formal do Paradigma de Programação Orientado a Agentes Utilizando Semântica de Aes*. PhD thesis, Federal University of Pernambuco, 2002.
3. Peter D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
4. Peter D. Mosses. Foundations of modular SOS (extended abstract). In *MFCS'99*, volume 1672 of *LNCS*, pages 70–80, 1999. Full version available at http://www.brics.dk/RS/99/54/.
5. D. A. Watt P. D. Mosses, S. B. Lassen. An introduction to an-2: The proposed new version of action notation. In *Proceedings of the Third International Workshop on Action Semantics*, number NS-00-6 in BRICS Notes Series, pages 19–36, 2000.

A Modular SOS for Action Notation, Revisited

Peter D. Mosses

BRICS & Department of Computer Science
University of Aarhus
Ny Munkegade, bldg. 540
DK-8000 Aarhus, Denmark
pdmosses@brics.dk

Abstract

Action Notation (AN) is a general notation for expressing the actions that are used as semantic entities in Action Semantics. The original version of AN from 1992 (here referred to as AN-1) was described in the Action Semantics book [2], and formally defined by giving a Structural Operational Semantics (SOS) for its kernel, together with definitions of bisimulation and testing equivalences, and some laws that allowed the full AN-1 to be reduced to its kernel. The framework of Unified Algebras was used as a meta-notation for defining AN-1, and features of Unified Algebras were exploited in AN-1 itself.

The SOS of AN-1 was unfortunately not easy to read, nor to work with. The large size of kernel AN-1 (with most primitive actions having so-called yielders as arguments) and the monolithic nature of its SOS were perhaps the major difficulties. As a prelude to the redesign of AN with a smaller, more tractable kernel [1], the author developed a more modular style of SOS [3], and used it to give a reformulated SOS for AN-1 [4].

A draft modular SOS for the new version of AN, referred to as AN-2, has been available via the AN-2 web pages at `http://www.brics.dk/Projects/AS/AN-2.html` since 2000. It is written in CASL (Common Algebraic Specification Language) and has been checked for well-formedness using CATS (CASL Tool Set). It appears to be significantly more accessible than the original SOS of AN-1. However, it now appears that further improvements are possible:

- The description of exceptions and failures can be made completely modular.
- A more perspicuous notation for accessing and changing components of labels is available.

It may also be desirable to avoid the explicit use of CASL, using a more streamlined meta-notation for the modular SOS rules.

After discussing the issues, we look at some illustrative examples taken from an improved modular SOS of AN-2 (in preparation). We also look at the possibility of empirical testing of the modular SOS by a straightforward translation to Prolog.

References

1. S. B. Lassen, P. D. Mosses, and D. A. Watt. An introduction to AN-2, the proposed new version of Action Notation. In *AS 2000*, number NS-00-6 in Notes Series, pages 19–36, BRICS, Dept. of Computer Science, Univ. of Aarhus, 2000. Available also at `http://www.brics.dk/~pdm/papers/LassenMossesWatt-AS-2000/`.
2. P. D. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
3. P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99*, volume 1672 of *LNCS*, pages 70–80. Springer-Verlag, 1999. Full version available at `http://www.brics.dk/RS/99/54/`.
4. P. D. Mosses. A modular SOS for Action Notation (extended abstract). In *AS'99*, number NS-99-3 in Notes Series, pages 131–142, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999. Full version available at `http://www.brics.dk/RS/99/56/`.

The Analysis of Secure Information-Flow in Actions by Abstract Interpretation

Ki-Hwan Choi, Kyung-Goo Doh[1] and Seung Cheol Shin[2]

[1] Hanyang University, Korea
doh@cse.hanyang.ac.kr
[2] Dongyang University, Korea

Abstract

This talk reports our ongoing work on the analysis of secure information-flow in actions.

Information flow analysis is to statically determine how an action's outputs are dependent, directly and indirectly, on its inputs. The analysis is used to certify that an action is secure meaning that the information classified as "secret" is not revealed to unauthorized objects during its flow through the action's performance.

The analysis is defined using the abstract interpretation framework, and is to be proved sound.

Type inference for the new action notation

Jørgen Iversen

BRICS* and Department of Computer Science, University of Aarhus
Ny Munkegade bldg. 540, DK-8000 Aarhus C, Denmark
E-mail: j.iversen@brics.dk

Abstract. Inferring types for actions has shown to be very useful in language design tools and action semantics based compiler generation. It provides the user with useful information about the safety of actions, and enables compiler generators to generate optimising compilers doing transformations based on type annotations. We point out some of the problems with inferring types for the new version of action notation and present an algorithm that solves the problems.

1 Introduction

Inferring types for actions has shown to be very useful in language design tools and action semantics based compiler generation. It provides the user with useful information about the safety of actions (i.e. will type errors cause the action to err when executed), and enables compiler generators to generate optimising compilers doing transformations based on type annotations. There has already been put a lot of effort into this area of research, but the appearance of a new version of action notation (AN-2) prompted us to improve on existing work. This paper presents a type inference algorithm that annotates AN-2 actions with function types. We solve some of the problems that have arisen in connection with the simplification of Action Notation (AN), and we also infer types for a bigger subset of AN, only omitting actions from the communicative facet.

According to the definition of AN all actions are legal, but some of them might always terminate exceptionally or failing. A type error in an action (for instance if a sub-action receives an integer instead of a boolean as expected) will just lead the sub-action to terminate exceptionally, which is completely legal, and the exception can always be trapped. In our notation an action is said to be type correct if every sub-action receives and produces data of the expected type based on the sub-actions it is combined with. Fig. 1 gives some examples of actions containing type errors.

The first action is not type correct, because the right-hand side of the *then* combinator is not given a pair of integers as it expects. In the second action the right-hand side of the action two values, but it is only given one. The second action can terminate normally in two ways, either it performs division by zero,

* Basic Research in Computer Science (http://www.brics.dk), funded by the Danish National Research Foundation

1. *(give truth and give 5) then give (the int#1 + the int#2)*
2. *give the cell then update*
3. *give (the int#1 / the int#2) exceptionally give truth*
4. *bind("x", truth) hence give the int bound to "y"*

Fig. 1. Examples of actions with type errors

raises an exception, traps the exception and gives a boolean value, or it does not perform division by zero and gives an integer value. The problem is that the action must yield data of the same type in both cases to be type correct. In the fourth action the right-hand side of the *hence* combinator does not receive the correct bindings.

Contrary to previous work we shall use the word type instead of sort to emphasise that AN-2 is no longer dependent on unified algebras, and to be more consistent with programming language terminology.

In section 2 we give an overview of previous work in the area, both the work on which we build and other approaches is described. The main reason for our work is the development of AN-2, which is described in section 3. Section 4 is devoted to explaining the algorithm, including the grammar of the types, examples of typing rules and illustration of how unification works. This section also contains a thorough specification of the problems we have solved. In section 5 we report on how the algorithm has been implemented and tested. And finally before the conclusion in section 7 we state the current status of our work and suggest various improvements in section 6.

We assume that the reader has some knowledge of the previous version of action notation.

2 Previous work

Inferring types for actions has been a research area since the beginning of the 1990's, and there have been different approaches to the problem. It is interesting to take a look at the different approaches to the problem and kinds of actions typeable in existing systems.

Even and Schmidt [1] were the first to infer types for actions. Their typing system supports an ML-style type inference algorithm, that does unification on type schemes, which means assigning types to type-variables to obtain equality between two type schemes. The subset of AN handled by the type inferrer only contains actions from the functional and the declarative facet, and they can only terminate normally. Furthermore it does not allow unfolding but it allows abstractions.

In the Cantor system Palsberg [2] chose to restrict the subset of AN handled, so that the languages describable were monomorphic and statically typed, so that type inference could be avoided and code generation would be easier. The actions used are typed and self-referential bindings are not allowed (they could

have been used in a semantics for recursive functions). All in all the type system is more restricted than Even and Schmidt's (and systems based on Even and Schmidt's).

Ørbæk's type checker [3], implemented in the OASIS system, is similar to Palsberg's. One exception is that he allows non-tail-recursive unfoldings, and this means that calculating a fixpoint, when typing unfolding, becomes a bit more complicated.

The work done by Doh and Schmidt [4] seems not to be comparable to other work done in this area, because their objective is to generate a type checker for the source-language and not for the actions, which describes the source-language. This has the advantage that they can give the user better error-messages. On the other hand it might be more useful for compiler generation to type the actions.

In [5] Doh and Schmidt emphasise the use of facets in type inference. They do not improve on the type system, but they claim that their way of specifying it is clearer. Since actions as data (abstractions) and unfolding are not allowed, it becomes impossible to describe interesting programming language features such as procedures and loops, but they are able to keep their type system very simple. Doh allows a bigger subset of AN in the algorithm described in [6], including both unfolding and abstractions, but his analysis of abstractions is still a bit weak. He also combines type-inference with an analysis of statically known data. This is done by using a two level type system describing compile-time types and run-time types.

Brown improved Even and Schmidt's type system and implemented it in the ACTRESS system [7]. The subset of AN used includes most of AN except the communicative facet and escaping actions. His use of action types as function types from pairs of records (transients and bindings) to pairs of records allows a very precise analysis of actions. The essential work of the algorithm lies in unifying records.

The type system used by Lee in his Genesis system [8] is almost the same as Brown's with a few minor improvements, the most important being type inference over tuples of yielders.

The algorithms used by Doh, Schmidt, Palsberg and Ørbæk are comparable because they can all be described as attribute grammars with inherited and synthesised attributes, whereas Even, Schmidt, Brown and Lee's algorithms use a style comparable to the ML type inferrer, which does a bottom-up traversal of the AST while collecting constraints.

3 AN-2

Two years ago Lassen, Mosses and Watt proposed a new version of Action Notation (AN-2) ([9], [10]). The main difference is that the kernel of AN-2 is significantly smaller than the kernel of the previous version of AN. Other interesting features of AN-2 are

- Bindings produced by actions are regarded as computed values and given as transients

– All yielders are expanded to data and data operation application
– Actions can be treated as data directly (no use of abstractions)
– All action combinators are also data operations that work on actions
– Self reference is described by binding actions to special tokens

This should make it easier to develop tools for working with AN that include all of AN, because it is only necessary to look at the kernel. The new notation is not adopted yet, it still needs to be checked whether there are some disadvantages or shortcomings when it comes to practical usage in tools.

4 Overview of the type inference algorithm

The structure of our type inference algorithm (TI) is identical to the one used in Genesis [8]. It has three stages:

1. Annotate the action AST in a bottom-up traversal using type inference rules. During the traversal records are unified and constraints are collected.
2. Solve the constraints. To infer the correct type all the constraints must be solvable. The constraints are solved by unifying types schemes.
3. Reduce types. The types are simplified by applying the global substitution.

Step one is an implementation of the type rules, where meeting the premises gives rise to a recursive call of TI on the sub-actions occurring in the premises. A rule without premises corresponds to an action constant (a leaf in the action AST).

There are around ten different constraints which will be described in later sections. Solving the constraints is done by iterating over the set of constraints trying to solve each one of them individually. If the constraint is solvable, by changing the global substitution (a global mapping from variables to types), it is removed from the set and the global substitution is updated. While satisfying some of the constraints we might assign default types ({} for row variables, datum to type variables) to unbound variables.

Reducing the types removes all type variables bound to other types. This is done by applying the substitution to all variables.

4.1 Type schemes

Fig. 2 shows a grammar of the type schemes used by TI.

Every action is annotated with an action type $((\tau, \beta) \hookrightarrow (\tau', \tau'_e, \kappa))$, which is a function type from transients (τ) and bindings (β) to a pair of transients and a flag. τ' and τ'_e represents the type of data produced by the action in the case where it terminates normally (τ') and exceptionally (τ'_e). The flag (κ) indicates whether the action terminates exceptionally without giving any data (for instance when data operator application goes wrong), if this is the case the flag is err otherwise it is ok (also in cases where the action terminates exceptionally with data). This allows us to infer types for an action like *inspect then raise*,

$$
\begin{array}{lll}
\text{(data type)} & \sigma & ::= \mathsf{nothing} \mid \mathsf{datum} \mid \mathsf{integer} \mid \mathsf{boolean} \mid \mathsf{cell}[\sigma] \mid \mathsf{token}(id^?) \mid \\
 & & \quad \sigma_1 \cup \ldots \cup \sigma_n \mid \theta \mid \mathsf{list}[\sigma] \mid \Gamma \mid \alpha \\
\text{(action)} & \alpha & ::= (\tau, \beta) \hookrightarrow (\tau', \tau'_e, \kappa) \\
\text{(termination)} & \kappa & ::= \mathsf{err} \mid \mathsf{ok} \\
\text{(transients)} & \tau & ::= \Gamma \\
\text{(bindings)} & \beta & ::= \Gamma \\
\text{(record)} & \Gamma & ::= \Gamma_1 \wedge \Gamma_2 \mid \Phi\Psi \mid \mu\rho.\ \Gamma \mid \emptyset \\
\text{(fields)} & \Phi & ::= \{id_1 : \sigma_1, \ldots, id_n : \sigma_n\} \\
\text{(row)} & \Psi & ::= \epsilon \mid \gamma \mid \rho
\end{array}
$$

Fig. 2. Type Schemes

where *inspect* can terminate exceptionally with no data and *raise* terminates exceptionally with a storable. Combining them with *then* gives an action that can terminate exceptionally with and without data. If we didn't have this property, both sides would have to terminate exceptionally giving the same type of data. We are not representing storage in action types, which means that TI can not determine if a cell given to *inspect* exists.

A type similar to this was proposed by Brown in his PhD thesis [7], but he used a simpler type in the Actress system, because it did not allow exceptionally terminating actions.

Both transients and bindings are represented by records, the only difference being that the identifiers in the transients are numbers which means we are representing a tuple by a record. A record can be a concatenation of two records (used when giving a type to the *and* combinators) or it can be a mapping from identifiers to fields together with a row-variable. The row variable is necessary because of the inherent polymorphism in AN, for instance the type of *copy* can be seen as a polymorphic identity function (see app. A rule 2). If the row variable is γ the meaning is that more fields are allowed but not used by the action. ρ means that more fields are allowed and they are propagated or used by the action. If no more fields are allowed the row variable is ϵ. Finally a record can also be the empty record type $\emptyset$. In an action type like $(\{1 : int\}, \gamma) \hookrightarrow (\{1 : bool\}, \emptyset, \mathsf{ok})$ it indicates that the corresponding action can only terminate normally (it can be used in a symmetric way to indicate exceptional termination).

The other types include nothing which is the empty type, usually the result of trying to unify two non-unifiable types. The type datum is the top element of the lattice of types, the union of all types. We also have some type constructors like unions, lists and cells and atomic types like integer and boolean. Notice that records is included among the regular types, which is useful when an action gives a bindings map as output (i.e. *give current bindings*, app. A rule 19). Also the action type is a regular type, because it is allowed (and very useful) for an action to give an action as output.

Among the data types we also find type variables, they serve the purpose of allowing polymorphism on the single fields and not just on the number of fields as it is the case with the row variables.

Compared to previous work we have added some more record types, changed the action type and removed individuals as types. AN-2 is not based on unified algebras so there is no need to look at individuals as types, and moreover this is very uncommon in type systems so we decided to remove them.

4.2 Typing rules

Most of the typing rules in our system are the same as the ones used in Genesis [8]. We have added and removed some rules because we are working on AN-2 and are looking at a bigger subset of AN. The rules looks somewhat different, because the type of an action has changed in two ways: it does not produce any bindings (except as transients) and it considers exceptional termination, which has not been handled in previous work.

Fig. 3 shows an example of a typing rule. The rule says that if A_1 has a type and A_2 has a type then A_1 *and* A_2 has a type. The transients (bindings) given to the two sub-actions should be the same as the transients (bindings) given to the whole action, and the resulting transients is the concatenation of the output of the sub-actions ($\oplus$ is a concatenation operator on record schemes).

$$\frac{\begin{array}{l}\epsilon \vdash A_1 : (\tau, \beta) \hookrightarrow (\tau_1', \tau_e') \\ \epsilon \vdash A_2 : (\tau, \beta) \hookrightarrow (\tau_2', \tau_e') \\ \tau' = \tau_1' \oplus \tau_2'\end{array}}{\epsilon \vdash A_1 \ \mathit{and}\ A_2 : (\tau, \beta) \hookrightarrow (\tau', \tau_e')}$$

Fig. 3. Typing rule for *and* combinator

All the typing rules can be found in app. A.

4.3 Unification

In the typing rules a premise for some of the typings is the equality of two type schemes (for instance the input transients to the two sub-actions of the *and* combinator must be equal). Two ensure this equality we have a unification operator, $unify$. The operator takes two type schemes as argument and returns a new type scheme, while altering the global substitution. Our unification operator is an extended version of the one found in [8] and can be seen in app. B. The two most important extensions define unify on the new record types.

$unify : \sigma \times \sigma \rightarrow \sigma$ (perhaps altering the global substitution)
$unify(\mu\rho_1.\Gamma_1, \mu\rho_2.\Gamma_2) =$
if $(\rho_1 \simeq \rho_2)$ then $\mu\rho_1.\Gamma_1$
else $[\rho_1 \simeq \rho_2;\ fold(unify(unfold(\mu\rho_1.\Gamma_1), unfold(\mu\rho_2.\Gamma_2)))]$
$| \ unify(\mu\rho_1.\Gamma_1, \Gamma_2) =$
if $(\Gamma_1 \simeq \Gamma_2)$ then $\mu\rho_1.\Gamma_1$
else $[\mu\rho_1.\Gamma_1 \simeq \Gamma_2;\ fold(unify(unfold(\mu\rho_1.\Gamma_1), \Gamma_2))]$
$| \ unify(\sigma, \mu\rho.\Gamma) = unify(\mu\rho.\Gamma, \sigma)$

$unify(\sigma, \emptyset) = \sigma$
$unify(\emptyset, \sigma) = \sigma$

The meaning of Ø is the empty record type used to indicate that an action does not terminate exceptionally/normally, and in that context it is clear that any type unified with Ø should return the type (if the second subtree of an action combinator might terminate exceptionally and the first subtree always terminates normally the whole action can terminate normally and exceptionally). Unification of recursive types is inspired by the standard technique as it was described by Kfoury and Pericas-Geertsen in [11]. The term $[t_1 \simeq t_2; f(t_1, t_2)]$ means that the equality relation between types is updated, and that this relation is used in the type expression $f(t_1, t_2)$.

4.4 The ML type inferrer

A nice way to infer types for actions might be to translate an action into an ML expression and then let the ML type inferrer do the job. Let's try to look at an example

bind("y", 4) hence ((give current bindings and give "x") then give bound)

This should not type check since the right-hand side of the *hence* combinator does not receive any bindings of "x" to anything. Under the assumption that actions are translated into functions from transients and bindings to transients, it appears not to be possible for ML's type inferrer to catch the type error in the above action. It would require every identifier to have its own type, and the type of bindings to depend on the identifiers contained in the bindings. But ML does not allow dependent types, so we can not translate actions into ML and let the ML type inferrer infer the types for us.

4.5 The declarative facet

In the declarative facet all action combinators are defined in terms of kernel actions involving combinators from the functional facet, *hence*, *give current bindings* and various data operations. Fig. 4 shows how *moreover* is expanded to kernel notation.

Since TI only works on kernel actions we apply it to the action on the right hand side of the equation. In previous work there was just a single typing rule

A_1 *moreover* A_2 = *(*A_1 *and* A_2*) then give overriding*

Fig. 4. *moreover* expanded to kernel actions

for *moreover*, but now TI has to combine three rules. This is illustrated in fig. 5 (notice that the typing rules has been simplified in this example; exceptional termination is not considered).

$$\frac{\dfrac{\epsilon \vdash A_1{:}(\tau,\beta)\hookrightarrow\tau_1 \quad \epsilon \vdash A_2{:}(\tau,\beta)\hookrightarrow\tau_2 \quad \tau_3 = concat\,\tau_1\tau_2}{\epsilon \vdash A_1 \textit{ and } A_2{:}(\tau,\beta)\hookrightarrow\tau_3} \quad \frac{\epsilon \vdash overriding{:}\tau_3\to\sigma \quad \sigma\&\Gamma\neq\mathsf{nothing}}{\epsilon \vdash give\ overriding{:}(\tau_3,\beta)\hookrightarrow\sigma}}{\epsilon \vdash A_1 \textit{ and } A_2 \textit{ then give overriding} : (\tau,\beta) \hookrightarrow \sigma}$$

Fig. 5. Proof tree for typing moreover

For TI to infer the correct type, τ_3 must be the type of a tuple of two records, and σ must be the type of the record which is the result of overriding the first record with the second. Since TI starts at the leafs of the action AST it cannot know anything about τ_3, because it depends on the type of A_1 *and* A_2. Therefore TI must "postpone" the type assignment to *give overriding*, and it does so by demanding the constraint in fig. 6 to be satisfied. By postponing the handling of overriding hopefully the row-variables will have been assigned record-types.

$$\frac{\begin{array}{c} OverridingConstraint(\rho_1,\rho_2,\rho_3) \\ \epsilon \vdash overriding : \{1:\{\}\rho_1, 2:\{\}\rho_2\} \to \{1:\{\}\rho_3\} \end{array}}{\epsilon \vdash give\ overriding : (\{1:\{\}\rho_1, 2:\{\}\rho_2\}, \{\}\gamma) \hookrightarrow \{1:\{\}\rho_3\}}$$

$$OverridingConstraint(\rho_1,\rho_2,\rho_3) \Leftrightarrow unify(override(\rho_1,\rho_2),\rho_3) \neq \mathsf{nothing}$$

Fig. 6. Overriding-constraint

Similar things have to be done for the data operations *binding*, *bound* and *disjoint-union*, which is used in the expansion of the other yielders and actions from the declarative facet.

4.6 Actions as data

AN-2 allows actions to be used directly as data without any abstraction wrapper. It is also allowed to use action combinators as data operations from actions to actions, and this causes some problems similar to the ones we had with data

operations in the declarative facet. Furthermore the solution seems to be the same. Fig. 7 illustrates how *then* can be used as a data operation.

*(give the datum#1) and (give the action#2) then give _then_*

Fig. 7. *then* used as data operation

When TI tries to annotate *give _then_* with a type, the same problem arises as we saw with the data operations in the declarative facet, TI does not have enough information at this point to assign a meaningful type to this action. Again we postpone the annotation by introducing a constraint (see fig. 8).

$$\frac{\begin{array}{c}\epsilon \vdash \_then\_ : \{1:\sigma_1, 2:\sigma_2\} \rightarrow \{1:\sigma_3\} \\ InfixActionConstraint(\sigma_1, \sigma_2, \sigma_3, then)\end{array}}{\epsilon \vdash give\ \_then\_ : (\{1:\sigma_1, 2:\sigma_2\}, \{\}\gamma) \hookrightarrow \{1:\sigma_3\}}$$

where

$$InfixActionConstraint(\sigma_1, \sigma_2, \sigma_3, ac) \Leftrightarrow \exists A_1, A_2 . \sigma_i \in TI(A_i) \wedge \sigma_3 \in TI(A_1\ ac\ A_2)$$

Fig. 8. Infix-action-constraint

($TI(A)$ means the type assigned to A by our typing algorithm TI). Similar things should be done for *provide_* and prefix-actions.

4.7 Recursion

The problem with action combinators as data operators becomes even worse when we look at actions with recursive bindings. Let's look at the *unfold* action as an example (fig. 9). A pseudo parse tree of the interesting part with some simplified type annotations is shown in fig. 10 (notice that we have made the same simplifications as we did with the example in fig. 5, and furthermore we are using meta-variables to range over records (t and b)). The reader can convince herself of the correctness of the simplified type annotations by following a simple argument like: The output of *give current bindings then give provide_* is an action giving a bindings map that binds "unf" to another action, because the current bindings must contain a binding of "unf" to an action, otherwise we could not type check *give the action bound to "unf"*. The type of the *and*-action has the same input as the two sub-actions, and as output a concatenation of the two sub-actions output. Since the typing rule for the *hence* combinator dictates that the output transients of the left action should contain bindings, which are unifiable with the bindings used by the right action, we have to do the following unification

$$unify(\{unf : (t, b) \rightarrow t'\}, b)$$

We have to make b "equal" to some construct containing b by changing the substitution of type variables. This is only possible if b is a recursive type and $\{unf : (t, b) \rightarrow t'\}$ is an unfolding of this recursive type. Therefore we have introduced recursive types in our type system.

```
unfold = (give the data then give provide_)
         and
           ((give current bindings then give provide_)
           and
           (give current bindings and provide "unf"
           then give bound_ then give the action)
           then give _hence_)
         then give _then_
         then enact
```

Fig. 9. Expansion of unfold

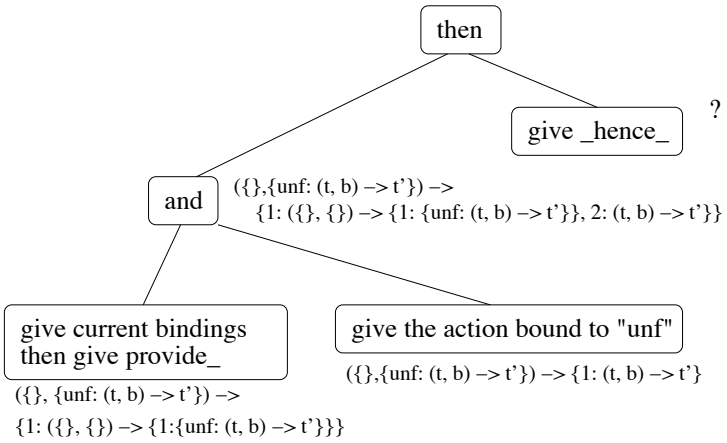

Fig. 10. AST of the interesting part of unfold expanded

4.8 Constraints

During the first stage of the algorithm we collect some constraints. Besides the constraints mentioned in the previous sections, we also have the *Unity* constraint used in connection with concatenation of records. This constraint has been described in [8].

Compared to previous work we collect more constraints to solve in the second stage of the algorithm. The extra generality of our constraints means that the order of solving them is important. We are investigating an optimal ordering.

4.9 Problems

Unfortunately we are not able to infer types for all the actions we would like. Actions describing recursive functions, TI tries to annotate with action types containing recursive record types so complicated, so that when the unification operator is applied to these types it can only return nothing. This result is of course correct, but not very useful because it means that TI must reject a type correct action.

One way of solving it would be to improve TI, and this is our preferred solution. Another solution is to change AN-2. The development of AN-2 is not completed yet, and feedback from using AN-2 may still have an impact on the design of it. Our experiences with using AN-2 could help in improving it, such that constructing tools for working with AN-2 becomes easier. We suggest introducing the *recursively* combinator in the kernel. This would avoid the problems described above. There is also a third solution, which consist in letting the user annotate actions in the semantic functions with types. The annotations would then be a help to TI and would only be necessary in the cases where TI rejects actions that are type correct.

5 Implementation

TI has been implemented in C++. There were many reasons for choosing this language for the implementation.

- Efficiency, the C++-compiler gives a very fast executable.
- Portability, C++ compilers exists for almost any platform.
- Tool support, There is a large set of programming tools and libraries for the language.
- Maintainability, the modularity of C++ makes it easy to maintain large-scale software.

The implementation has been tested on a representative set of examples based on a small imperative language. More exhaustive tests should be performed in the future, involving actions generated from languages supporting other programming paradigms like functional and object oriented. The algorithm should also be run on actions coded by hand, which can test some special cases.

6 Future work

There are several ways of improving TI.

- Allowing a bigger subset of AN-2 to be typeable. As described in section 4.9 there are some actions which will run without problems but which are not accepted by TI. Furthermore we do not handle actions including the communicative facet. We are only allowing statically typeable actions, but it might also be interesting to be able to do something with the dynamically typeable actions.
- Introduce overloaded operators. In some way we allow overloaded operators in TI namely the builtin operators on bindings (*binding*, *bound*, *overriding* and *disjoint-union*). They are overloaded in the way that they operate on many different bindings maps, and the type system regards two bindings maps as different types if they are not equivalent mappings. This might give us a clue of how to implement overloaded operators in general; we could use constraints as we did with the operators on bindings.
- Introduce subtypes. Having subtype relations between the types could give partly the same effect as overloaded operators, for instance instead of having a plus operator on integers and one on naturals we could have a subtype relation between the two types. Sekiguchi and Yonezawa proposed an algorithm for inferring types in a system with subtyped recursive types [12], which we might use as inspiration.

In most of the previous work done in inferring types for actions a proof for the soundness of the algorithms has been given. Proving soundness in this context means proving that the type inferred for an action is consistent with the transients and bindings received and produced by an action. For proving soundness of TI it might be useful to use the same framework as the one Brown used in his thesis [7], in which he relates each type inference rule for an action combinator or constant to the corresponding semantic rule.

Inferring types serves different purposes. First of all it is useful to see whether an action is safe with respect to types. This of course only goes for the statically typeable actions. Secondly if we want to do action transformations like binding elimination as part of compiling actions we need type information for the actions. Our long-term goal is to compile actions, as part of an action semantics based compiler generator, so TI will be used as one of the first modules in our compiler.

7 Conclusion

We have shown how to infer types for AN-2 actions using an algorithm similar to the ML type inferrer. Our work is an improvement and adjustment of the work performed by Brown and Lee in their PhD thesis.

Our improvements are in the following areas: First of all the algorithm now works on the new version of AN, which is not a trivial improvement due to

the enhanced expressibility in AN-2. Secondly we have tried to handle a bigger subset of actions, including every facet except the communicative one.

We conclude that it is possible to infer types for an non-trivial subset of AN-2. Our algorithm has been implemented and it has shown to be useful in practice. At the same time we observe that our work can be improved and that the design of AN-2 could be changed to aid the construction of tools.

Acknowledgements The author is grateful for improvements to drafts of this paper suggested by Peter D. Mosses.

References

1. Susan Even and David A. Schmidt. Type inference for action semantics. In *ESOP'90, Proc. European Symposium on Programming, Copenhagen*, volume 432 of *Lecture Notes in Computer Science*, pages 118–133. Springer-Verlag, 1990.
2. Jens Palsberg. *Provably Correct Compiler Generation*. PhD thesis, daimi, 1992. xii+224 pages.
3. Peter Ørbæk. Analysis and Optimization of Actions. M.Sc. dissertation, Aarhus University, Computer Science Department, September 1993. `ftp://ftp.daimi.aau.dk/pub/empl/poe/index.html`.
4. Kyung-Goo Doh and David A. Schmidt. Extraction of strong typing laws from action semantics definitions. In *ESOP'92, Proc. European Symposium on Programming, Rennes*, volume 582 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1992.
5. Kyung-Goo Doh and David A. Schmidt. The facets of action semantics: Some principles and applications (extended abstract). In *Proceedings of the First International Workshop on Action Semantics*, pages 1–15, 1994.
6. Kyung-Goo Doh. Action transformation by partial evaluation. In *Proceedings of the ACM SIGPLAN Symposium on Partial evaluation and semantics-based program manipulation*, pages 230–240. ACM Press, 1995.
7. Deryck Forsyth Brown. *Sort inference in action semantics*. PhD thesis, Department of Computing Science, University of Glasgow, 1996.
8. K.D. Lee. *Action Semantics-based Compiler Generation*. PhD thesis, Department of Computer Science, University of Iowa, 1999.
9. Peter D. Mosses. AN-2: Revised action notation—syntax and semantics. Available at `http://www.brics.dk/~pdm/papers/Mosses-AN-2-Semantics/`, October 2000.
10. Søren B. Lassen, Peter D. Mosses, and David A. Watt. An introduction to AN-2, the proposed new version of Action Notation. In *AS 2000, Proc. 3rd International Workshop on Action Semantics, Recife, Brazil*, pages 19–36, 2000. Available also at `http://www.brics.dk/~pdm/papers/LassenMossesWatt-AS-2000/`.
11. Assaf Kfoury and Santiago M. Pericas-Geertsen. Type inference for recursive definitions. Technical Report 2000-007, Comp. Sci. Dept., Bosto University, 6, 2000. `http://cs-people.bu.edu/santiago/Papers/Kfo+Per:TR-1999.pdf`.
12. Tatsurou Sekiguchi and Akinori Yonezawa. A complete type inference system for subtyped recursive types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789, pages 667–686. Springer-Verlag, 1994.

A Typing rules

provide *d*

$$\frac{\epsilon \vdash d : \tau; \tau \& \Gamma \neq \mathsf{nothing}}{\epsilon \vdash \text{provide d} : (\{\}\gamma, \{\}\gamma) \hookrightarrow (\tau, \emptyset, \mathsf{ok})} \tag{1}$$

copy

$$\frac{}{\epsilon \vdash \text{copy} : (\{\}\rho, \{\}\gamma) \hookrightarrow (\{\}\rho, \emptyset, \mathsf{ok})} \tag{2}$$

***A*$_1$ then *A*$_2$**

$$\frac{\begin{array}{c}\epsilon \vdash A_1 : (\tau, \beta) \hookrightarrow (\tau', \tau'_e, \kappa_1) \\ \epsilon \vdash A_2 : (\tau', \beta) \hookrightarrow (\tau'', \tau'_e, \kappa_2)\end{array}}{\epsilon \vdash A_1 \text{ then } A_2 : (\tau, \beta) \hookrightarrow (\tau'', \tau'_e, \kappa_1 \vee \kappa_2)} \tag{3}$$

***A*$_1$ and then *A*$_2$**

$$\frac{\begin{array}{c}\epsilon \vdash A_1 : (\tau, \beta) \hookrightarrow (\tau'_1, \tau'_e, \kappa_1) \\ \epsilon \vdash A_2 : (\tau, \beta) \hookrightarrow (\tau'_2, \tau'_e, \kappa_2) \\ \tau' = \tau'_1 \oplus \tau'_2\end{array}}{\epsilon \vdash A_1 \text{ and then } A_2 : (\tau, \beta) \hookrightarrow (\tau', \tau'_e, \kappa_1 \vee \kappa_2)} \tag{4}$$

***A*$_1$ and *A*$_2$**

$$\frac{\begin{array}{c}\epsilon \vdash A_1 : (\tau, \beta) \hookrightarrow (\tau'_1, \tau'_e, \kappa_1) \\ \epsilon \vdash A_2 : (\tau, \beta) \hookrightarrow (\tau'_2, \tau'_e, \kappa_2) \\ \tau' = \tau'_1 \oplus \tau'_2\end{array}}{\epsilon \vdash A_1 \text{ and } A_2 : (\tau, \beta) \hookrightarrow (\tau', \tau'_e, \kappa_1 \vee \kappa_2)} \tag{5}$$

indivisibly A

$$\frac{\epsilon \vdash A : (\tau, \beta) \hookrightarrow (\tau', \tau'_e, \kappa)}{\epsilon \vdash \text{indivisibly A} : (\tau, \beta) \hookrightarrow (\tau', \tau'_e, \kappa)} \tag{6}$$

raise

$$\frac{}{\epsilon \vdash \text{raise} : (\{\}\rho, \{\}\gamma) \hookrightarrow (\emptyset, \{\}\rho, \mathsf{ok})} \tag{7}$$

A_1 exceptionally A_2

$$\frac{\begin{array}{c}\epsilon \vdash A_1 : (\tau, \beta) \hookrightarrow (\tau', \tau'_e, \kappa_1) \\ \epsilon \vdash A_2 : (\tau'_e, \beta) \hookrightarrow (\tau', \tau''_e, \kappa_2)\end{array}}{\epsilon \vdash A_1 \text{ exceptionally } A_2 : (\tau, \beta) \hookrightarrow (\tau', \tau''_e, \kappa_1 \vee \kappa_2)} \tag{8}$$

A_1 and exceptionally A_2

$$\frac{\begin{array}{c}\epsilon \vdash A_1 : (\tau, \beta) \hookrightarrow (\tau', \tau'_{e1}, \kappa_1) \\ \epsilon \vdash A_2 : (\tau, \beta) \hookrightarrow (\tau', \tau'_{e2}, \kappa_2) \\ \tau'_e = \tau'_{e1} \oplus \tau'_{e2}\end{array}}{\epsilon \vdash A_1 \text{ and exceptionally } A_2 : (\tau, \beta) \hookrightarrow (\tau', \tau'_e, \kappa_1 \wedge \kappa_2)} \tag{9}$$

give o

$$\frac{\begin{array}{c}\epsilon \vdash \text{o} : \tau \rightarrow ?\sigma \\ \sigma \& \Gamma \neq \mathsf{nothing}\end{array}}{\epsilon \vdash \text{give o} : (\tau, \{\}\gamma) \hookrightarrow (\sigma, \emptyset, \mathsf{err})} \tag{10}$$

$$\frac{\begin{array}{c}\epsilon \vdash \text{o} : \tau \rightarrow \sigma \\ \sigma \& \Gamma \neq \mathsf{nothing}\end{array}}{\epsilon \vdash \text{give o} : (\tau, \{\}\gamma) \hookrightarrow (\sigma, \emptyset, \mathsf{ok})} \tag{11}$$

give #i

$$\frac{}{\epsilon \vdash \text{give \#i} : (\{1 : \Delta, 2 : \Delta, \cdots, i : \theta\}\gamma, \{\}\gamma) \hookrightarrow (\{1 : \theta\}, \emptyset, \mathsf{ok})} \tag{12}$$

give the σ

$$\frac{unify\, \tau\ \sigma \neq \mathsf{nothing}}{\epsilon \vdash \text{give the } \sigma : (\tau, \{\}\gamma) \hookrightarrow (\tau, \emptyset, \mathsf{ok})} \tag{13}$$

check p

$$\frac{\epsilon \vdash \text{p} : pred\ \tau}{\epsilon \vdash \text{check p} : (\tau, \{\}\gamma) \hookrightarrow (\{\}, \emptyset, \mathsf{err})} \tag{14}$$

fail

$$\frac{}{\epsilon \vdash \text{fail} : (\{\}\gamma, \{\}\gamma) \hookrightarrow (\emptyset, \emptyset, \mathsf{ok})} \tag{15}$$

A_1 otherwise A_2

$$\frac{\begin{array}{c}\epsilon \vdash A_1 : (\tau, \beta) \hookrightarrow (\tau', \tau'_e, \kappa_1) \\ \epsilon \vdash A_2 : (\tau, \beta) \hookrightarrow (\tau', \tau'_e, \kappa_2)\end{array}}{\epsilon \vdash A_1 \text{ otherwise } A_2 : (\tau, \beta) \hookrightarrow (\tau', \tau'_e, \kappa_1 \vee \kappa_2)} \tag{16}$$

select(A_1 or ... or A_n)

$$\frac{\begin{array}{l}\forall i \in 1..n.\ \epsilon \vdash A_i : (\tau_i, \beta_i) \hookrightarrow (\tau'_i, \tau'_e i, \kappa_i) \\ \tau = switch\, \tau_1 \ldots \tau_n \\ \beta = switch\, \beta_1 \ldots \beta_n \\ \tau' = select\, \tau'_1 \ldots \tau'_n \\ \tau'_e = select\, \tau'_{e1} \ldots \tau'_{en}\end{array}}{\epsilon \vdash \text{select}(A_1 \text{ or } \ldots \text{or } A_n) : (\tau, \beta) \hookrightarrow (\tau', \tau'_e, \bigvee_i \kappa_i)} \tag{17}$$

choose natural

$$\frac{}{\epsilon \vdash \text{choose natural} : (\{\}\gamma, \{\}\gamma) \hookrightarrow (\{1 : \mathsf{integer}\}, \emptyset, \mathsf{ok})} \tag{18}$$

give current bindings

$$\frac{}{\epsilon \vdash \text{give current bindings} : (\{\}\gamma, \{\}\rho) \hookrightarrow (\{1 : \{\}\rho\}, \emptyset, \mathsf{ok})} \tag{19}$$

A_1 hence A_2

$$\frac{\begin{array}{c}\epsilon \vdash A_1 : (\tau, \beta_1) \hookrightarrow (\tau'_1, \tau'_e, \kappa_1) \\ \epsilon \vdash A_2 : (\tau, \beta_2) \hookrightarrow (\tau'_2, \tau'_e, \kappa_2) \\ unify\, \tau'_1 \{1 : \beta_2\} \neq \mathsf{nothing}\end{array}}{\epsilon \vdash A_1 \text{ hence } A_2 : (\tau, \beta_1) \hookrightarrow (\tau'_2, \tau'_e, \kappa_1 \vee \kappa_2)} \tag{20}$$

enact

$$\frac{}{\epsilon \vdash \text{enact} : (\{1 : (\{\}, \{\}) \hookrightarrow (\{\}\rho, \emptyset, \mathsf{ok})\}, \{\}\gamma) \hookrightarrow (\{\}\rho, \emptyset, \mathsf{ok})} \tag{21}$$

create

$$\frac{unify(\mathsf{storable}, \theta) \neq \mathsf{nothing}}{\epsilon \vdash \text{create} : (\{1 : \theta\}, \{\}\gamma) \hookrightarrow (\{1 : \mathsf{cell}[\theta]\}, \emptyset, \mathsf{err})} \tag{22}$$

destroy

$$\frac{unify(\mathsf{storable}, \theta) \neq \mathsf{nothing}}{\epsilon \vdash \text{destroy} : (\{1 : \mathsf{cell}[\theta]\}, \{\}\gamma) \hookrightarrow (\{\}, \emptyset, \mathsf{err})} \tag{23}$$

update

$$\frac{unify(\mathsf{storable}, \theta) \neq \mathsf{nothing}}{\epsilon \vdash \text{update} : (\{1 : \mathsf{cell}[\theta], 2 : \theta\}, \{\}\gamma) \hookrightarrow (\{\}, \emptyset, \mathsf{err})} \tag{24}$$

inspect

$$\frac{unify(\mathsf{storable}, \theta) \neq \mathsf{nothing}}{\epsilon \vdash \text{inspect} : (\{1 : \mathsf{cell}[\theta]\}, \{\}\gamma) \hookrightarrow (\{1 : \theta\}, \emptyset, \mathsf{err})} \tag{25}$$

Auxiliary operations

$\wedge$ and $\vee$ are binary operators ($\{\mathsf{err}, \mathsf{ok}\} \times \{\mathsf{err}, \mathsf{ok}\} \rightarrow \{\mathsf{err}, \mathsf{ok}\}$) defined in the following way

A	B	$A \wedge B$
err	err	err
err	ok	err
ok	err	err
ok	ok	ok

A	B	$A \vee B$
err	err	err
err	ok	ok
ok	err	ok
ok	ok	ok

The concatenation operator on records, $\oplus$, is defined as

$$
\begin{array}{l}
\oplus : \Gamma \times \Gamma \rightarrow \Gamma \\
\oplus(\emptyset, \Gamma_j) = \emptyset \\
\oplus(\Gamma_j, \emptyset) = \emptyset \\
\oplus(\Phi_i \rho_i, \Gamma_j) = (unity((\Phi_i \rho_i) \wedge \Gamma_j, \{\}\rho); \{\}\rho) \\
\mid \oplus(\Gamma_i, \Phi_j \rho_j) = (unity(\Gamma_i \wedge (\Phi_j \rho_j), \{\}\rho); \{\}\rho) \\
\mid \oplus(\Gamma_i, \Gamma_j) = (unity(\Gamma_i \wedge \Gamma_j, \{\}\gamma); \{\}\gamma)
\end{array}
$$

where the execution of the operator leads to the creation of a *unity*-constraint. If we try to concatenate something with the empty record type we get back a the empty record type. This corresponds to concatenating the output from an *and*-combinator where one of the subactions doesn't terminate normally.

The rest of the operators used are defined in app. B

B Unify

$unify : \sigma \times \sigma \to \sigma$ (perhaps altering the global subsitution)

(1) $unify(\theta_i, \theta_j) = ([\theta_i \mapsto \theta, \theta_j \mapsto \theta]; \theta)$
(2) $\mid unify(\theta_i, \sigma_j) = ([\theta_i \mapsto \sigma_j]; \theta_i)$
(3) $\mid unify(\sigma_i, \theta_j) = unify(\theta_j, \sigma_i)$

(10) $\mid unify((\sigma_{i_1} | \cdots | \sigma_{i_m}), (\sigma_{j_1} | \cdots | \sigma_{j_n})) =$
let $M = (\sigma_{i_1} | \cdots | \sigma_{i_m})$ & $(\sigma_{j_1} | \cdots | \sigma_{j_n})$
$\sigma'_i = prune(unify(\sigma_{i_1}, M) \mid \cdots \mid unify(\sigma_{i_m}, M))$
$\sigma'_j = prune(unify(\sigma_{j_1}, M) \mid \cdots \mid unify(\sigma_{j_n}, M))$
in
case (σ'_i, σ'_j) of
$(\sigma'_{i_1} | \cdots | \sigma'_{i_g}, \sigma'_{j_1} | \cdots | \sigma'_{j_h}) \Rightarrow |_{k=1,l=1}^{g,h} unify(\sigma_{i_k}, \sigma_{j_l})$
where $\sigma_{i_k} \& \sigma_{j_l} \neq \mathsf{nothing}$
$\mid \quad (\_, \_) \Rightarrow unify(\sigma'_i, \sigma'_j)$
end
(11) $\mid unify((\sigma_{i_1} | \cdots | \sigma_{i_n}), \sigma_j) =$
let $M = (\sigma_{i_1} | \cdots | \sigma_{i_n})$ & σ_j *in*
$unify(\sigma_j, M); \forall k.\, unify(\sigma_{i_k}, M); \forall k.\, unify(\sigma_{i_k}, \sigma_j)$
where $\sigma_{i_k} \& \sigma_j \neq \mathsf{nothing}$;
$prune\ (\sigma_{i_1} | \cdots | \sigma_{i_n})$
end
(12) $\mid unify(\sigma_i, (\sigma_{j_1} | \cdots | \sigma_{j_n})) = unify((\sigma_{j_1} | \cdots | \sigma_{j_n}), \sigma_i)$

(13) $\mid unify(\mathsf{token}(I_1), \mathsf{token}(I_2)) =$ if $(I_1 = I_2)$ then $\mathsf{token}(I_1)$ else $\mathsf{nothing}$
(14) $\mid unify(\mathsf{token}(I_1), \mathsf{token}()) = \mathsf{token}(I_1)$; set identifier of $\mathsf{token}()$
(15) $\mid unify(\mathsf{token}(), \mathsf{token}(I_2)) = \mathsf{token}(I_2)$; set identifier of $\mathsf{token}()$
(16) $\mid unify(\mathsf{token}(.), \sigma) = \mathsf{nothing}$

(17) $\mid unify(\mathsf{cell}[\sigma_1], \mathsf{cell}[\sigma_2]) =$
let $\sigma_3 = unify(\sigma_1, \sigma_2)$ in
if $(\sigma_3 \neq \mathsf{nothing})$ then $\mathsf{cell}[\sigma_3]$
else $\mathsf{nothing}$
end
(20) $\mid unify(\mathsf{cell}[\sigma_1], \sigma_2) = \mathsf{nothing}$

(21) $\mid unify(\mathsf{list}[\sigma_1], \mathsf{list}[\sigma_2]) =$
let $\sigma_3 = unify(\sigma_1, \sigma_2)$ in
if $(\sigma_3 \neq \mathsf{nothing})$ then $\mathsf{list}[\sigma_3]$
else $\mathsf{nothing}$
end
(22) $\mid unify(\mathsf{list}[\sigma_1], \sigma_2) = \mathsf{nothing}$

(27) | $unify(\Gamma, \emptyset) = \Gamma$
(28) | $unify(\emptyset, \Gamma) = \Gamma$

(24) | $unify(\mu\rho_1.\Gamma_1, \mu\rho_2.\Gamma_2) =$
if $(\rho_1 \simeq \rho_2)$ then $\mu\rho_1.\Gamma_1$
else $[\rho_1 \simeq \rho_2;\ fold(unify(unfold(\mu\rho_1.\Gamma_1), unfold(\mu\rho_2.\Gamma_2)))]$
(25) | $unify(\mu\rho_1.\Gamma_1, \Gamma_2) =$
if $(\Gamma_1 \simeq \Gamma_2)$ then $\mu\rho_1.\Gamma_1$
else $[\mu\rho_1.\Gamma_1 \simeq \Gamma_2;\ fold(unify(unfold(\mu\rho_1.\Gamma_1), \Gamma_2))]$
(26) | $unify(\Gamma_1, \mu\rho.\Gamma_2) = unify(\mu\rho.\Gamma_2, \Gamma_1)$

(23) | $unify(\Gamma_1, \Gamma_2) = combine(unify, unifyRow, \Gamma_1, \Gamma_2)$

(4) | $unify((\{\} \wedge \sigma_j), \sigma_k) = unify(\sigma_j, \sigma_k)$
(5) | $unify((\sigma_i \wedge \{\}), \sigma_k) = unify(\sigma_i, \sigma_k)$
(6) | $unify((\Phi_i \wedge \Phi_j), \sigma_k) = unify((\Phi_i \cdot \Phi_j), \sigma_k)$
(7) | $unify((\Phi_i\Psi_i \wedge \Phi_j\Psi_j), \Phi_k) =$
if (length$(\Phi_i \cdot \Phi_j) =$ length(Φ_k) then
$([\Psi_i \mapsto \{\}, \Psi_j \mapsto \{\}]; unify((\Phi_i \cdot \Phi_j), \Phi_k))$
else if$(\Psi_i = \epsilon \wedge \Psi_j \neq \epsilon)$ then
$unify((\Phi_i \cdot \Phi_j)\Psi_j, \Phi_k))$
else
$raise\, concatFailure$
(8) | $unify((\Gamma_i \wedge \Gamma_j), \Gamma_k) = raise\, concatFailure$

(29) | $unify((\tau_1, \beta_1) \hookrightarrow (\tau'_1, \tau'_{e1}, \kappa_1), (\tau_2, \beta_2) \hookrightarrow (\tau'_2, \tau'_{e2}, \kappa_2)) =$
let
$\tau_3 = unify(\tau_1, \tau_2)$
$\beta_3 = unify(\beta_1, \beta_2)$
$\tau'_3 = unify(\tau'_1, \tau'_2)$
$\tau'_{e3} = unify(\tau'_{e1}, \tau'_{e2})$
$\kappa_3 = \kappa_1 \vee \kappa_2$
in
if $(\tau_3 \neq$ nothing$) \wedge \beta_3 \neq$ nothing $\wedge\, \tau'_3 \neq$ nothing $\wedge\, \tau_e 3' \neq$ nothing
then $(\tau_3, \beta_3) \hookrightarrow (\tau'_3, \tau'_{e3}, \kappa_3)$
else nothing
end

(30) | $unify($integer, integer$)$ = integer
(30) | $unify($boolean, boolean$)$ = boolean

(31) | $unify(\sigma_1, \sigma_2) =$ nothing

If a type variable has already been assigned a type then this type must be unified with the type we want to assign to it and the result assigned to the type variable.

unifyRow

$$
\begin{array}{l}
\ \ unifyRow : \Psi \times \Psi \rightarrow \Psi \text{ (perhaps altering the global substitution)} \\
\ \ unifyRow(\epsilon, \epsilon) = \epsilon \\
\mid unifyRow(\epsilon, \rho_j) = ([\rho_j \mapsto \{\}]; \epsilon) \\
\mid unifyRow(\epsilon, \gamma_j) = ([\gamma_j \mapsto \{\}]; \epsilon) \\
\mid unifyRow(\rho_i, \rho_i) = \rho_i \\
\mid unifyRow(\rho_i, \rho_j) = ([\rho_j \mapsto \{\}\rho_i]; \rho_i) \\
\mid unifyRow(\gamma_i, \gamma_i) = \gamma_i \\
\mid unifyRow(\gamma_i, \gamma_j) = ([\gamma_j \mapsto \{\}\gamma_i]; \gamma_i) \\
\mid unifyRow(\rho_i, \gamma_j) = ([\gamma_j \mapsto \{\}\rho_i]; \rho_i) \\
\mid unifyRow(\Psi_i, \Psi_j) = unifyRow(\Psi_j, \Psi_i)
\end{array}
$$

If a row variable has already been assigned a type then this type must be unified with the type we want to assign to it and the result assigned to the type variable. An occurrence check is also performed to see if we should introduce a recursive record type.

combine

$$
\begin{array}{l}
combine\ (op_f, op_r, \Phi_i\Psi_i, \Phi_j\Psi_j) = \\
\ \ \text{let}\{id_1 : \phi'_{1_i}, \cdots, id_n : \phi'_{n_i}\}\Psi'_i = extendRecord((\Phi_j - \Phi_i), \Phi_i\Psi_i) \\
\ \ \ \ \{id_1 : \phi'_{1_j}, \cdots, id_n : \phi'_{n_j}\}\Psi'_j = extendRecord((\Phi_i - \Phi_j), \Phi_j\Psi_j) \\
\ \ \ \ \Phi = \{id_1 : op_f(\phi'_{1_i}, \phi'_{1_j}), \cdots, id_n : op_f(\phi'_{n_i}, \phi'_{n_j})\} \\
\ \ \ \ \Psi = op_r(\Psi'_i, \Psi'_j) \\
\ \ \text{in} \\
\ \ \ \ \Phi\Psi \\
\ \ \text{end}
\end{array}
$$

extendRecord assures that that the records have the same number of fields by assigning new records to the row variables.

The *select* and *switch* operations are similar to the ones used in Lee's thesis [8] and compared to the *unify* operator they return the union of two types schemes, instead of trying to find a substitution, such that the type schemes becomes identical.

Using ASM specification for automatic test suite generation for mpC parallel programming language compiler

A. Kalinov, A. Kossatchev, M. Posypkin, and V. Shishkov

Institute for System Programming of Russian Academy of Sciencies
{ka,kos,posypkin,vova}@ispras.ru

Abstract. The paper presents an approach to automatic compiler test suite generation based on formal language specification. The language specification implemented using ASM formalism is discussed. The practical results for mpC parallel programming language compiler are presented. The advantages and drawbacks of proposed approach are discussed.

1 Introduction

Developing an adequate set of tests also called a **test suite** is an important part of software development process. We faced this problem while working on mpC parallel programming language [9] compiler. The general task was to develop a test suite for checking whether the particular compiler implementation correctly processes the programming language.

In this case study we focus on testing language expressions. mpC provides powerful operators for array-based and parallel computations. That is why mpC expressions are complicated and difficult to implement and require thorough testing. However the proposed technique is also applicable to other parts of the language.

Our approach is a sort of "specification-based testing" [11, 12, 2]. We use Abstract State Machines [6] formalism for modeling mpC expressions semantics. The formal specification is implemented using the ASM-based Montages framework [8] – a new method for giving the semantics of a programming language.

We use the specification for three different purposes:

- **Generating test cases.** mpC specification consists of several Montages. Each montage defines the semantics of a particular abstract syntax tree node. Test programs are generated by combining abstract syntax tree nodes. Incorrect tests are filtered out by the specification. Correct tests constitute the test suite.
- **Generating test oracle.** Executable specification is used for generating trustable output of the given test program. Test oracle compares actual and trustable outputs for a particular test. If results are not identical the verdict is failure.

– **Providing test coverage criteria.** Analysis of the specification coverage allows one to see whether all specification rules were involved while executing the test suite. If the coverage criteria are satisfied then no more test cases are needed, otherwise additional test programs should be added to the test suite.

The paper is organized as follows. Section 2 explains how mpC expressions semantics was defined using Montages. Section 3 overviews the test generation process. Practical results and future work are discussed in sections 4 and 5 respectively.

2 The ASM Specification for mpC Expressions

2.1 Overview of the mpC Language.

mpC is a parallel programming language supporting computations on a variety of parallel platforms ranging from local area networks to high performance supercomputers. mpC language is a strict two-level ANSI C extension.

First level (also called C[]) [5] supports array-based computations in the spirit of FORTRAN 90 [10]. The language introduces special operators for manipulating arrays as a whole.

The **grid** operator is used for addressing array sections. It has the following syntax:

```
expr[l : r : s]
```

where `expr` is an expression of an array or a pointer type and expressions `l`, `r`, `s` are expressions of integral type. The operands must satisfy the conditions: $l \leq r$, $l \leq 0$, $r \leq 0$, and $s > 0$. The result of the expression is a vector (an ordered sequence of objects) v containing $(r - l)/s + 1$ elements, with the value of the i-th element of the vector v being the value of the expression $expr[l + s * i]$. If $e[l + s * i]$ is an address expression, then v_i denotes the same object in the memory as $expr[l + s * i]$. Two or more grid operators applied consequently address sections of multidimensional arrays.

In C[] language unary and binary operators admit vector operands. In this case the operator is applied elementally to vector operands. For instance the following code computes the sum of arrays `a` and `b` elements and stores the result in array `c`:

```
int a[N], b[N], c[N];

...

c[0 : N - 1] = a[0 : N - 1] + b[0 : N - 1];
```

Another feature of C[] is **reduction operators**. The binary operators `+`, `*`, `|`, `&`, `^`, `||`, `&&`, `?>`, and `?<` have corresponding reduction counterparts: `[+]`, `[*]`,

`[|]`, `[&]`, `[^]`, `[||]`, `[&&]`. If `op` is a binary operator admitting operands of type T then corresponding reduction operator `[op]` is applicable to an expression of type "vector of elements of type T". The value of the expression `[op] expr` equals to the value of the expression `(...(v1 op v2) ... vn)`, where `v` is a n-element vector value of `expr` and `vi` denotes its i-th elements.

The following code gives an example of calculating dot product of two vectors:

```
double a[N], b[N], c;
...

c = [+](a[0 : N - 1 : 1] * b[0 : N - 1 : 1]);
```

mpC extends C[] with facilities for parallel computations. mpC expressions could be used for expressing both computations and data exchange between different computing nodes. More information could be found at [9] or [1]. For simplicity, examples in this paper use only vector expressions. However the proposed technique is implemented for all kinds of mpC expressions.

Expressions in mpC have much more sophisticated semantics than in C language. Thus the part of the mpC compiler implementing expressions is rather complex and require thorough testing.

2.2 Abstract State Machines

Abstract State Machine (ASM) is a new and powerful approach to specification of large-scale realistic software and hardware systems. We refer the reader to [6] for a detailed definition.

The state of an Abstract State Machine is given by the collection of functions on an abstract set called **superuniverse**. The basic ASM operation is an **update** which is defined as a function value modification at a given location (set of arguments):

$$f(t_1, \ldots, t_r) := t_{r+1}$$

The ASM is driven by transition rules. The expression above called an **update instruction** is a basic transition rule. More complex transition rules are obtained by recursive application of **sequence** and **conditional** constructors.

Sequence constructor The sequence of rules is a rule. The execution of a sequence of rules is defined as a simultaneous execution of rules comprising the sequence (i.e. all updates defined by the rules take place simultaneously).

Conditional constructor If $g_1, \ldots, g_k$ are Boolean terms and $R_1, \ldots, R_k$ are rules then the following expression is a rule:

$$
\begin{aligned}
&if\ g_1\ then\ R_1\\
&elseif\ g_2\ then\ R_2\\
&\vdots\\
&elseif\ g_k\ then\ R_k\\
&endif
\end{aligned}
$$

If at a given state S guard g_i holds and every g_j with $j < i$ fails then the execution of the rule described above is defined as the execution of the rule R_i.

The pure ASM constructs described above provides sufficient basis for the specification of any system. However in practice pure ASM specification may appear to be too cumbersome. That is why a number of ASM extensions have been introduced. One of those extensions is the XASM [3] language. It enriches ASM with several constructs providing more convenient way for specification of different aspects of the system behavior.

The XASM **do-forall** construct:

$$
\begin{aligned}
&do\ forall\ i\ in\ domain\\
&\ R\\
&enddo
\end{aligned}
$$

executes the rule R for all i from *domain* in parallel. This facility is extremely useful for giving the semantics of data-parallel language constructs such as binary operators and assignments of vector operands in mpC.

2.3 Montages.

Montages [8] are a semi-visual formalism for describing programming language syntax, static and dynamic semantics. Montages have been successfully used for the specification SQL [4], C [7] and other programming languages.

A language specification is given as a collection of Montages, each of which is associated with a production rule. A Montage consists of a production rule, static semantics rule, condition, dynamic semantics rules and a control-flow graph. Condition, static and dynamic semantics rules are written in XASM. Sample Montage for assignment operator is demonstrated on Fig. 1.

The specification of mpC expression semantics was implemented using GemMex – a tool for developing Montages based specifications. The tool produces executable module implementing the interpreter for the specified language.

2.4 The Specification of mpC Expressions

The specification for mpC expressions consists of more then 30 Montages for mpC declarations and operators. In addition to $Value$ and *Addr* attributes which are

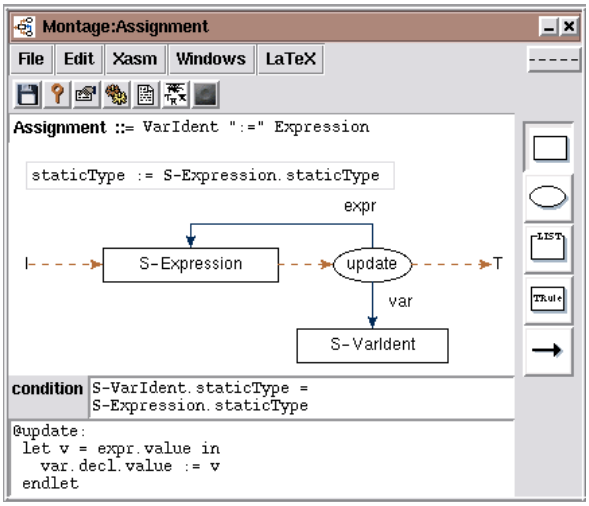

Fig. 1. Montage for assignment operator.

typically used for describing C expressions semantics two new – $VecValue$ and $VecAddr$ are introduced. Those attributes hold the vector value elements and addresses of vector elements respectively.

The XASM code portion below defines the dynamic semantics of a binary operator for the case of vector operands of the same size:

$$
\begin{aligned}
&do\ forall\ i\ in\ set\ \{0..left.Type.Size-1\}\\
&\ self.VecValue(i) = ApplyBinaryOper(S-BinaryOper.Sign,\\
&\ \ left.VecValue(i), right.VecValue(i))\\
&enddo
\end{aligned}
$$

In this example elements of the expression vector value are assigned to the result of applying the binary operator to corresponding elements of operands vector values.

The semantics of mpC expressions is given for AST nodes. The input language for executable specification is a text form of mpC AST representation. The fig. 2 demonstrates mpC expression and corresponding abstract syntax tree in graphical and text forms.

3 Generating the Test Suite from the Specification

The Test Suite Architecture. The test suite consists of mpC programs accompanied with their trustable outputs. A test program contains several initializations of variables involving in testing expression, testing expression itself and the "printf" function call for outputting the expression value.

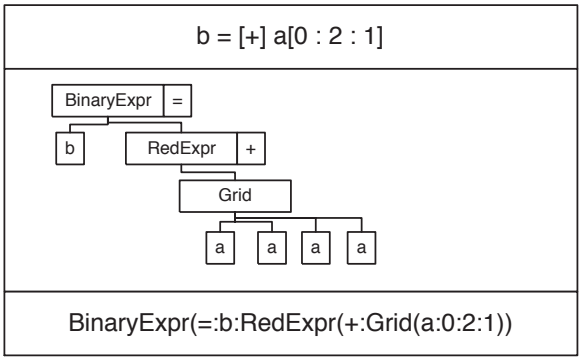

Fig. 2. mpC expression and its AST in graphical and text forms.

The test suite run is organized as follows. First, every program from the test suite is compiled by the compiler under test. Second, obtained binary file is executed to produce **actual output**. Third, actual output is compared with trustable one produced by the specification. If one of the mentioned steps fails the verdict is failure.

The Test Cases Generation Scheme. The proposed scheme of test suite generation is depicted at Fig. 3. We omit some technical details in order to make explanation clear and concise.

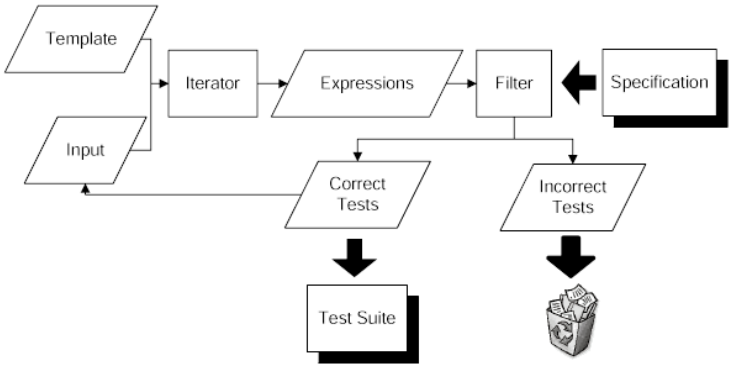

Fig. 3. The scheme of the test suite generation.

The **Iterator** produces syntactically correct mpC expressions. Then produced expressions are processed by the Montages specification and expressions violating semantics constraints are filtered out.

Each expression from the generated set is used for constructing corresponding test program by adding the initialization and output parts to it. Obtained test program is processed by the specification in order to produce trustable output.

The **Iterator** generates test programs from two files: **Template** and **Input**. **Template** is a set of several mpC operators. **Input** is a set of several mpC expressions. The generating is implemented as a substitution of operands from **Input** as operands of operators from **Template**.

Initially **Input** consists of basic expressions like constants and identifiers. Therefore first step of the generating produces only expressions containing one operator. At the second step expressions generated at the first step are used as an **Input** thus allowing to generate expressions containing two operators. The third step uses expressions generated at the second step and so on.

The natural question arises: when to stop? Intuitively it is clear that first step (expressions with only one operator) is not enough. Obvious approach in testing is to use coverage-based heuristics to measure the test suite quality. We consider the coverage of the specification in order to provide implementation-independent test suite adequacy criterion.

We incorporate the coverage tracking into the Filter. It provides the possibility to track the coverage and a possibility to adjust filtering criteria upon coverage.

For the moment we use an update rule coverage [2] to check whether every update rule in static and dynamic semantics parts of each Montage is exercised. Experimental evaluation shows that second step produces the test suite satisfying this coverage criterion.

Obtaining the Test Case. Once the expression is generated we need several steps further to obtain a test case (see Fig. 4).

The first step is the **normalization**. The problem is that the generated expression may have an arbitrary type. If the expression has an arithmetic type the comparison with the trustable output is simple. In either case the normalization may be more difficult. For example if the expression has pointer type the straightforward comparison of values is senseless. The utility called **normalizer** applies several mpC operators to the generated expression in order to obtain the expression of the arithmetic type.

The test program is constructed by accomplishing the normalized expression with necessary declarations and initializations. The test program is processed by the executable specification to obtain the trustable output.

The final step in constructing the test case is the restoration of mpC source code from the AST test program. It is performed by the utility called **restorer**.

Example. Here we present a sample run of the generating scheme. Consider following Input and Template files:

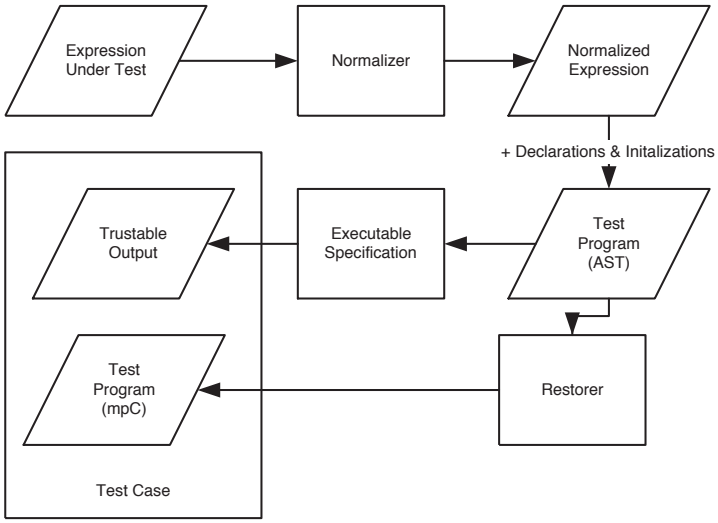

Fig. 4. Constructing the test case.

File	*Contents*	*Semantics*
Input	`s` `Grid(I:0:2:1)`	the variable of structure type and the expression `I[0:2:1]`
Template	`BinaryExpr(+ : $1 , $2)`	the binary operator

Tokens $1 and $2 denotes positions for substituting strings from the Template file. Having 2 entries in Input file and 2 places for substituting in Template file the Iterator produces 4 combinations:

```
BinaryExpr(+ : s : s),
BinaryExpr(+ : s : Grid(I:0:2:1)),
BinaryExpr(+ : Grid(I:0:2:1) : s),
BinaryExpr(+ : Grid(I:0:2:1) : Grid(I:0:2:1)).
```

Since the "+" operator doesn't admit operators of structure type the only semantically valid expression is `BinaryExpr(+ : Grid(I:0:2:1) : Grid(I:0:2:1))`, three remaining expressions are filtered out.

The first iteration produces one test case based on `BinaryExpr(+ : Grid(I:0:2:1) : Grid(I:0:2:1))` and the second iteration Input file consists of three entries:
`s`,
`Grid(I:0:2:1)`,
`BinaryExpr(+ : Grid(I:0:2:1) : Grid(I:0:2:1))`.

The listing below demonstrates the resulting mpC test program obtained at the first iteration:

```
#include<stdio.h>
```

```
main(){
typedef int tInt;
typedef tInt*:(1) tPointer;
typedef struct {
tInt f;
tInt g;
} tStruct;
typedef tInt tArrInt[(3):(1)];
typedef tPointer tArrPointer[(3):(1)];
typedef tStruct tArrStruct[(3):(1)];
typedef tArrInt tArrArrInt[(3):(1)];
tInt i;
tPointer p;
tStruct s;
tArrInt I;
tArrPointer P;
tArrStruct S;
tArrArrInt II;
tInt Result;
(i)=(1);
(p)=(I);
((s).f)=(1);
((s).g)=(1);
((I)[(0):(2):(1)])=(i);
((P)[(0):(2):(1)])=(p);
((S)[(0):(2):(1)])=(s);
(((II)[(0):(2):(1)])[(0):(2):(1)])=(i);
printf("%d\n",([+](((I)[(0):(2):(1)])+((I)[(0):(2):(1)]))));
}
```

4 Practical Results

The proposed technique has been successfully applied to mpC compiler testing. The initial **Template** contains all mpC operators. The first and second steps of the test suite generation produces 135 and 13473 test cases respectively. The following table presents the results of testing for both test suites:

	1st Step	2nd Step
No Errors	47	1007
Static Semantics	51	7271
Code Generating	30	3995
Segmentation Fault	6	1138
Result Mismatch	1	60
Run-time Error	0	2

Analysis of failed tests shows that there are 11 distinct errors in the compiler under test. Step 2 introduces a new kind of errors (run-time error) and new errors of existing kinds. This confirms the intuitive idea that test suite consisting of expressions with only one operator is not sufficient for comprehensive testing.

The advantage of the proposed approach is that the test suite generator is obtained for the price of almost nothing. The formal specification is a useful thing itself: the specification for mpC expressions discovered a lot of inconsistent places in the language specification as well as bugs in the compiler. In the test generating process we reuse the formal specification three times: for oracle, for filtering and for coverage tracking. The only thing we developed specifically for test generator is a set of simple scripts implementing the scheme depicted at Fig. 3.

5 Future Work

The bottleneck of the proposed technique is a huge amount of tests. For example in our case second step consumed 63 hours on a 1GHz Linux workstation. The estimated time for the third step is approximately one year. The main time-consuming part of the test suite generation is a run of the specification for filtering out incorrect test cases.

For typical language a very small percentage of syntactically correct programs are also semantically correct. Thus we can significantly reduce the time of tests generation by providing more "intelligent" **Iterator** producing less amount of semantically incorrect tests. Currently we are working on more complex **Iterator** relying not only on syntactic but on semantics structure of the language also.

Another direction of future research is developing more elaborate coverage notion for Montage specification. For the moment we use update rule coverage – a weakest of all possible coverage measures for ASM-based specification. We plan to consider other coverage measures for both dynamic and static semantics parts of the specification.

Since Montages specifications are based on BNF representation of the language syntax from one hand and ASM specification of the language semantics it seems to be reasonable to combine grammar coverage and ASM-coverage metrics to obtain integral coverage measure.

Besides testing compiler on correct input it is very important to check whether compiler processes semantics errors properly, i.e. generates adequate error report. We plan to develop an efficient technique for handling not only correct but also incorrect test cases which are filtered out for the moment (see Fig. 3).

Acknowledgments. We would like to thank Philipp Kutter and Mathias Anlauff for excellent assistance with Gem-Mex tool and Dr. A. Petrenko who inspired this work in ISPRAS.

References

1. www.ispras.ru/∼ mpc.

2. A. Gargantini and E. Riccobene. ASM-based Testing: coverage criteria and automatic tests generation. In *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*, pages 262–265, February 2001.
3. M. Anlauff. XASM – An Extensible, Component-Based Abstract State Machines Language. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer-Verlag, 2000.
4. B. DiFranco. Specification of ISO SQL using Montages. Master's thesis, Università di l'Aquila, 1997.
5. Sergey Gaissaryan and Alexey Lastovetsky. ANSI C superset for vector and superscalar computers and its retargetable compiler. *Journal of C Language Translation*, 5, 1994.
6. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
7. J. Huggins and W. Shen. The Static and Dynamic Semantics of C. Technical Report CPSC-2000-4, Kettering University, Computer Science Program, 2000.
8. P. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.
9. Alexey Lastovetsky, Dmitrij Arapov, Alexey Kalinov, and Ilya Ledovskih. A parallel language and its programming system for heterogeneous networks. *Concurrency: Practice and Experience*, 12:1317 – 1343, 2000.
10. M. Metcalf and J. Ried. *Fortran 90 Explained*. Oxford University Press, 1992.
11. A. Petrenko. Specification Based Testing: Towards Practice. In *Proceedings of "Perspectives of System Informatics"*, volume 2244 of *LNCS*, pages 287–300, 2001.
12. W. Grieskamp and Y. Gurevich and W. Schulte and M. Veanes. Testing with Abstract State Machines. In *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*, pages 257–261, February 2001.

Computation and Specification Models: A Comparative Study*

Egon Börger

Università di Pisa, Italy
boerger@di.unipi.it

Abstract. For each of the principal current models of computation and of high-level system design, we present a uniform set of transparent easily understandable descriptions, which are faithful to the basic intuitions and concepts of the investigated systems. Our main goal is to provide a mathematical basis for the technical comparison of established models of computation which can contribute to rationalize the scientific evaluation of different system specification approaches in the literature, clarifying in detail their advantages and disadvantages. As a side effect we obtain a powerful yet simple new conceptual framework for teaching the fundamentals of computation theory.

1 Introduction

The presentation of this work in the *Action Semantics* workshop started from Peter Mosses' question to compare Action Notation (AN) [46] and Abstract State Machines (ASMs) [27]. Answering that question naturally led to a broader investigation, namely a comparative analysis of current specification and computation systems in terms of ASMs.

In this paper we assume the reader to know the definition of AN and of the notion of ASM. The main difference between the Action Semantics framework and the ASM method is their different goal. Action Notation has been tailored to support the development of programming languages. The notion of ASMs has been equipped with a general purpose method for high-level hardware and software system analysis and design and its stepwise refinement to code. There is also a difference in the origin of AN and ASMs which shaped the two approaches. AN was developed aiming at enriching denotational features with practically useful operational ones. In an attempt to overcome pragmatically dissatisfactory aspects of a purely denotational approach, primitive and composed actions were directly reflected in close correspondence to programming concepts (semantic mapping of abstract syntax trees to predefined actions) and led to a compromise between competing language development requirements, corresponding to views of the designer, the implementer and the programmer. Gurevich's foundational

* A preliminary version has been presented under the title *Definitional Suggestions for Computation Theory* to the Dagstuhl Seminar on "Theory and Application of Abstract State Machines", Schloss Dagstuhl, March 4-8, 2002.

concern to sharpen the Church-Turing thesis [39] led to an arguably most general notion of virtual machine which became the mathematical basis of the broad-spectrum high-level ASM method for practical system design and analysis [17].

The differences in origin and goal explain also the major technical differences in the realization of AN and ASMs. Actions in AN categorize what in ASMs comes as abstract, a priori unclassified function updates and declarations. Three parameters, organized into so-called facets, serve as basis for the classification: a) different computational aspects, b) types of effect propagation of actions, and c) types of action performance. The *basic facet* covers fundamental control patterns like sequentiality, parallelism, non-determinism; data storage phenomena are dealt with in the *functional facet* in case they are transient between actions, or in the *imperative facet* if they are stable in cells; the *communicative facet* describes interactions between distributed agents; scope information is treated in the *declarative facet*. Most of these features are not directly available in ASMs, though they are definable in a natural way (see [27]). Furthermore AN aims at the generation of a tool environment from language specifications, e.g. the semantics-directed generation of interpreters, compilers, etc., whereas ASMs support a general-purpose method which covers all system design and analysis aspects. In fact ASMs have been specialized to provide an executable semantics for AN, see [6] which contains also further details on tailoring ASMs to fit the AN framework.

In the rest of this paper we use ASMs as a framework for a comparative analysis of other specification and computation systems, comprising the following ones:

- UML Diagrams for System Dynamics
- Classical Models of Computation
  - Automata: Moore-Mealy, Stream-Processing FSM, Co-Design FSM, Timed FSM, PushDown, Turing, Scott, Eilenberg, Minsky, Wegner
  - Substitution systems: Thue, Markov, Post
  - Tree computations: backtracking in logic and functional programming, context free grammars, attribute grammars, tree adjoining grammars
  - Structured and functional programming
    - Programming constructs: seq, while, case, alternate, par
    - Gödel-Herbrand computable functions: Böhm-Jacopini Theorem
    - Recursion
- Specification and Computation Models for System Design
  - Executable high-level design languages: UNITY, COLD
  - State-based specification languages
    - distributed (Petri Nets)
    - sequential: VDM, Z, B
  - Virtual machines
  - Logic-based modeling systems
    - axiomatic systems: denotational, algebraic
    - process algebras (CSP, LOTOS, etc.)

2 Motivation

Since we will use Abstract State Machines (ASMs) as modeling framework, a question to answer before proceeding is why we do not use the proof for the synchronous parallel version of the ASM thesis which claims a form of computational universality for ASMs. The general thesis, as formulated in 1985 by Gurevich in a note to the American Mathematical Society [39], reads as follows (where dynamic structures stand for what nowadays are called ASMs):

> Every computational device can be simulated by an appropriate dynamic structure—of appropriately the same size—in real time

For the synchronous parallel case of this thesis Blass and Gurevich [11] discovered postulates from which every synchronous parallel computational device could be proved to be simulatable in lock-step by an appropriate ASM. Why are we not satisfied with the ASMs constructed by this proof?

The answer has to do with the price to be paid for *proving* computational universality from abstract postulates which cover a great variety of systems. On the one side, the ASM method emphasizes to model algorithms and systems *closely and faithfully, at their level of abstraction*, laying down the essential computational ingredients completely and expressing them directly, without using any encoding which is foreign to the computational device under study. On the other side, if one looks for a mathematical argument proving from explicitly stated assumptions the computational universality of ASMs as claimed in the thesis, some generality in stating the postulates is unavoidable, to capture the huge class of data structures and of the many ways they can be used in a basic computation step, which for every proposed concrete system have to be derived (*decoded*) from the postulates.

The construction by Blass and Gurevich in op.cit., which associates to every synchronous parallel computational system an ASM simulating the system step-by-step, depends in fact on the way the abstract postulates capture the amount of computation (by every single agent) and of the communication between the synchronized agents which is allowed in a synchronous parallel computation step. The necessity to uniformly unfold arbitrary concrete basic parallel communication and computation steps from the postulates as a matter of fact yields some encoding overhead, to guarantee for *every* computational system which possibly could be proposed a representation by the abstract concepts of the postulates. As side effect of this— epistemologically significant—generality of the postulates, the application of the general transformation scheme to established models of computation may yield ASMs which are more involved than necessary and may blur features which really distinguish different concrete systems.

Furthermore, postulating by an existential statement e.g. that states are appropriate equivalence classes of structures of a fixed signature (in the sense of logic), that evolution happens as iteration of single steps, that the single-step exploration space is bounded (i.e. that there is a uniform bound on memory locations basic computation steps depend upon, up to isomorphism), does not by itself provide, for a given computation or specification model, a standard

reference description of its characteristic states, of the objects entering a basic computation step, and of the next-step function. In addition no proof is known to include distributed systems.

Our goal is that of *naturally* modeling systems of specification and computation, based upon a careful analysis of the characteristic conceptual features of each of them. We look for ASM descriptions for each established model of computation or of high-level system design which

- for every framework directly reflect the basic intuitions and concepts, by gently capturing the basic data structures and single computation steps which characterize the investigated system,
- are formulated in a way which is uniform enough to allow explicit comparisons between the classical system models,
- include asynchronous distributed systems.

By deliberately keeping the ASM model for each proposed system as close as possible to the original usual description of the system, so that it can be recognized to be simulated faithfully and step by step by the ASM model, we provide for the full ASM thesis a strong pragmatic argument which

- avoids a sophisticated existence proof for the ASM models from abstract postulates,
- avoids decoding of concrete concepts from abstract postulates,
- avoids a sophisticated proof to establish the correctness of the ASM models.

Since despite of listening carefully to the specifics of each investigated system and of tailoring the simulating ASM models accordingly we can achieve a certain uniformity, we provide a mathematical basis for technical comparison of established system design approaches which we hope will

- contribute to rationalize the scientific evaluation of different system specification approaches, clarifying their advantages and disadvantages,
- offer a powerful yet simple framework for teaching computation theory, unraveling the basic common structure of the myriad of different machine concepts which are studied in computation theory.

3 UML Diagrams for System Dynamics

For the modeling purpose, we use a generalization of Finite State Machines (FSMs) to a class of Abstract State Machines (ASMs) which have been introduced in [15] under the name of control state ASMs and are tailored to UML diagram visualizable machines. A *control state ASM* is an ASM whose rules are all of the following form:

if $ctl\_state = i$ **then**
  if $cond_1$ **then**
    $rule_1$
    $ctl\_state := j_1$
      ...
  if $cond_n$ **then**
    $rule_n$
    $ctl\_state := j_n$

In a given control state i, these machines do nothing when no condition $cond_k$ is satisfied; otherwise for every $cond_k$ which is satsfied, $rule_k$ is executed and the control state changes to j_k so that usually the conditions are supposed or guaranteed to be disjoint to avoid conflicting updates which would stop the ASM computation. Control state ASMs represent a normal form for UML activity diagrams (see[18]) from where they inherit the graphical representation of control states by circles or by (possibly named) directed arcs, to visually distinguish the control-passing role of control states from that of the update actions concerning the underlying data structure which are expressed by the ASM rules inscribed into rectangles, separated from the rule guards written into rhombs. Control state ASMs thus offer to use arbitrarily complex parallel (synchronized) data structure manipulations below the main control structure of finite state machines. The resemblance to FSMs is reflected by the following notation:

FSM(i, **if** $cond$ **then** $rule$, j) =
  if $ctl\_state = i$ **and** $cond$ **then**
    $rule$
    $ctl\_state := j$

so that the control state ASM rule above becomes the set of rules FSM(i, **if** $cond_k$ **then** $rule_k, j_k$) for $k = 1, \ldots, n$.

When writing ASMs M we will use below the distinction between functions which are *controlled* by M (meaning that they are updated by rules of M and not by the environment) and those which are *monitored* by M (i.e. updated only by the environment, but read by M) or shared (i.e. updatable and readable by both M and the environment).

This intuitive understanding of control state ASMs and of different function types suffices for most of the machines defined in this paper. Otherwise we will state what more is needed. For a detailed textbook definition of these machines and of their synchronous or asynchronous multi-agent version we refer the reader to [27].

4 Classical Models of Computation

We show here that the classical automata and substitution systems, ranging from FSMs to computationally universal systems including the structured and the functional programming approaches, are all natural variations of classes of (mostly control state) ASMs. Introducing those formalisms as ASMs, as we do in our lectures on computation theory, avoids having to redefine the semantics of such systems again and again for each variation of the underlying concept of algorithm. This generalizes the uniform semantical frame underlying Scott's definitional suggestions for automata theory [52]. In this section we suppose the reader to know the basic concepts of computation theory (see any textbook on the subject, e.g. [14]).

4.1 Automata

We model here classical automata concepts, computationally universal ones as well as restricted machines.

Finite State Machines The standard FSMs, also known as Mealy automata, are control state ASMs whose rules have the following form with a dynamic input function *in* and a dynamic output function *out*:

$$\textsc{Fsm}(i, \textbf{if } in = a \textbf{ then } out := b, j)$$

in, out usually range over letters a, b, but one may also have words or other value types and also sets or sequences of input or output lines (ports), like in networks of finite automata [28].

The subclass of Moore automata is characterized by the same form of rules but with *skip* instead of the output assignment. This give rise to the generalization to *Mealy/Moore ASMs* defined in [15], a subclass of control state ASMs where the emission of output is replaced by arbitrary ASM rules:

$$\textsc{MealyAsm} = \textsc{Fsm}(i, \textbf{if } in = a \textbf{ then } rule, j).$$

MealyAsms appear for example as components of *co-design FSMs* where rules are needed to compute arbitrary combinational (external and instantaneous) functions. Co-design FSMs are used in [44] for high-level architecture design and specification and for a precise comparison of current models of computation. Other examples of MealyAsms will be shown below.

If one prefers to write FSM programs in the usual tabular form, with one entry (i, a, j, b) for every instruction "in state i reading input a, go to state j and print output b", one obtains the following guard-free Mealy FSM rule scheme for updating (ctl_state, out). The parameters $Nxtctl, Nxtout$ are the two projection functions which define the program table, mapping 'configurations' (i, a) of control state and input to the next control state j and next output b.

$\textsc{MealyFsm}(Nxtctl, Nxtout) =$
  $ctl\_state := Nxtctl(ctl\_state, in)$
  $out := Nxtout(ctl\_state, in)$

Since the input functions in are monitored, they are not updated in the rule scheme, though one can certainly make them shared, e.g. to formalize an input tape which is scanned piecemeal say from left to right.

1-way automata are turned into 2-way automata by including into the instructions also $Moves$ of the input head (say on the input tape), yielding additional updates of the $head$ position and a refinement of in to $in(head)$ (the input portion seen by the new reading head):

$\textsc{TwoWayFsm}(Nxtctl, Nxtout, Move) =$
  $ctl\_state := Nxtctl(ctl\_state, in(head))$
  $out := Nxtout(ctl\_state, in(head))$
  $head := head + Move(ctl\_state, in(head))$

Non-deterministic versions of FSMs, as well as of all the machines we consider below so that there we will only mention deterministic machine versions, are obtained by placing the rules $R_1, \ldots, R_m$ to be chosen from under the **choose** operator, obtaining ASMs with rules of the following form:

choose $R \in \{R_1, \ldots, R_m\}$ **in** R.

Stream Processing FSMs Stream processing FSMs are a specialization of FSMs to machines which compute stream functions $S^m \to S^n$ over a data set S (typically the set $S = A^*$ of finite or $S = A^N$ of infinite words over a given alphabet A), yielding an output stream out resulting from consumption of the input stream in. Non-deterministically in each step these automata

- read (consume) at every input port a prefix of the input stream in,
- produce at each output port a part of the output stream out,
- proceed to the next control state ctl_state.

This can be captured by introducing into the $\textsc{MealyFsm}$ model two choice-supporting functions $Prefix\colon Ctl \times S^m \to PowerSet(S^m_{fin})$, yielding sets of finite prefixes among which to choose for given control state and input stream, and $Transition\colon Ctl \times (S^m_{fin}) \to PowerSet(Ctl \times S^n_{fin})$ describing the possible choices for the next control state and the next finite bit of output. The rule extension for stream processing FSMs is then as follows, where input consumption is formalized by deletion of the chosen prefix from the shared function in:

$\textsc{StreamProcessingFsm}(Prefix, Transition) =$
  choose $pref \in Prefix(ctl\_state, in)$
    choose $(c, o) \in Transition(ctl\_state, pref)$
      $ctl\_state := c$
      $out := concatenate(o, out)$
      $in := delete(pref, in)$

In [42] these machines are used to enrich the classical networks of stream processing FSMs (stream processing components communicating among each other via input/output ports) by ASM state transformations of individual components.

Timed Automata In timed automata [5] letter input comes at a real-valued occurrence time which is used in the transitions where clocks record the time difference of the current input with respect to the previous input:

$$time_{\Delta} = occurrenceTime(in) - occurrenceTime(previousIn).$$

Firing of transitions may be subject to clock constraints and includes clock updates (resetting a clock or adding to it the last input time difference). Typically the constraints are about input to occur within $(<, \leq)$ or after $(>, \geq)$ a given (constant) time interval, leaving some freedom for timing runs, i.e. choosing sequences of $occurrenceTime(in)$ to satisfy the constraints. Thus timed automata can be modeled as control state ASMs where all rules have the following form:

TimedAutomaton($Constraint, Reset$) =
  Fsm(i, **if** $TimedIn(a)$ **then** $ClockUpdate(Reset), j$)
  where
    $TimedIn(a) = (in = a$ **and** $Constraint(time_{\Delta}) = true)$
    $ClockUpdate(Reset) =$
      forall $c \in Reset$ **do** $c := 0$
      forall $c \notin Reset$ **do** $c := c + time_{\Delta}$

Push Down Automata In pushdown automata the Mealy automaton 'reading from the input tape' and 'writing to the output tape' is extended to reading from input and/or a *stack* and writing on the *stack*. Since these machines may have control states with no input-reading or no stack-reading, pushdown automata are control state ASMs with rules of one of the following forms and the usual meaning of the *stack* operations $push, pop$ (optional items are enclosed in []):

PushDownAutomaton =
  Fsm(i, **if** $Reading(a, b)$ **then** $StackUpdate(w), j$)
  where
    $Reading(a, b) = [in = a]$ **and** $[top(stack) = b]$
    $StackUpdate(w) \;=\; stack := push(w, [pop](stack))$

Turing-like Automata Writing pushdown transitions in tabular form

PushDownAutomaton($Nxtctl, Write$) =
  $ctl\_state := Nxtctl(ctl\_state, in, top(stack))$
  $stack := Pop\&Push(stack, Write(ctl\_state, in, top(stack)))$

identifies the 'memory refinement' of FSM *in*put and *out*put tape to *in*put and *stack* memory. The general scheme becomes explicit with Turing machines which combine *in*put and *out*put into one *tape* memory with moving *head*. All the *Turing-like machines* we mention below are control state ASMs which in each step, placed in a certain *pos*ition of their *mem*ory, read this *mem*ory in the *env*ironment of that *pos*ition and react by updating mem and pos. Variations of these machines are due to variations of mem, pos, env, whereas their rules are all of the following form:

$\textsc{TuringLikeMachine}(mem, pos, env) =$
$\textsc{Fsm}(i, \textbf{if } Cond(mem(env(pos)) \textbf{ then update } (mem(env(pos)), pos), j)$

For the original *Turing* machines this scheme is instantiated by $mem = tape$ containing words, integer positions $pos\colon Z$ where single letters are retrieved, $env = identity$, $Write$s in the position of the *tape head*. This leads to extending the rules of TwoWayFsm as follows (replacing in by $tape$ and $Nxtout$ by $Write$):

$\textsc{TuringMachine}(Nxtctl, Write, Move) =$
$ctl\_state := Nxtctl(ctl\_state, tape(head))$
$tape(head) := Write(ctl\_state, tape(head))$
$head := head + Move(ctl\_state, tape(head))$

The extension of the 1-tape Turing machine to a k-tape and to an n-dimensional TM results from data refining the 1-tape Turing *mem*ory and the related operations and functions. Register machines are a data refined instance of k-tape Turing machines ([14, Ch.AI1]).

Scott [52] and Eilenberg [32] instead of read/write operations on words stored in a tape provide data processing for arbitrary data, residing in abstract *mem*ory, by arbitrarily complex global mem-transforming functions. Eilenberg's *X-machines* (and similarly their stream processing version) can be modeled as instances of Mealy ASMs whose rules in addition to yielding *out*put also update mem via global memory functions f (one for each input and control state):

$\textsc{XMachine} = \textsc{Fsm}(i, \textbf{if } in = a \textbf{ then } \{out := b, mem := f(mem)\}, j)$

The global *mem*ory *Action*s of *Scott machines* together with their standard $IfThenElse$ control flow, directed by global memory $Test$ predicates, yield control state ASMs consisting of rules of the following form:

$\textsc{ScottMachine}(Action, Test) =$
$ctl\_state := IfThenElse(ctl\_state, Test(ctl\_state)(mem))$
$mem := Action(ctl\_state)(mem)$

Interacting Turing Machines Wegner's interactive Turing machines [54] in each step can receive some input from the environment and yield output to the environment. Thus they simply extend the TuringMachine by an additional *input* parameter and an *output* action

TuringInteractive($Nxtctl, Write, Move$) =
  $ctl\_state := Nxtctl(ctl\_state, tape(head), input)$
  $tape(head) := Write(ctl\_state, tape(head), input)$
  $head := head + Move(ctl\_state, tape(head), input)$
  $output(ctl\_state, tape(head), input)$

Considering the output as written on an in-out tape comes up to define $output := concatenate(input, Out(control, tape(head), input))$ as the output action using a function Out defined by the program. Viewing the input as a combination of preceding inputs/outputs with the new user input comes up to define $input$ as a derived function $input = combine(output, user\_input)$ depending on the current $output$ and $user\_input$. The question of single-stream versus multiple-stream interacting Turing machines (SIM/MIM) is only a question of instantiating input to a stream vector $(inp_1, \ldots, inp_n)$.

Substitution Systems Replacement systems à la Thue, Markov, Post are Turing-like machines operating over mem: A^* for some finite alphabet A with a finite set of word pairs (v_i, w_i) where in each step one occurrence of a 'premisse' v_i in mem is replaced by the corresponding 'conclusion' w_i. The difference between *Thue systems* and *Markov algorithms* is that Markov algorithms have a fixed scheduling mechanism for choosing the replacement pair and for choosing the occurrence of the to be replaced v_i. In the semi-Thue ASM rule below we use $mem([p, q])$ to denote the subword of mem between the p-th and the q-th letter of mem, which *match*es v if it is identical to v. By $mem(w/[p, q])$ we denote the result of substituting w in mem for $mem([p, q])$. The non-determinism of Thue systems is captured by two selection functions.

ThueSystem($ReplacePair$) =
  let $(v, w) = select_{rule}(ReplacePair)$
  let $(p, q) = select_{sub}(mem)$
    if $match(mem([p, q]), v)$ **then** $mem := mem(w/[p, q])$

The Markov ASM is obtained from the Thue ASM by a pure data refinement, instantiating $select_{rule}(ReplacePair, mem)$ to yield the first $(v, w) \in ReplacePair$ with a premise occurring in mem, and $select_{sub}(mem, v)$ to determine the leftmost occurrence of v in mem. Note that we include the condition on *match*ing already into the specification of these selection functions. Similarly by instantiating $select_{rule}(ReplacePair, mem)$ the ASM for *Post normal systems* is obtained to yield a pair $(v, w) \in ReplacePair$ with a premise occurring as initial subword of mem, $select_{sub}(mem)$ to determine this initial subword of mem, and by updates of mem which delete the initial subword v and copy w at the end of mem.

4.2 Tree Computations

In this section we model some basic tree computation schemes including language generating grammars like context free, attribute and tree adjoining grammars.

Essentially we show how the notion of tree generation and traversal using a backtracking scheme can be captured by an ASM in such a way that applying to it appropriate data refinements yields well-known logic and functional programming patterns and generative grammars (context free and attribute grammars). For the underlying refinement notion see [16].

Backtracking We define here a BACKTRACK machine which dynamically constructs a tree of alternatives and controls its traversal. When its *ctl_state* which we call here *mode* is $ramify$, it creates as many new children nodes to be computation *candidates* for its *currnode* as there are computation *alternatives*, provides them with the necessary *env*ironment and switches to *select*ion mode. In $mode = select$, if at *currnode* there is no more candidate the machine BACKtracks, otherwise it lets the control move to TRYNEXTCANDIDATE to get *execute*d. The external function *alternatives* determines the solution space depending upon its parameters and possibly the current state. The dynamic function *env* records the information every new node needs to carry out the computation determined by the alternative it is associated with. The macro BACK moves *currnode* one step up in the tree, to $parent(currnode)$, until the *root* is reached where the computation stops. TRYNEXTCANDIDATE moves *currnode* one step down in the tree to the *next* candidate, where *next* is a possibly dynamic choice function which determines the order for trying out the alternatives. Typically the underlying execution machine will update *mode* from *execute* to $ramify$, in case of a successful execution, or to *select* if the execution fails. This model is summarized by the following definition.

BACKTRACK =
  if $mode = ramify$ **then**
    let $k = |alternatives(Params)|$
    let $o_1, \ldots, o_k = new(NODE)$
      $candidates(currnode) := \{o_1, \ldots, o_k\}$
      forall $1 \leq i \leq k$
        $parent(o_i) := currnode$
        $env(o_i) := i\text{-th}(alternatives(Params))$
      $mode := select$
  if $mode = select$ **then**
    if $candidates(currnode) = \emptyset$ **then** BACK **else**
      TRYNEXTCANDIDATE
      $mode := execute$
  where
    BACK =
      if $parent(currnode) = root$ **then** $mode := Stop$
      else $currnode := parent(currnode)$
    TRYNEXTCANDIDATE =
      $currnode := next(candidates(currnode))$
      DELETE$(next(candidates(currnode)), currnode)$

We show now that by pure data refinements BACKTRACK can be turned into the backtracking engine for the core of ISO Prolog [22], of IBM's constraint logic programming language CLP(R) [24], of the functional programming language Babel [20], and also for context free and for attribute grammars [43].

Logic Programming Engine We data refine here BACKTRACK to the backtracking engine for Prolog by instantiating the function *alternatives* to the function $procdef(stm, pgm)$. This is a Prolog specific function which yields the sequence of clauses in *pgm* to be tried out in this order to execute the current goal *stm*; these clauses come together with the needed state information from *currnode*. We determine *next* as *head* function on sequences, reflecting the depth-first left-to-right tree traversal strategy of ISO Prolog. It remains to add the execution engine for Prolog specified as ASM in [22], which switches *mode* to $ramify$ if the current resolution step succeeds and otherwise switches *mode* to *select*.

The backtracking engine for CLP(R) is the same, one only has to extend $procdef$ by an additional parameter for the current set of *constraints* for the indexing mechanism and to add the CLP(R) engine specified as ASM in [24].

The functional language Babel uses the same function *next*, whereas the function *alternatives* is instantiated to $fundef(currexp, pgm)$ yielding the list of defining rules provided in *pgm* for the outer function of *currexp*. The Babel execution engine specified as ASM in [20] applies the defining rules in the given order to reduce *currexp* to normal form (using narrowing, a combination of unification and reduction).

Context-Free and Attribute Grammars To instantiate BACKTRACK for context free grammars G generating leftmost derivations we define $alternatives(currnode, G)$ to yield the sequence of symbols $Y_1, \ldots, Y_k$ of the conclusion of a G-rule whose premisse X labels *currnode*, so that *env* records the label of a node, either a variable X or terminal letter a. The definition of *alternatives* includes a choice between different rules $X \to w$ in G. For leftmost derivations *next* is defined as for Prolog. As machine in $mode = execute$ one can add the following rule. For nodes labeled by a variable it triggers further tree expansion, for terminal nodes it extracts the yield (concatenating the terminal letter to the word generated so far) and moves the control to the parent node to continue the derivation in $mode = select$.

EXECUTE(G) =
  if $mode = execute$ **then**
    if $env(currnode) \in VAR$ **then** $mode := ramify$ **else**
      $output := output * env(currnode)$
      $currnode := parent(currnode)$
      $mode := select$

For attribute grammars it suffices to extend the instantiation for context free grammars as follows. For the synthesis of the attribute $X.a$ of a node X from

its childrens' attributes we add to the else-clause of the BACK macro the corresponding update, e.g. $X.a := f(Y_1.a_1, \ldots, Y_k.a_k)$ where $Y_i = env(o_i)$ for children nodes o_i and $X = env(parent(currnode))$. Inheriting an attribute from the parent and siblings can be included in the update of *env* (e.g. upon node creation), extending it to update also node attributes. The attribute conditions for grammar rules are included into EXECUTE(G) as additional guard to yielding output, of the form $Cond(currnode.a, parent(currnode).b, siblings(currnode).c)$.

In a similar way one can formulate an ASM for tree adjoining grammars, generalizing Parikh's analysis of context free languages by 'pumping' of contex free trees from *basis trees* (with terminal yield) and *recursion trees* (with terminal yield except for the root variable), see [43].

4.3 Structured Programming

In this section we model standard structured programming constructs by natural classes of ASMs. In [25] two operators **seq** and **iterate** have been defined to compose ASMs sequentially and iteratively, capturing these two notions in a black-box view which fits the synchronous parallelism of ASMs, hiding internals of subcomputations by compressing them into one step (so that the resulting machines became known as *turbo ASMs*). This allows one to provide succinct ASMs for standard programming constructs, as we are going to illustrate by turbo ASMs for the celebrated Structured Programming Theorem of Böhm and Jacopini [12], thus showing how to combine the advantages of Gödel-Herbrand style functional and of Turing style imperative programming.

Call *Böhm-Jacopini-ASM* any ASM M which can be defined, using only **seq, while**, from ASMs whose non-controlled functions are restricted to one (a 0-ary) input function (whose value is fixed by the initial state), one (a 0-ary) output function, and the initial functions of recursion theory as static functions. The purpose of the 0-ary input function which we write in_M is to contain the number sequence which is given as input for the computation of the machine. Similarly out_M is used to receive the output of M. The *initial* functions of recursion theory are the following functions from Cartesian products of natural numbers into the set of natural numbers: $+1$, all the projection functions U_i^n, all the constant functions C_i^n and the characteristic function of the predicate $\neq 0$. The **while**-operator can be defined in the usual way from an iteration operator:

$$\textbf{while } (cond)\ R \ = \ \textbf{iterate } (\textbf{if } cond \textbf{ then } R).$$

As usual a number theoretic function $f\colon N^n \to N$ is called *computable by an ASM* M if for every n-tuple $x \in N^n$ of arguments on which f is defined, the machine *started with input* x *terminates with output* $f(x)$. By 'M started with input x' we mean that M is started in the state where all the dynamic functions different from in_M are completely undefined and where $in_M = x$. Assuming the monitored function in_M not to change its value during an M-(turbo) computation, it is natural to say that M 'terminates in a state with output' y, if in this state out_M gets updated for the first time, namely to y. In the machines F we are going to

construct now by induction for every partial recursive function f, the termination state will always be the state in which the intended turbo-computation reached its final goal.

Each initial function f is computed by the machine F of only one function update which reflects the defining equation of f.

$$F \equiv out_F := f(in_F)$$

In the inductive step we construct, for every partial recursive definition of a function f from its constituent functions f_i, a machine F which mimics the standard evaluation procedure underlying that definition. We use the following macros which describe inputting from some external input source in to a machine F before it gets started respectively extracting the machine output upon termination of F to some external target location out. These macros reflect the mechanism for providing arguments and yielding values which is implicit in the standard use of functional equation systems to determine the value of a function for a given argument.

$$F(in) \equiv in_F := in \textbf{ seq } F$$
$$out := F(in) \equiv in_F := in \textbf{ seq } F \textbf{ seq } out := out_F$$

Function Composition If functions $g, h_1, \ldots, h_m$ are computed by Böhm-Jacopini-ASMs $G, H_1, \ldots, H_m$, then their composition f defined by $f(x) = g(h_1(x), \ldots, h_m(x))$ is computed by the following machine $F = \textsc{FctCompo}$ where for reasons of simplicity but without loss of generality we assume that the submachines have pairwise disjoint signatures:

$$\textsc{FctCompo}(G, H_1, \ldots, H_m) =$$
$$\{H_1(in_F), \ldots, H_m(in_F)\} \textbf{ seq } out_F := G(out_{H_1}, \ldots, out_{H_m})$$

Unfolding this structured program reflects the order one *has* to follow for evaluating the subterms in the defining equation for f, an order which is implicitly assumed in the equational (functional) definition. First the input is passed to the constituent functions h_i to compute their values, whereby the input functions of H_i become controlled functions of F. The parallel composition of the submachines $H_i(in_F)$ reflects that their computations are completley independent from each other; what counts and is expressed is that all of them have to terminate before the next 'functional' step is taken. That next step consists in passing the sequence of out_{H_i} as input to the constituent function g. Finally g's value on this input is computed and assigned as output to out_F.

Primitive Recursion Let a function f be defined from g, h by primitive recursion:

$$f(x, 0) = g(x), \quad f(x, y + 1) = h(x, y, f(x, y))$$

and let Böhm-Jacopini-ASMs G, H be given which compute g, h. Then the following machine $F = \textsc{PrimitiveRecursion}$ computes f, composed as sequence

of three submachines. The start submachine evaluates the first defining equation for f by initializing the recursor *rec* to 0 and the intermediate value *ival* to $g(x)$. The *while* submachine evaluates the second defining equation for f for increased values of the recursor as long as the input value y has not been reached. The output submachine provides the final value of *ival* as output. As in the case of simultaneous substitution, the sequentialization and iteration described here make the bare minimum on ordering computational substeps explicit which is assumed and in fact needed in the standard functional use of the defining equations for f.

$\textsc{PrimitiveRecursion}(G, H) \;=\; \textbf{let } (x, y) = in_F \textbf{ in}$
$\quad \{ival := G(x),\ rec := 0\} \textbf{ seq}$
$\quad (\textbf{while } (rec < y)\ \{ival := H(x, rec, ival),\ rec := rec + 1\}) \textbf{ seq}$
$\quad out_F := ival$

Minimalization If f is defined from g by the μ-operator, i.e. $f(x) = \mu y(g(x, y) = 0)$, and if a Böhm-Jacopini-ASM G computing g is given, then the following machine $F = \mu\text{-}\textsc{Operator}$ computes f. The start submachine computes $g(x, rec)$ for the initial recursor value 0, the iterating machine computes $g(x, rec)$ for increased values of the recursor until 0 shows up as computed value of g, in which case the reached recursor value is set as output.

$\mu\text{-}\textsc{Operator}(G) =$
$\quad \{G(in_F, 0),\ rec := 0\} \textbf{ seq}$
$\quad (\textbf{while } (out_G \neq 0)\ \{G(in_F, rec + 1),\ rec := rec + 1\}) \textbf{ seq}$
$\quad out_F := rec$

4.4 Functional Programming (Recursion)

In this section we show how to model basic functional programming constructs by a natural subclass of turbo ASMs. A black-box submachine concept for value returning turbo ASMs has been defined in [25] which abstractly models the standard imperative calling mechanism. Triggered by the question raised in [45]: 'If algorithms are machines, then which machine is the mergesort?', the definition has been applied in [13] for simultaneous calls of multiple submachines, to seamlessly integrate functional description and programming techniques into ASMs. We illustrate this by a natural model for widely used forms of recursion.

The atomic view of an entire turbo ASM computation as one step is rendered by a set $[\![R(a_1, \ldots, a_n)]\!]^A$ of updates produced through executing the turbo ASM call $R(a_1, \ldots, a_n)$ in state A. This set represents the total effect of executing the submachine R in the call state A and is defined by

$$[\![R(a_1, \ldots, a_n)]\!]^A = [\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^A,$$

where the submachine R is declared by $R(x_1, \ldots, x_n) = body$. The characteristic *functional abstraction* consists in abstracting from everything in a computation except the intended input/output relation, for example when using a machine to

return a value and then passing it by value to other machines S. This is easily reflected in turbo ASMs by projecting that value out of the total computational effect $[\![R(a_1, \ldots, a_n)]\!]^A$ and passing it to S via the **let**-construct. Without loss of generality we assume expected output to be stored in a reserved location **result** which the programmer can change to a location l where he wants the expected return value to be transfered. We adopt the standard notation $l \leftarrow R(a)$ to denote the turbo computation outcome $[\![R_l(a)]\!]^A$ where R_l is the result of replacing **result** in R by l, i.e. $R_l = R(l/\mathbf{result})$, so that when the computation is terminated its expected value can be retrieved from the location l. We use a function *new* to provide for each submachine call a fresh location (read: a 0-ary dynamic function, the variables of programming) where to record the result of the subcomputation, given that simultaneous calls—also of the same machine but with different parameters—may yield different results. This explains the following definition.

Definition. Let R_i, S be arbitrary turbo ASMs where R_i may come with formal parameter sequences x_i and S with formal parameter sequences y_i . We define:

let $\{y_1 = R_1(a_1), \ldots, y_n = R_n(a_n)\}$ **in** $S =$
  let $l_1, \ldots, l_n = new(FUNCTION_0)$ **in**
    forall $1 \leq i \leq n$ **do** $l_i \leftarrow R_i(a_i)$
    seq
    let $y_1 = l_1, \ldots, y_n = l_n$ **in** S

The use of turbo ASM return values allows one to explicitly capture the abstract machine(ry) which underlies the common mathematical evaluation procedure for functional expressions, including those defined by forms of recursion. We illustrate this by the following turbo ASM definitions of Quicksort and of Mergesort which exactly mimic the usual recursive definition of the algorithms to provide as *result* a sorted version of any given list. This answers the question raised in [45]: 'If algorithms are machines, then which machine is the mergesort?'

Quicksort The computation suggested by the well-known recursive equations to quicksort L proceeds as follows: *first* partition the *tail* of the list into the two sublists $tail(L)_{<head(L)}, tail(L)_{\geq head(L)}$ of elements $< head(L)$ respectively $\geq head(L)$ and quicksort these two sublists separately (independently of each other), *then concatenate* the results taking $head(L)$ between them. The fact that this description uses various auxiliary list and comparison operations is reflected by the appearance of corresponding auxiliary functions in the following turbo ASM.

QUICKSORT$(L) =$
  if$\mid L \mid \leq 1$ **then result** $:= L$ **else**
    let
      $x =$ QUICKSORT$(tail(L)_{<head(L)})$
      $y =$ QUICKSORT$(tail(L)_{\geq head(L)})$
    in result $:= concatenate(x, head(L), y)$

Mergesort The computation suggested by the usual recursive equations to mergesort a given list L consists in *first* splitting it into a $LeftHalf(L)$ and a $RightHalf(L)$ (if there is something to split) and mergesort these two sublists separately (independently of each other), *then* to $Merge$ the two results by an auxiliary elementwise $Merge$ operation. This is expressed by the following turbo ASM which besides two auxiliary functions $LeftHalf$, $RightHalf$ comes with an external function $Merge$ defined below as a submachine.

MERGESORT(L) =
  if| L |≤ 1 **then result** := L **else**
    let
      x = MERGESORT($LeftHalf(L)$)
      y = MERGESORT($RightHalf(L)$)
    in result := $Merge(x, y)$

Usually also $Merge$ is defined by a recursion, suggesting the following computation scheme which is formalized by the turbo ASM below. If both lists are non-trivial, by a case distinction the smaller one of the two list heads is determined and placed as the first element of the *result* list, concatenating it with the result of a separate and independent $Merge$ operation for the two lists remaining after having removed the chosen smaller head element. The ι-operator in $\iota x(P)$ denotes the unique x with property P (if there is such an x).

MERGE(L, L') =
  if $L = \emptyset$ **or** $L' = \emptyset$ **then result** := $\iota l(l \in \{L, L'\}$ **and** $l \neq \emptyset)$
  elseif $head(L) \leq head(L')$ **then**
    let $x = Merge(tail(L), L')$ **in result** := $concatenate(head(L), x)$
  else
    let $x = Merge(L, tail(L'))$ **in result** := $concatenate(head(L'), x)$

5 System Design Models

In this section we show how to model by ASMs the basic semantical concepts of currently used high-level design languages. We use in this section also the concept of asynchronous multi-agent ASMs, roughly speaking sets of ASMs whose runs are defined by appropriately constrained partial orders to reflect the intended causal dependencies between steps of different machines. The definition can be found in [40] and in [27, Ch.6].

5.1 Executable High-Level Design Languages

We discuss here two major executable high-level design languages of the 90'ies. We relate their characteristic semantical features to ASMs, without mentioning further the important executability aspect which clearly influenced the choice of the language constructs. The languages are UNITY [29] and COLD [33].

UNITY *Unity* computations are sequences of state transitions where each step comprises the simultaneous execution of multiple conditional variable assignments, including quantified array variable assignments of form **forall** $0 \leq i < N$ **do** $a(i) := b(i)$. States are formed by variables (0-ary dynamic functions which may be shared, respecting some naming conventions), conditions are typically formulated in terms of $<, =$, steps are executions of program statements which correspond in a direct way to ASM rules. The steps are scheduled using a global clock (the Unity system time) which synchronizes the system components for an interleaving semantics: per step one statement of one component program in the system is scheduled using non-deterministic schedulers (required to respect a certain fairness condition on infinite runs). (Dijkstra's guarded commands come with the same type of non-deterministic choice of one command per step.) Like in basic ASMs, there is no further control flow. Identifying components with basic ASMs and systems with sets of components leads therefore to the following computational model for Unity systems. Unity comes with a particular proof system, geared to extract proofs from the program text, equipped with appropriately specialized composition and refinement concepts we do not discuss here.

UnitySystem(S) =
  choose $com \in Component(S)$
    choose $rule \in Rule(com)$
      $rule$

COLD In the *C*ommon *O*bject-oriented *L*anguage for *D*esign states are realized as structures, including abstract data types (ADT) linked to an underlying dynamic logic proof system which is geared to provide proofs for algebraic specifications of states and their dynamics (à la Z and VDM). Computations are sequences of state transitions (due to the execution of procedure calls, built from statements viewed as expressions with side effects) allowing synchronous parallelism of simultaneous multiple conditional variable assignments (but no explicit **forall** construct) and non-deterministic choices among variable assignments and rules (procedure invocations). Thus a Cold class (with a set of states, one initial state, and a set of transition relations) corresponds in a standard way to a control state ASM, except that different states of a same class are allowed to have different signatures. The black box view offered for sequencing and iteration is directly reflected by the corresponding turbo ASM constructs, taking into account that Cold provides a separate *guard statement* for blocking evaluation of guards which is executed only (with *skip* effect) when the guard becomes true.

There is an idiomatic high-level construct Mod of Cold which supports non-determinism in choosing subsets of variables to be updated by chosen values. It is modeled by the following ASM.

ColdModify(Var) =
  choose $n \in N$, **choose** $x_1, \ldots, x_n \in Var$, **choose** $v_1, \ldots, v_n \in Value$
    forall $1 \leq i \leq n$ **do** $val(x_i) := v_i$

A similar construct Use permits to choose procedures from a set $Proc$ to be called in sequence.

$$\begin{aligned}&\textsc{ColdUse}(Proc) = \quad \textbf{choose}\ n \in N, \textbf{choose}\ p_1, \ldots, p_n \in Proc\\ &\quad p_1\ \textbf{seq}\ \ldots\ \textbf{seq}\ p_n\end{aligned}$$

5.2 State-based Specification Languages

For sequential state-based specification languages we discuss three representative systems: VDM [34] (denotational), Z [58] (axiomatic), and B [1] (pseudo-code). As representative distributed state-based modeling systems we relate Petri nets [48] to asynchronous ASMs.

VDM, Z, B These three high-level design languages share the notion of computation as sequence of state transitions given by a before-after relation, where states are formed by variables taking values in certain sets (in VDM built up from basic types by constructors) with explicitly or implicitly defined auxiliary functions and predicates. The single (in basic B sequencing-free and loop-free) transitions can be modeled in a canonical way by basic ASM rules which capture also the 'unbounded' as well as the 'bounded' choice and the parallelism B offers in terms of simultaneous ('multiple generalized') substitution. The basic scheme is determined by what Abrial calls the 'pocket calculator model' which views a machine (program) as offering a set of operations (in VDM procedures with side effects) which are callable one at a time, e.g. in the non-deterministic form **choose** $R \in Operation$ **in** R or harnessed by a scheduler **let** $R = scheduled(Operation)$ **in** R; similarly for events which in event-B are allowed to happen only one per time unit.

This view points to a methodological difference between the forces which drove the development of the B method compared to that of the ASM method. Abrial's B method is the result of an engineer's bottom-up analysis: 'The ideas behind the concept of abstract machine have all been borrowed from those ideas that are behind some well-known programming features such as modules, packages, abstract data types or classes' [2, pg.175]. Also the event-B notion of basic events, which corresponds to the guarded update rules of basic ASMs, came out of the concern to 'separate assignments from scheduling'. Gurevich's concept of ASMs is the result of a logician's top-down analysis, brought to light by a mathematical investigation of the ASM thesis (and supported by an extensive experimentation with the concept, see [17], [27, Ch.10] for the historical details).

The structuring mechanisms for large and refined B machines are captured by turbo ASMs, including also the machine state hiding mechanism operations typically come with: it is allowed to activate (call) an operation for certain parameters, which results in an invariant preserving state modification, but besides calling the operation and taking its result no other direct access to the state is granted. Historically, this view has led to a certain bias to functional modeling one can observe for uses of VDM.

By the logical nature of Z specifications, their before-after expressions define the entire system dynamics. In B as in the ASM method, the formulation of the system dynamics—in B by operations (in event-based B by events [2–4]), in ASMs by rules—is separated from the formulation of the static state invariants and of the dynamic run constraints, which express desired system properties one has to prove to hold through every possible state evolution. However for carrying out these proofs, in contrast to the ASM method, there is a fixed link between B and a computer assisted proof system relating syntactical program constructs to proof rules which are used to establish program invariants and dynamic constraints along with the program construction. Thus defining modules becomes intimately related to inventing lemmas. This fits also the basically axiomatic foundation of B as of Z and VDM: VDM by a denotational semantics; Z by axiom systems formulated in (mainly first-order) logic; B by Dijkstra's weakest precondition theory, interpreted in set-theoretic models and based upon the syntactic global concept of substitution (from which local assignment $x := t$ and parallel composition are derived). Differently from Z, which due to the purely axiomatic character of Z descriptions has intrinsic problems to turn specifications into executable code (see [41]), VDM and B are geared to obtain software modules from abstract specifications via refinements which are tailored to the proof rules used for proving that the refined operations satisfy 'unchanged' properties of their abstract counterparts.

Petri Nets The general view of Petri nets is that of distributed transition systems transforming objects under given conditions. In Petri's classical instance the objects are marks on *places* ('passive net components' where objects are stored), the *transitions* ('active net components') modify objects by adding and deleting marks on the places. In modern instances (e.g. the predicate/transition nets) places are locations for objects belonging to abstract data types (read: variables taking values of given type, so that a marking becomes a variable interpretation), transitions update variables and extend domains under conditions which are described by arbitrary first-order formulae. The distributed nature of Petri nets is captured by modeling them as asynchronous ASMs, associating to each transition one *agent* to execute the transition. Each single transition is modeled by a basic ASM rule of the form defined below, where pre/post-places are sequences or sets of places which participate in the 'information flow relation' (the local state change) due to the transition and *Cond* is an arbitrary first-order formula. By modeling Petri net states as ASM states we include the abstract Petri net view proposed in [48] where states are interpreted as logical predicates which are associated to places and transformed by actions.

PetriTransition = **if** $Cond(prePlaces)$ **then** $Updates(postPlaces)$
where $Updates(postPlaces)$ = a set of function updates

Virtual Machines Virtual machines by definition are machines. Typically they work over a specific set of states, appropriate to the specific purpose. Thus

they 'are' particular ASMs. In fact for design or analysis purposes numerous virtual machines have been explicitly modeled as ASMs, e.g. the Warren Abstract Machine [23] and its extensions [8, 24, 10, 9, 7], the Transputer [19], the RISC machine DLX [21], the Java Virtual Machine [53], the Neural Net (abstract data flow) Machine [26], the UPnP architecture [37]), etc.

5.3 Logic Based Modeling Systems

There is a myriad of logic-based and algebraic specification and 'declarative programming' languages and calculi, like Prolog and its numerous variants, VDM, Z, structural operational or natural semantics systems, process algebra languages like CSP, LOTOS, innumerable 'logics of programs', dynamic logics, temporal logics, rewriting logics, offering proof calculi to support verification of program properties, etc. These approaches have the pattern of logic in common: specifications are typically expressed by systems of equations (with fixpoint operators to solve equations) or of general axioms and inference rules, so that they all are exposed to the frame problem and the difficulty to control the order of inference rule applications. Most of these systems are not conceived to serve general-purpose specifications but are tailored to specific goals, the way Plotkin's *Structural Operational Semantics* [36] or Kahn's *Natural Semantics* or Mosses' Action Semantics [46] are tailored for dealing with the semantics of programming languages. Numerous of these approaches are driven by structural patterns where the syntax dictates the principles of compositionality. Since this is not the place to evaluate the advantages or disadvantages of such often stateless approaches with respect to state-based transition systems, we limit ourselves to observe that the ASM method allows one to use such logic-based design and verification techniques *where appropriate*—desired, technically feasible and cost-effective—, integrating them into the high-level but state-based, genuinely semantical and computation oriented specification and analysis techniques which are possible with ASMs. Successful projects in this direction have been reported using theorem proving systems (KIV, PVS, Isabelle) and model checkers, see e.g. [47, 38, 50, 59, 31, 30, 51, 49, 35] and [55–57] for details.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial. Extending b without changin it (for developing distributed systems). In H. Habrias, editor, *1st Conference on the B Method*, number ISBN 2-906082-25-2, pages 169–190, 1996.
3. J.-R. Abrial and L. Mussat. Specification and design of a transmission protocol by successive refinements using b. In M. Broy and B. Schieder, editors, *Mathematical Methods in Program Development*. Springer, 1996.
4. J.-R. Abrial and L. Mussat. Introducing dynamic constraints in b. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *LNCS*, pages 82–128. Springer, 1998.

5. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
6. M. Anlauff, S. Chakraborty, P. Kutter, A. Pierantonio, and L. Thiele. Generating an Action Notation environment from montages descriptions. *Software Tools and Technology Transfer, Springer*, 3:431–455, 2001.
7. C. Beierle. Formal design of an abstract machine for constraint logic programming. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 377–382, Elsevier, Amsterdam, the Netherlands, 1994.
8. C. Beierle and E. Börger. Correctness proof for the WAM with types. In E. Börger, G. Jäger, H. Kleine Büning, and M. M. Richter, editors, *Computer Science Logic*, volume 626 of *LNCS*, pages 15–34. Springer-Verlag, 1992.
9. C. Beierle and E. Börger. Refinement of a typed WAM extension by polymorphic order-sorted types. *Formal Aspects of Computing*, 8(5):539–564, 1996.
10. C. Beierle and E. Börger. Specification and correctness proof of a WAM extension with abstract type constraints. *Formal Aspects of Computing*, 8(4):428–462, 1996.
11. A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 3, 2002.
12. C. Böhm and G. Jacopini. Flow diagrams, Turing Machines, and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.
13. T. Bolognesi and E. Börger. Remarks on turbo asms for computing functional equations and recursion schemes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003*, volume xxx of *LNCS*. Springer, 2003.
14. E. Börger. *Computability, Complexity, Logic (English translation of* Berechenbarkeit, Komplexität, Logik , volume 128 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1989.
15. E. Börger. High level system design and analysis using abstract state machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.
16. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 14, 2002.
17. E. Börger. The origins and the development of the ASM method for high level system design and analysis. *J. of Universal Computer Science*, 8(1):2–74, 2002.
18. E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000 Proceedings*, volume 1816 of *LNCS*, pages 293–308. Springer-Verlag, 2000.
19. E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.
20. E. Börger, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A model for mathematical analysis of functional logic programs and their implementations. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 410–415, Elsevier, Amsterdam, the Netherlands, 1994.
21. E. Börger and S. Mazzanti. A practical method for rigorously controllable hardware design. In J. P. Bowen, M. B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer-Verlag, 1997.
22. E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24:249–286, 1995.

23. E. Börger and D. Rosenzweig. The WAM — definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 2, pages 20–90. North-Holland, 1995.
24. E. Börger and R. Salamone. CLAM specification for provably correct compilation of CLP($\mathcal{R}$) programs. In E. Börger, editor, *Specification and Validation Methods*, pages 97–130. Oxford University Press, 1995.
25. E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 41–60. Springer-Verlag, 2000.
26. E. Börger and D. Sona. A neural abstract machine. *J. of Universal Computer Science*, 7(11):1007–1024, 2001.
27. E. Börger and R. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
28. A. Brüggemann, L. Priese, D. Rödding, and R. Schätz. Modular decomposition of automata. In E. Börger, G. Hasenjäger, and D. Rödding, editors, *Logic and Machines: Decision Problems and Complexity*, number 171 in LNCS, pages 198–236. Springer, 1984.
29. K. M. Chandy and J. Misra. *Parallel Program Design. A Foundation*. Addison Wesley, 1988.
30. A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, July 1998.
31. A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-based mechanized verification of compiler back-ends. In U. Glässer and P. Schmitt, editors, *Proceedings of the Fifth International Workshop on Abstract State Machines*, pages 50–67. Magdeburg University, 1998.
32. S. Eilenberg. *Automata, Machines and Languages Vol.A*. Academic Press, 1974.
33. L. M. G. Feijs and H. B. M. Jonkers. *Formal Specification and Design*. Cambridge University Press, 1992.
34. J. Fitzgerald and P. G. Larsen. *Modelling Systems. Practical Tool and Techniques in Software Development*. Cambridge University Press, 1998.
35. A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 303–322. Springer-Verlag, 2000.
36. G.D.Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN19, Department of Computer Science at University of Aarhus, 1981.
37. U. Glässer, Y. Gurevich, and M. Veanes. High-level executable specification of the universal plug and play architecture. In *Proceedings of 35th Hawaii International Conference on System Sciences — 2002*, pages 1–10. IEEE Computer Society Press, 2002.
38. W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler correctness and implementation verification: The verifix approach. In P. Fritzson, editor, *International Conference on Compiler Construction*, volume Proceedings of the Poster Session of CC'96, IDA Technical Report LiTH-IDA-R-96-12, Linköping/Sweden, 1996.
39. Y. Gurevich. A new thesis. *Abstracts, American Mathematical Society*, page 317, August 1985.
40. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

41. J. A. Hall. Taking Z seriously. In *ZUM'97*, volume 1212 of *Springer LNCS*, 1997.
42. J. W. Janneck. *Syntax and Semantics of Graphs*. PhD thesis, ETH Zürich, 2000.
43. D. E. Johnson and L. S. Moss. Grammar formalisms viewed as Evolving Algebras. *Linguistics and Philosophy*, 17:537–560, 1994.
44. L. Lavagno, A. Sangiovanni-Vincentelli, and E. M. Sentovitch. Models of computation for system design. In E. Börger, editor, *Architecture Design and Validation Methods*, pages 243–295. Springer, 2000.
45. Y. N. Moschovakis. What is an algorithm? In B. Engquist and W. Schmid, editors, *Mathematics Unlimited—2001 and beyond*. Springer, 2001.
46. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
47. C. Pusch. Verification of compiler correctness for the WAM. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 347–362. Springer-Verlag, 1996.
48. W. Reisig. *Elements of Distributed Algorithms*. Springer, 1998.
49. G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Universität Ulm, 1999.
50. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. of Universal Computer Science*, 3(4):377–413, 1997.
51. G. Schellhorn and W. Ahrendt. The wam case study: Verifying compiler correctness for prolog with kiv. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*. Kluwer, 1998.
52. D. Scott. Definitional suggestions for automata theory. *J. Computer and System Sciences*, 1:187–212, 1967.
53. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001. .
54. P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40:80–91, 1997.
55. K. Winter. Model checking for Abstract State Machines. *J. of Universal Computer Science*, 3(5):689–701, 1997.
56. K. Winter. Towards a methodology for model checking ASM: Lessons learned from the flash case study. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 341–360. Springer-Verlag, 2000.
57. K. Winter. Automated checking of control tables. E-mail to E. Börger, December 24, 2001.
58. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996.
59. W. Zimmerman and T. Gaul. On the construction of correct compiler back-ends: An ASM approach. *J. of Universal Computer Science*, 3(5):504–567, 1997.

Recent BRICS Notes Series Publications

NS-02-8 **Peter D. Mosses, editor.** ***Proceedings of the Fourth International Workshop on Action Semantics, AS 2002,*** **(Copenhagen, Denmark, July 21, 2002), December 2002. vi+133 pp.**

NS-02-7 **Anders Møller.** ***Document Structure Description 2.0*****. December 2002. 27 pp.**

NS-02-6 **Aske Simon Christensen and Anders Møller.** ***JWIG User Manual*****. October 2002. 35 pp.**

NS-02-5 **Patrick Cousot, Lisbeth Fajstrup, Eric Goubault, Maurice Herlihy, Martin Raußen, and Vladimiro Sassone, editors.** ***Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency Theory, GETCO '02,*** **(Toulouse, France, October 30–31, 2002), October 2002. vi+97.**

NS-02-4 **Daniel Gudbjartsson, Anna Ingólfsdóttir, and Augustin Kong.** ***An BDD-Based Implementation of the Allegro Software*****. August 2002. 2 pp.**

NS-02-3 **Walter Vogler and Kim G. Larsen, editors.** ***Preliminary Proceedings of the 3rd International Workshop on Models for Time-Critical Systems, MTCS '02,*** **(Brno, Czech Republic, August 24, 2002), August 2002. vi+141 pp.**

NS-02-2 **Zoltán Ésik and Anna Ingólfsdóttir, editors.** ***Preliminary Proceedings of the Workshop on Fixed Points in Computer Science, FICS '02,*** **(Copenhagen, Denmark, July 20 and 21, 2002), June 2002. iv+81 pp.**

NS-02-1 **Anders Møller and Michael I. Schwartzbach.** ***Interactive Web Services with Java: JSP, Servlets, and JWIG*****. April 2002. 99 pp.**

NS-01-8 **Anders Møller and Michael I. Schwartzbach.** ***The XML Revolution (Revised)*****. December 2001. 186 pp. This revised and extended report superseeds the earlier BRICS Report NS-00-8.**

NS-01-7 **Patrick Cousot, Lisbeth Fajstrup, Eric Goubault, Jeremy Gunawardena, Maurice Herlihy, Martin Raußen, and Vladimiro Sassone, editors.** ***Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency Theory, GETCO '01,*** **(Aalborg, Denmark, August 25, 2001), August 2001. vi+97 pp.**